Applied Software Visualisation:

Active and Continuous Synchronisation of Code and Requirements in Agile Development Processes

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY (PhD)

Supervisors

Professor Stephen G. MacDonell, and Jim Buchan

April, 2023

By

Mujtaba Alshakhouri

School of Engineering, Computer and Mathematical Sciences

Attestation of Authorship

I declare that the work presented in this thesis is that of me alone, except where explicitly acknowledged, and that to the best of my knowledge and belief, it contains no material previously published or written by another person, nor material which has been submitted for the award of any other degree or diploma of a university of other institution of higher learning, and that this thesis is the result of work carried out since the official commencement date of the approved research program.

Mujtaba Alshakhouri
Auckland, New Zealand

1 September, 2022

Abstract

Software engineering has long been facing a challenge of ever-increasing product/process complexity due to the large numbers of interdependent components that developers need to work with on a daily basis. Making sense of how those components relate, interact, and affect each other is required before developers can optimally fulfil a task at hand. As a result, researchers report that up to 50% of developers' time is used—and perhaps wasted—on trying to figure such things out. This problem is further exacerbated by the fact that documentation and design requirements typically exist in a separate and disconnected state from their actual implementations, making it harder for developers to access, form, and understand the necessary associations and their implications for work.

Software visualisation has been acknowledged as offering highly promising potential in presenting visual representations that allow software practitioners to approach their systems with reduced complexity—with many empirical works demonstrating advantages gained across varying sets of tasks and contexts. However, real-world software practitioners have not yet benefited from these advantages, and the field is facing an increase in calls for work that is "applied" to specific and real-world usage contexts. There is increasing focus on making software visualisation more practical, and more accessible to those who need it. For example, 'integration into development environments' has been a recurring topic for the past few years in the field's top specialised research venues.

The research reported in this thesis recognises the current disconnection between software visualisation research and practitioners in the real world. It employs a design science research approach that incorporates the end user in the design and development process from an early stage, with a sequence of iterative feedback-design loops. It argues that this is important to better align the resulting research artefact with the actual needs, environments, and ways of working of intended users. It further applies the technology to address important and outstanding issues in software engineering.

This research contributes to the discipline by reporting on the processes followed, lessons learned, and the findings of a three-stage evaluation process with expert users. It further introduces a promising application of the technology to the software traceability problem, offering users the benefit of an advanced metaphorical software visualisation within the highly popular Vscode platform. It introduces the first language-agnostic visualisation approach, demonstrated over 15 open-source multi-language projects. Lastly, it introduces live and continuous synchronisation of design artefacts existing on agile dashboards with their actual code artefacts in the development environment, allowing users to access both side by side and so form and understand those important associations on the spot.

Acknowledgements

I am immensely grateful to my two supervisors, Stephen MacDonell and Jim Buchan. This work would not have seen light without the incredible support from you both—at personal and academic levels. The journey was especially arduous at times with personal challenges. Thank you, Steve, for your unceasing support, for believing and trusting in me—especially during those very difficult times—and for offering me the opportunity from the start. Thank you for all the time and efforts you invested to help, support, and inspire me in this journey. I would not have been here without it. Thank you, Jim, for your continuous encouragements, stimulation of ideas and thoughts, for those late in the day stand-up talks while you miss your bus, and for all the time you have dedicated to help and support me. Thank you for always checking on, for your uplifting jokes, and for making the lab feel home.

I would also like to thank all my colleagues and peers at the SERC laboratory for their support and encouragement over the years, for their motivating discussions, and for keeping the lab always a happy place to be. A special thank you for all the participants who have generously dedicated considerable time of their workdays to take part in the demonstration and interviews—some of whom went far beyond expectation by repeating their participation in all three phases of the evaluation over the years. A big thank you to Mona Rahimi and Tudor Gîrba for taking the time to happily answer my emails and queries, and offering helpful directions. Thank you all very much.

I would also like to extend my gratitude and warm thanks to my two examiners, Prof. Alexandre Bergel and Dr Craig Anslow. Thank you both for willing to examine my thesis, and for taking the time to read it. Thank you for the humbling feedback and for your invaluable suggestions and recommendations to strengthen the thesis.

This work would not have also been possible without the work of many other prior researchers, whose work greatly inspired, motivated, and informed this one, and whose work appears repeatedly throughout this research. This is a continuation of your efforts.

This work was made possible by a Vice Chancellor's Doctoral Scholarship from Auckland University of Technology, which I am greatly honoured and grateful to have received. Thank you, AUT!

I would also like to thank my parents who throughout their many hardships and sacrifices made it possible for me to pursue my higher education, and supported me in every step.

To my kids, Reima, and Ali, who the circumstances had us living afar from each other for yet an unknown stretch of time, and to my daughter, Mariam. I love you all. Thank you for your patience, sacrifices, and love.

List of Tables

List of Figures

List of Code Blocks

Glossary of Terms

SV	Software Visualisation.
Design Concept	a mental construct embodying a form of software requirement or specification that at a given point in time might be actioned by developers and turned into a working piece of source code. The term is particularly used in conceptual discussions.
Design Artefact	a Design Concept that has materialised into a written or documented software requirement or specification.
Design Item (DI)	an alternative alias of Design Artefact used especially in user-facing places as a relaxed and a more familiar term addressing software practitioners.
Code Artefact	a well-demarcated piece of source code written in fulfilment to a Design Artefact or in testing to its implementation.
Code Item (CI)	an alternative alias of Code Artefact used especially in user-facing places as a relaxed and a more familiar term addressing software practitioners.
SE	Software Engineering.
Viewlet	A user interface (UI) component typically used as a dynamic side panel in an application to display contextual information to the user
Vscode	Microsoft Visual Studio Code Application (an Integrated Development Environment).
Agile Dashboard	An electronic project and software development management dashboard that is advocating agile methodology
TTO	Automated Three-legged Tagging Operation, or alternatively, Automated Three-legged Tracelink Operation

Table of Contents

Chapter 1
INTRODUCTION

About this Research

This work is driven by a desire to bring the benefits of software visualisation research to software practitioners in the outside world. It works towards accomplishing this by developing and deploying visualisation technology to a specific area of practice, to directly support practitioners in some of their real-world work tasks where the advantages and capabilities of this technology are thought to be most needed and useful. It seeks to bring the technology closer to the user by designing a solution around their actual needs *as expressed by them*, and by tailoring it around their standard operational environments and their ways of working. A key novelty of the work stems from the concept of synchronising design artefacts with their implementation code artefacts—a concept whose origins are rooted in our earlier work, and that is now developed to an operational level taking advantage of state-of-the-art technology. The result is a tool that is potentially accessible to a wider spectrum of software practitioners, bringing them the benefits of software visualisation regardless of the programming language they use, or the development practice they adopt.

About Software Visualisation

Software visualisation is a relatively recent area of software engineering research that is primarily concerned with the utilisation of visualisation techniques to ease the craft of software development. Work in this area tends to employ advanced metaphorical visualisations to cognitively support software practitioners, who normally have to work with large-scale complexities and have to constantly navigate their way through large numbers of interconnected components. Despite resulting in many successful and prominent solutions, the discipline's innovations and outputs are described by researchers as largely confined to academic circles, with research tools produced hardly making it to actual practitioners in the outside world. This research attempts to work towards alleviating this issue by embracing a design science approach that consciously and intentionally incorporates the intended users in its process.

1.1 Background and Context

Research in Software Visualisation (SV) has over the past decade established itself as a robust and promising domain of scientific inquiry among the wider discipline of software engineering (SE). The field has quickly evolved from early attempts at conceiving and experimenting with a spectrum of metaphorical concepts—exploring their feasibility and efficacy—to the current, more mature position of exploring techniques and methods of how this technology can actually be employed to benefit the software development community. Instead of approaches described as supporting a multitude of user tasks and roles, researchers are now increasingly calling upon peer colleagues to target specific applications of use and user roles (Chotisarn et al. 2020). Their argument is that, in order to promote the technology among real practitioners, approaches and tools must be tailored around the environment of those users and around their more specific needs (Paredes, Anslow, and Maurer 2014).

In fact, the most outstanding challenge currently facing software visualisation—as expressed by recent research and in literature reviews (Leonel Merino et al. 2019; L. Merino et al. 2018; Baum et al. 2017; A.-L. Mattila et al. 2016; Seriai et al. 2014)—is its weak (or rather absent) adoption by the SE community and industry. The most often cited reason behind SV failing to reach its intended users is the lack of sufficient empirical demonstration of its utility and efficacy. While this has certainly been true for a decade or more, the situation with regard to empirical support has been steadily improving across recent years, as is evident by examining recent publications, as well as being reported by researchers conducting recent literature reviews (Chotisarn et al. 2020; Bedu, Tinh, and Petrillo 2019). We argue—in addition—that the failure of software visualisation technology to reach the practicing community stems from the fact that minimal prior research has focused on studying the organisational and application context. Very few research endeavours, if any, have first investigated the actual needs of users to understand how and where the technology could be put to effective use before designing their approaches. Instead, large numbers of devised approaches and solutions appear to be the result of exploratory research and exercises that have succeeded in producing a viable technique, only to start then conceiving of potential uses. Merino et al. express similar findings in their 2018 paper, stating “we observed that a main threat in software visualization is the disconnect between the development concerns that are the focus of visualization, and the most complex and frequent problems that arise during real-life development” (L. Merino et al. 2018). As a result, those authors recommended researchers to conduct surveys to collect software visualisation requirements, before they propose an approach that utilises visualisation techniques to address those requirements or problems. They expressed their position strongly emphasising “the compulsion of identifying a concrete real-world case”. Paredes et al. also reached a similar conclusion, recommending the conduct of observational

studies to better understand users' real needs prior to developing solutions (Paredes, Anslow, and Maurer 2014).

We thus perceive a rising demand among the research community to operationalise software visualisation and make it more practical and more accessible to the development community. This is also evident in research venues that specialise in the topic, such as IEEE's Working Conference on Software Visualization (VISSOFT) and the Special Issue on software visualisation of the Information and Software Technology Journal. Both venues appear emphasise in their latest issues on "applying" the technology to specific areas of software engineering, motivating works on "task-specific" support for software engineering tasks[1], "Integration of software visualization with development environments"[2], and supporting "development activities", "design", and "requirements"[3].

What is Software Visualisation? Before elaborating further with this work, it is important to formally define or delineate the software visualisation discipline. Software Visualisation is popularly identified among the SE research community as a subfield of ***information visualisation*** (Balzer et al. 2004; Diehl 2007b)—placing it as primarily dealing with "abstract and intangible data" as opposed to "physical objects", the latter of which is ascribed to the *Scientific Visualisation* field. The field is characterised by a number of varying definitions, the most cited of which are traceable back to those appearing in Stephan Diehl's 2007 book (or variations of them). In his seminal book, Diehl presented multiple definitions, the most widely known and circulated of which is where he refers to SV as "visualizing the structure, behavior, and evolution of software" (Diehl 2007a). In fact, that statement was only a closing remark where he was summing up the problem domains addressed by the field. In this work, we adopt—and pragmatically prefer—his original full definition in which he states: "*we define software visualization as the visualization of artifacts related to software and its development process (a wide definition)*". He further adds: "*In addition to the program code, these artifacts include requirements and design documentation, changes to the source code, and bug reports, for example*". Unfortunately, this definition is hardly reproduced in today's literature, despite offering a more accurate and more encompassing view.

In our earlier work we noted that the focus on the (infamous) former definition appears to have inadvertently stifled the research landscape of software visualisation, prompting the majority of researchers to focus on the three referenced aspects, leaving the 'development process', and the other artefacts such as requirements and documentation, to receive little to no attention (Alshakhouri 2013).

[1] Cfp 2018 for both venues
[2] Cfp 2018, 2021 and 2022 for both venues
[3] Cfp 2021 and 2022 for both venues

Mattila et al. expressed a similar sentiment in their 2016 literature review (A.-L. Mattila et al. 2016). Attention to the development process and its constituent activities thus started to appear only recently. Moreover, and as will be discussed fully in Chapter 2, the majority of those studies utilise node graph visualisation or other diagrammatic techniques (A.-L. Mattila et al. 2016), that do not capitalise on the full potential offered by advanced metaphorical techniques. The addressing of design and requirements—and the other artefacts—is either very rare or hardly existing, if at all (A.-L. Mattila et al. 2016; Chotisarn et al. 2020).

In summary, and in contrast to the majority of studies carried out in software visualisation, the research reported in this thesis is presented as investigating visualisation technology in the context of the *software development process and its artefacts* (mainly design and implementation).

Current Directions in Applied Software Visualisation Research

By examining recent research efforts in the software visualisation community a number of recognisable trends towards applicability and usage can be uncovered that come to shape the current areas of research interest. These trends can be identified as:

1) Expanding the reach of software visualisation (new application domains, new utilisation approaches)
2) Enhancements or new approaches targeting the three traditional application areas (software static structure, software evolution, and program behaviour).
3) Exploitation of new technological advancements (hardware/software)
4) Efforts towards better adoption by the development community and industry

The work reported here can be categorised as primarily concerned with the first and forth areas in terms of its goals, while utilising the third area to work towards achieving those goals. It uses visualisation of the software static structure as a baseline (groundwork). Work on software evolution and program behaviour are not addressed in this work.

Drawing on recent examples of research can also help to put the field into context. Under the first category above, we see Ardigo et al. present work to extend the city metaphor to represent data stored on databases and other storage files (Ardigo et al. 2021), while Weninger et al. perform a similar extension to visualise software memory (Weninger, Makor, and Mossenbock 2020). Under the second category, Dashuber et al. extends the city metaphor to visualise program runtime behaviour (Dashuber and Philippsen 2021), while Heidmann introduces a new virtual reality metaphor to study the evolution of software architecture (Heidmann et al. 2020). In the third category, Limberger et al. present a

number of advanced metaphors and techniques that feature unique visual aesthetics, and discuss their suitability and expressiveness for different software tasks (Limberger et al. 2019). Seipel et al. introduce, on the other hand, speech recognition and natural language processing as an alternative mechanism to interact with software visualisation worlds in augmented reality (Seipel et al. 2019). Under the last category, we identify the recent work of Merino et al. who curated a catalogue of 70 available software visualisation tools in an effort to help practitioners discover suitable tools for their particular concerns (Leonel Merino et al. 2019).

Research Groups Influencing/Inspiring this Work

Before moving on from this section, it may be useful to introduce some of the active research groups in the field whose work has inspired, informed, or influenced this research[4]:

- REVEAL (Reverse Engineering, Visualization, Evolution Analysis Lab) group, University of Lugano: are prominent contributors to the advancement of the software visualisation field, and are behind several key and influential research efforts. Their primary domain of research sets around the realms of software comprehension, evolution, as well as reengineering. Key Author: Michele Lanza.
- Computer Graphics Systems Group—the Hasso-Plattner Institute: stemming from a primarily computer graphics and multimedia domain, this group has over the past decade added prominent contributions to the software visualisation field by employing advancements in the graphics realm to benefit the SV field as well as the wider field of information visualisation. The group describes itself as promoting Human Computer Communication. Key Authors: Jürgen Döllner and Daniel Limberger.
- Software Visualization In 3D And Virtual Reality Group, Leipzig University: this group is contributing strongly to the promotion of SV solutions to the software industry and the larger SE community by investigating some practicality aspects, and creating tools aimed at bringing the technology closer to the user. The group is notable for their work in automatic generation of software visualisations across the three traditional aspects (behaviour, structure, and evolution). Key Author: Ulrich Eisenecker.
- (ISCLab) Intelligent Software Construction Laboratory[5], University of Chile: is behind the Roassal Agile Visualisation Engine (in collaboration with other groups) and is primarily concerned with the betterment of data visualisation techniques, as well as support of software development—part of which is the utilisation of software visualisation technology. Key Author: Alexandre Bergel.
- PALUNO - the Ruhr Institute for Software Technology (Visualisation division): this group specialises in a broad range of visualisation research (Information Visualisation, Software Visualisation, and Visual Analytics). Of particular interest is their innovative exploration of new techniques, and utilisation of technology, to create practical new applications of SV that serve the wider SE discipline. Key Author: Fabian Beck.
- Software Composition Group, University of Bern: not particularly specialised in software visualisation approaches or techniques but are notable for their efforts in making existing software visualisation technology more accessible to their target users, by studying and developing better communication approaches and investigating impeding factors. Key Author: Oscar Nierstrasz.

[4] The order is not to be taken to reflect any significance.

[5] Previously, known as the Pleiad research laboratory.

The above list is by no means complete, as there are a number of recognised key authors whose work has influenced or informed ours, but these are not necessarily identified as part of a particular group. Moreover, part of this work is also informed by research from the software traceability discipline, with the works of research groups led by Jane Cleland-Huang and Barbara Paech visibly guiding our work. These and other relevant works are examined in detail in Chapter 2.

1.2 Motivation

This work is driven by potentiality across four problem/opportunity landscapes. This section presents and discusses these landscapes in order to highlight their importance and relevance to the SE research community, as well as to industry and the practicing community—arguing in the process for the research opportunities they present. The anticipated scope of the work's contributions is also outlined for each area. Elaboration and further technical details of each are covered in Chapter 2, and partially in Chapter 4.

Software Visualisation—reaching towards practitioners

The value of visualisation as a means of delivering and communicating information is well-established in essentially all scientific disciplines. Its utility, whether from a cognitive aspect or that of facilitating a shared and perceptible visual imagery, is therefore rarely debated—if at all—in scientific or engineering disciplines. Its power to create a common basis of mental concepts among a working team is generally accepted as invaluable (van Wijk 2005; Oppl 2017; Roberts et al. 2018).

While in the early days of the field this debate, questioning the value of visualisation and its utility, did indeed exist among the SE research community, such debate would hardly receive any considerable attention nowadays after several studies have unequivocally established the enabling power of software visualisation—both via sound empirical work (Wettel, Lanza, and Robbes 2011; Fittkau, Krause, and Hasselbring 2017), as well as from a cognitive aspect (Duru, Çakir, and Işler 2013; Petre, Blackwell, and Green 1998; Petre 2010; Storey, Fracchia, and Muller 1999). More empirical evidence also keeps steadily appearing in the latest research works (Moreno-Lumbreras et al. 2021; Mehra et al. 2020; Steinbeck, Koschke, and Rudel 2019). Given the ever-increasing size and complexity of the intangible artefacts that software practitioners have to deal with, it would be surprising to question that visualising those artefacts and their interconnected relationships would bring significant benefit to the user. In fact, it is largely this specific characteristic of the technology—bringing to the surface what is otherwise intangible and invisible—that lends it its unique capacity in delivering a wide spectrum of advantages and uses that benefit from cognitive scaffolding. Indeed, the potential value of visualisation as a technology for the software engineering discipline has acquired consensus among researchers, and rather than seeking empirical evidence of the benefits and advantages of this technology to establish its core value, what is of interest to researchers nowadays is acquiring empirical evidence for the *individual* approaches (or techniques) that are being brought to the stage (Chotisarn et al. 2020; Leonel Merino, Ghafari, and Nierstrasz 2018). The present overarching objective is to create a rigorous foundation where the utility and efficacy of these solutions can be promoted inside the SE research

community, as well as to general practitioners in the development community. This subject is further discussed in the last point of this section.

Unfortunately, despite this well-established potential, the benefits and advantages of software visualisation have hardly been realised among the actual intended users in the practising community—as was introduced in an earlier section. The problem is not a shortage of tools available. In fact, a large number of tools have been produced by researchers in the field across a number of years (Shahin, Liang, and Babar 2014; Salameh and Aljammal 2016; Leonel Merino et al. 2019; Chotisarn et al. 2020; Cruz et al. 2016), and a good number of those tools are available (Leonel Merino et al. 2019; Chotisarn et al. 2020). There are even prominent autonomous platforms such as the MOOSE platform (Nierstrasz, Ducasse, and Girba 2005; Fabry et al. 2014) and the more recent Glamorous Toolkit (Girba and Chis 2015) that both offer rich visual analysis and exploration capabilities for software systems (as well as of data in general). However, the majority of these existing tools appear to be confined to corridors of academia and are hardly reaching the general practitioners. We argue that one reason for this is that many of these tools are disconnected from the popular toolsets used by practitioners, and were not from the start oriented towards those developers, but rather towards research purposes.

An objective of the research reported here is thus to work towards addressing this problem, by designing a solution around both a popular development environment and a familiar technology stack, to ensure its reachability and accessibility to the user. Another issue is that the majority of these tools are built around the object-oriented framework and Java in particular (Seriai et al. 2014; Leonel Merino, Ghafari, and Nierstrasz 2016). For example, in the recent investigation of the city metaphor usage by Jeffery, we can observe that Java is clearly predominant, with 9 tools out of 14 surveyed targeting the language (Jeffery 2019). On the other hand, C++ appeared three times only, and C# appeared once. However, latest statistics[6] reveal a rising number of other programming languages that are hardly, if at all, addressed by visualisation research and its existing tools. For example, we were able to identify only a single work that each addressed JavaScript (Viana, Hora, and Valente 2017), and C# (Khaloo et al. 2017) in their studies. As importantly, almost all works are only able to visualise systems in a single language, whereas present software projects are hardly written in a single language, and in fact often comprise of a set of languages at the same time. Hence this work is also motivated to work towards an approach and a tool that are able to visualise source code in an agnostic manner, irrespective of its programming language or paradigm.

[6] https://insights.stackoverflow.com/survey/2021

Synchronising Design Artefacts with their Implementation

A core concept underpinning this research leverages the novel idea of linking together implementation source code with the original design artefacts that the implementation is fulfilling. It then presents a synchronised visualisation where both 'worlds' of artefacts are tightly linked, enabling the user to visually navigate and explore both worlds simultaneously in a novel manner. We look at these key artefacts as the building blocks of software development, but they normally exist separately. Visualising them in a synchronised manner appears to have promise in supporting developers in some of their most complex tasks. Early evidence for this has been found and reported in our earlier work (Alshakhouri, Buchan, and MacDonell 2018). Moreover, by integrating such a visualisation into developers' commonly used toolset and environment, and into their way of working, further opportunities emerge for supporting some of their daily tasks. This includes: feature location, artefact traceability, refactoring, and change impact analysis as primary benefits, with secondary benefits such as promoting awareness and team communication. Chapter 4 discusses this subject in more detail from a conceptual perspective, while Chapter 7 and Chapter 8 presents detailed evaluations of its implementations.

Earlier research that bears some similarity to the above includes the work of Delater and Paech (2013), Rubasinghe, Meedeniya, and Perera (2018) and Cleland-Huang et al. (2021)—which are elaborated on in Chapter 2. These works are different from that reported here in that they require a specific data model for their artefacts. They are also based on establishing tracelinks in a post-event manner by applying advanced algorithms to reconstruct (or extract) them from existing data sources—with the exception of the work of Delater and Paech who only partially incorporate that method. The current work is motivated from a different perspective where the goal is to actively unify both worlds of artefacts (design and implementation) by capturing the knowledge directly from developers in real-time while the development is happening. Establishing tracelinks is hence only an advantageous by-product, rather than a principal goal in our work. In addition, this work introduces a flexible approach to capture and link artefacts at multiple granularities and irrespective of their type or nature. This is covered in detail in Chapter 4.

Artefact Traceability—a promising benefit for the agile practicing community

While software traceability is a well-established sub-discipline, the area of agile development has as yet received little attention from the traceability research community (Hess, Diebold, and Seyff 2018; Curcio et al. 2018). For example, in their 2022 paper, Maro et al. state that "existing literature lacks concrete guidelines for practitioners to systematically define such a strategy", in reference to the lack of frameworks for establishing traceability during the agile development process (Maro et al. 2022).

This appears to be due to the fact that the majority of efforts had predominantly been directed at perfecting algorithmic-based retrieval and rediscovery methods to build the links in a post-event manner, more than at developing solutions that establish the traceability proactively during development (Seiler, Hubner, and Paech 2019; Wang et al. 2018). These methods are often described as "heavy weight" and "burdensome" by leading researchers in the field—deterring the agile practicing community from adopting them, as they are seen to defeat their 'agility' mantra (Cleland-Huang, Rahimi, and Mäder 2014; Cleland-Huang and Vierhauser 2018). This subject is explored in more detail in Chapter 2.

This research is therefore partly motivated to work towards a solution that makes it possible for traceability knowledge to be captured during development time from its original developers. It devises robust techniques and integrates them into developers' regular tool stack and their way of working, to ensure their practicality and the imposition of minimal additional effort. It aspires in the process to encourage a disciplined practice where developers come to see it as natural and part of their work to systematically tie their source code to its original design intents. In summary, given the proper tool support and low footprint mechanisms, there is potential promise to offer a lightweight approach for establishing traceability proactively, and which could prove to suit the agile practicing community as well. Despite the common practice among agile practitioners to disregard user stories (or other similar artefacts) after their implementation, it is well-known and well-established that retaining a connection between these artefacts and their implementation offers substantial advantage in later development and maintenance activities (Tian et al. 2021). Our early phase study showed that developers—including agile developers—are willing to retain and declare such connections if offered suitable tool support.

Empirical Evaluation—a long standing researchers' call

As described earlier, empirical evaluation is one issue that is frequently surfacing in software visualisation literature, receiving strong attention and calls by researchers. The focus is on demonstrating the effectiveness of specialised approaches in addressing particular application contexts (Chotisarn et al. 2020). This research is motivated to work towards addressing this issue by adopting a design science methodology where experts and practitioners are incorporated into the design and development process from the very early stage. By doing so, it distinguishes itself from the majority of existing works where the end user is disconnected from the actual design process, and only involved in the final evaluation phase. As a result, this work is motivated to be better aligned with practitioners' needs and their environment, and to be strongly informed and inspired by their input. It thus does not only aim to evaluate itself in terms of effectiveness and usability, but to also focus on practicality

aspects and how well it integrates into practitioners' operational environments and their ways of working. This aspect is covered in detail in Chapter 3 and Chapter 8.

1.3 Research Objectives

Our research objectives are naturally set around the same problem spaces that are motivating the research. However, this work is largely driven by potentiality and has an exploratory nature, rather than being strictly defined by precise research questions. While not particularly common, such a position is well-credited and supported in the design science methodology—and is sometimes even called for by researchers, especially those in design and engineering disciplines. It is particularly acknowledged that justified "potentiality" and "opportunities" play an important role in design science research, paving the way towards advancement and innovation (Iivari 2007). This position is discussed and argued further in Chapter 3.

Nonetheless, the first three problem spaces that are driving this research, and that were identified above, have received preliminary support for their potentiality from our earlier work, as well as from an early-phase evaluation with experts (Alshakhouri 2013; Alshakhouri, Buchan, and MacDonell 2018). This is perceived to lend further legitimacy and justification to the problem areas being worked on—and their anticipated results—as working towards fulfilling real user needs and towards solving real world problems.

In this regard, this research adopts the specific methodology of Hevner et al., with particular attention paid to its Relevance Cycle in order to establish alignment and connection to the target application environment and its users (A. R. Hevner et al. 2004; A. R. Hevner 2007). This is executed around two primary objectives that summarise the problem spaces of the research and that are defined as follows.

Objective 1

To bring the benefits of software visualisation technology closer to the development community, so its advantages can be reaped by practitioners. To address this, it works on three grounds:

- Investigate and utilise the appropriate technological infrastructure
- Design and apply visualisation to benefit a specific application context and defined user tasks
- Incorporate expert users in the design cycle to inform and guide the research by their actual needs and their way(s) of working

Objective 2

To synchronise software design artefacts with their implementation artefacts, and present the results in an interactive visualisation. This is perceived to bring a number of benefits, most readily of which is artefact traceability. To address this, the research works on two grounds:

- Investigate an approach where software artefacts can be captured in a flexible manner, and with the least prescribed constraints.
- Investigate and devise a mechanism that enables the capturing of connections (tracelinks) between the artefacts from their original developers with minimal interruption and operational footprint. This is important to ensure the practicality of the approach.

Definitions:

User/Practitioner	For the purpose of this research, the *User* is defined to include primarily the following roles: developer, maintainer, and tester. However, it overall includes any actor who is directly involved in creating or maintaining software design or implementation artefacts.

1.4 Contributions

The primary contributions of this work are summarised as follows:

- We introduce a generalised concept for working with software design and implementation artefacts in abstract form
- We introduce the first language-agnostic software visualisation based on the city metaphor
- We apply the above to introduce an approach with promising potential for establishing artefact traceability proactively during development.
- We integrate the above visualisation into a popular development environment, making it readily accessible to practitioners
- We evaluate the work with expert users across three design and feedback cycles , including the use of a real-world dataset, followed by a brief visualisation showcase of 15 open-source systems that utilise a range of different programming languages.

The above can be summarised in one statement as 'we introduce a language-agnostic software visualisation, that is applied to address a specific software engineering problem, and that is designed and evaluated alongside expert users, and integrated into their familiar toolset and workflow'. This work is also seen to contribute to the discipline's body of knowledge through its results and findings, but also through the reported experiences and processes followed.

1.5 Structure of this Thesis

The remainder of this thesis is structured as follows:

Chapter 2 presents an examination of the relevant literature that informed and inspired this research. The review is structured into four problem domains reflecting on their overlap with this work.

Chapter 3 discusses the adopted research framework and methodology, justifying its selection and presenting the actual design of the research.

Chapter 4 presents the development of the research concept, discussing its background and motivations. It then introduces a generalised abstraction for representing design and implementation artefacts to enable the development of flexible tools that are not tied to a certain development practice or programming language.

Chapter 5 covers the technology research and assessments carried out to select the appropriate technology infrastructure, such as APIs and environments, to develop the research tool.

Chapter 6 covers the technical implementation details and the tool development.

Chapter 7 reports on two laboratory mini-case studies using real-world datasets, conducted prior to expert demonstration interviews. It also presents a showcase of visualisations of 15 open-source systems across different programming languages.

Chapter 8 presents the evaluation process, conducted across three phases. It presents detailed coverage and justification of various aspects such as dataset selection, scenario and task design, and the design of demonstration sessions.

Chapter 9 concludes this thesis, highlighting its key contributions and results, and reflecting on prospective future work.

Chapter 2
RELATED WORK

2.1 Prelude

Since its emergence in the late 1980s, research in software visualisation has witnessed steady progress in terms of both the advancement of knowledge and technological development. The past 20 years have seen the discipline gain a spotlight among the wider software engineering discipline, as it gradually established itself on par with other fields. From elementary and rather modest attempts to utilising basic graphical representations (e.g., glyphs) to bring software building blocks to life (Feijs and De Jong 1998; Rilling and Mudur 2002), the discipline has advanced to utilising metaphors that are more natural and familiar to the user—learning from cognitive science that the human brain is better at processing visual information that is relatable to their environment (Storey, Fracchia, and Muller 1999). The desire to learn from the user environment grew from attempts that took natural metaphors to literal extents—from using houses (Knight and Munro 1999), realistic cities (Panas, Berrigan, and Grundy 2003), and planetary systems (Graham, Yang, and Berrigan 2004)—to the more practical metaphors that took and retained the essence of natural metaphors, but remained abstract enough to make them operational—such as the Information Pyramids of (Andrews, Wolte, and Pichler 1997), the City Metaphor of (Wettel and Lanza 2008), and the Evo-Streets of (Steinbrückner and Lewerentz 2010). Tree maps and space-filling hierarchical trees (Johnson and Shneiderman 1991; Shneiderman 1992) have been well-established as effective techniques in the broader field of information visualisation, and the software visualisation field has naturally benefited from this, with numerous works trying to adapt techniques to their purpose. The introduction of the city metaphor in 2007 (Wettel and Lanza 2007a; 2007b) came undoubtedly to represent a key defining milestone in the discipline, enjoying ever increasing popularity and empirical support since then. It inspired a number of important works that attempted to build on and extend its use—e.g., to visualise software memory (Weninger, Makor, and Mossenbock 2020), runtime traces (Krause, Hansen, and Hasselbring 2021), security vulnerabilities (Sinhabahu, Wimalaratne, and Wijesiriwardana 2020), and software evolution (Pfahler et al. 2020)—with it proving to be a highly versatile metaphor. In this regard, the paper of (Jeffery 2019) presents an excellent case for the versatility of the city metaphor in its general forms (including variations such as software maps and evo-streets). Its unprecedented success is probably due to the fact that it combines essential elements from the natural environment—hence users find it very relatable—while at the same time building on top of the tree map structure—thus benefiting of its compactness characteristic. The end result is an abstract and practical approach that is very operational, yet at the same time harbours natural features that scaffold human cognition. Soon after the take-off of the city metaphor in 2007, the Software Maps (Bohnet and Döllner 2011) came as another variant that inherently employed the same metaphorical principle which is based on tree maps but presented it in a 2.5D representation to improve rendering performance.

Over the past seven years, virtual reality and augmented reality came to underpin another shift in software visualisation research, adding a strongly visible influence on recent works. Again, the desire to take advantage of humans' natural cognitive abilities—made possible through the familiar environment that such mediums enjoy—appears to be behind this rising interest. Interestingly, the embrace of playfulness and user engagement is another factor driving this interest, as researchers begin to acknowledge such qualities as beneficial and advantageous in the learning landscape (Romano et al. 2019). In fact, over the three-year period of 2019 to 2021 alone, we were able to readily identify at least 14 papers that were particularly addressing or introducing virtual/augmented reality approaches (Table 10.1 in Appendix I shows some of these). This was by no means a comprehensive or a systematically obtained result—the actual number is likely to be larger—but rather, it serves to illustrate the rising attention to this area. A good number of these works are also co-authored by leading researchers in the software visualisation field—e.g., Baum et al. (2020), Moreno-Lumbreras et al. (2021), and earlier efforts of Merino, Bergel, and Nierstrasz (2018) and Merino et al. (2017).

The software visualisation field has matured significantly since its early beginnings, so presenting a chronological coverage of its history and development is of no particular relevance to this work. However, Frank Steinbrückner in his 2012 doctoral thesis offers an excellent treatment of the field's development (Steinbrückner 2012). Our earlier work has also presented a modest overview of the discipline's history and development (Alshakhouri 2013). More importantly, a number of systematic literature reviews are available that are excellent in offering an overview of the discipline, its key concerns, its various approaches and areas of focus, its challenges and weaknesses, as well as potential research opportunities. A very recent work in this category is that of Chotisarn et al., which focuses particularly on the 'modern' state of the art of the field (Chotisarn et al. 2020). The work of (Bedu, Tinh, and Petrillo 2019) is another good example in this regard.

Instead, in seeking to offer a focused and manageable examination of the literature, we have limited our scope in the majority of circumstances in this chapter to papers published over the past five years—that is from 2017 to 2021—with a few exceptions made where the work is found of particular relevance, or in case of low numbers of publications in the area treated. This is rather an ad-hoc decision driven by the sheer proliferation of recent works in the field, and hence an attempt to control the span of our review. Moreover, this work could in one aspect be seen as interdisciplinary as it shares particular scope with software traceability research, and two other areas from the wider information visualisation domains—artefact traceability visualisation, and requirement visualisation. Hence, despite originating from and being grounded in the software visualisation literature, this work must nonetheless address relevant works in those domains with which it shares particular interest and overlap.

As was indicated earlier, this work is motivated by the desire to bridge the gap between academia and the practicing community with regard to software visualisation tools and their use. It was highlighted earlier that while a large number of tools have been produced by researchers, lack of adoption by the development community is still a key challenge facing the field. It was also highlighted that limited attention paid to practicality and the absence of a focus on applied research are major factors behind this lack of adoption (in addition to availability and accessibility), as well-reported by important recent research. Hence to guide our work, we chose to structure our review around areas of application that are strongly relevant to this work. In this regard, three particular application domains have been identified and are addressed in this chapter:

1- Visualisation Applications Addressing the Development Process

2- Visualisation Applications Addressing Requirement Engineering

3- Visualisation Applications Addressing Artefact Traceability

Aside from the visualisation literature, topics from the software traceability domain come to represent the fourth area treated in this chapter, which is the last section titled:

4- Other Related Work—Traceability Research (Non-SV)

Traditionally well-established application areas in the software visualisation discipline are focused around software structure comprehension, software evolution, and runtime traceability. These areas are well-recognised since the fields' early days (Diehl 2007b) and are well-represented in literature—in more or less the same order they are mentioned. Only the comprehension area is relevant to this work, however, given its existing representation in literature, it is not given particular focus and will only be treated here under the scope of other application areas when relevant.

Lastly, by examining the most recent software visualisation literature (the five past years), we were able to identify six topics that we believe come to collectively shape and drive the discipline at the present day. These are:

Empiricism	Applications and Extensions to City Metaphor
Metaphors	Rise of Applications in Virtual and Augment Reality
Visual and Cognitive Improvements	Layouting

In consideration of previously examined works, these topics (except for VR/AR to some extent) are found to also apply very well to earlier literature in the discipline, and so indeed represent key research directions. A good number of publications under these topics were also instrumental to this work.

However, these topics are generally well-treated in literature by other researchers, hence we choose to focus on areas that are less often addressed and are more relevant to the overarching focus of this work. Nonetheless, some relevant works from these areas appear throughout this work in later chapters to support particular arguments and positions. Moreover, Table 10.1 in Appendix I presents some of the latest research in these areas. This can serve to illustrate research trends and directions in the field, but also to give credit to some of the works that inspired or informed this research in a wider sense.

The diagram in Figure 2.1 summarises the above discussion and classification, and represents our perspective on the current software visualisation literature[7]. It also visually conveys the breakdown and the focus of this chapter, with the areas appearing in yellow boxes receiving particular attention in this literature review.

[7] Except for the Software Traceability Research, which is clearly designated as 'Non-SV'

Figure 2.1: A diagram summarising our classification of software visualisation literature from the perspective of this work, along with other relevant research areas from other disciplines. Only the areas in yellow are addressed in this chapter.

2.2 Applied Software Visualisation

2.2.1 Applications in the Software Development Process

Background. Earlier in this work it has been highlighted that the software development process has received significantly lower attention from the software visualisation community. This is particularly true when it comes to approaching the development process from a practical and operational perspective. For example, we came to notice during our literature examination that there seems to be a general lack of inclusion of the end user in the actual research and development process (i.e. the design science process), and most works only involve the end user in the final evaluation phase. By excluding practitioners from the early phases of the problem solving and discovery process, researchers risk missing out on invaluable insights and input from the actual end users. Most importantly though, we risk creating a disconnection between the solution being developed, and the actual needs of users and their environment. Moreover, a majority of works do not particularly focus on the accessibility of their solutions, probably because their primary concern tends to be on the academic robustness of the approach itself, rather than its potential availability and applicability to the end user. This work is hence aiming towards developing an operational solution that is capable of being readily accessible and available to the development community, and to target real-world problems and needs that are justified by the users themselves. In this way, it attempts to promote the practicality of the software visualisation technology and bring it closer to its intended users.

The limited attention shown to the software development process in software visualisation research was reported and examined in our earlier work (Alshakhouri 2013). While the situation appears to have slightly improved recently, the same argument still largely holds, as made clear by our literature review carried out for this research. For example, by examining recent publications of the dedicated working conference on software visualization (VISSOFT) over the past five years, we were able to identify only 9 publications that specifically addressed the development process, or that explicitly identified specific development tasks targeted by their study[8]. Moreover, only two papers employed modern software visualisation techniques[9], with the rest rather employing information visualisation techniques. By extending our search to other venues, we further identified 7 other papers that used visualisation to address the development process or aspects of it. Only one of these was found to employ software visualisation techniques. Admittedly, this effort was not intended as a systematic approach to arrive to this conclusion, but rather, as an informal support to our argument. Nonetheless, support for this

[8] Comprehension was deemed too generic, and was not recognised as a development task.

[9] Those that are based on metaphorical representations that specifically leverage the human cognitive abilities and take advantage of rich spatial dimensions. More is discussed later in this section.

position can be readily found in the recent works of other researchers. For example, in their 2016 systematic literature review of software visualisation research, Mattila et al have explicitly expressed this fact as a key finding from their survey, stating: "The main result shows that the most studied topics in the past six years are related to software structure, behavior and evolution. Software process and usage are addressed only in few studies" (A.-L. Mattila et al. 2016). They further cast light on this area as representing a research opportunity by emphasising that "software processes" and "business aspects of software engineering" are excellent candidates to benefit from the visual techniques brought about by software visualisation. Further, in a 2019 tertiary systematic literature review (essentially a 'survey of surveys') Bedu et al. highlight a similar finding identifying only one paper that specifically addressed the software process, and four under generic maintenance tasks (Bedu, Tinh, and Petrillo 2019). They identify 'Education' and 'Architecture' as the "best-represented" subjects, followed by 'Maintenance', 'Product Lines', and 'Evolution'. In their recommendations, they call on the software visualisation community "to identify real software engineering tasks" to better support intended users. In fact, the authors cite the same aforementioned quotation of Mattila et al. in support of the current "broadness" they find in software visualisation tools. In their 2020 literature review, Chotisarn et al. appear to reach to a different conclusion stating that the number of visualisations that support "the design and implementation processes" are significant (they report a figure of 48.58%), however, this conclusion appears to be a result of their broad criteria used to classify the papers (Chotisarn et al. 2020). This rationale seems to be supported by their statement that 'envisioned users' of these tools have 'multiple roles', and that those works proposed to support 'all software engineering processes'. Indeed, the same authors proceed later to call on "researchers in the software visualization field to identify the specific context of the targeted users of their proposed visualizations".

In this section, we explore a number of those recent works referred to above that address the development process from the perspective of software visualisation research, and the wider field of information visualisation to some extent.

Visualisation to Support Process Management. In a possible move to address the lack reported in their earlier findings (see above), Mattila and co-authors (Anna-Liisa Mattila et al. 2017) reported the following year on their investigation of two industrial case studies where they developed *a software process* visualisation tool based on data extracted from JIRA repositories. In principle, the tool is a timeline-based visualisation that depicts the status of *bug reports*, *epics*, and *user stories* (see Figure 2.2). The goal of the researchers was to investigate if such views could reveal deviations between the *planned* processes compared to the actually *executed* ones—which according to the study, resulted in largely positive findings. Of particular interest to our work is their remark that while *"raw data"*

pertaining to the software process is largely available today in various forms and repositories, this data is hardly "*illuminating*" to their potential users in their original form. This aligns very well with our intentions of exploiting visualisation techniques to transform such 'raw data' into 'meaningful'—*and contextually relevant*—information, and to bring it up to the surface via interactive visual representations.

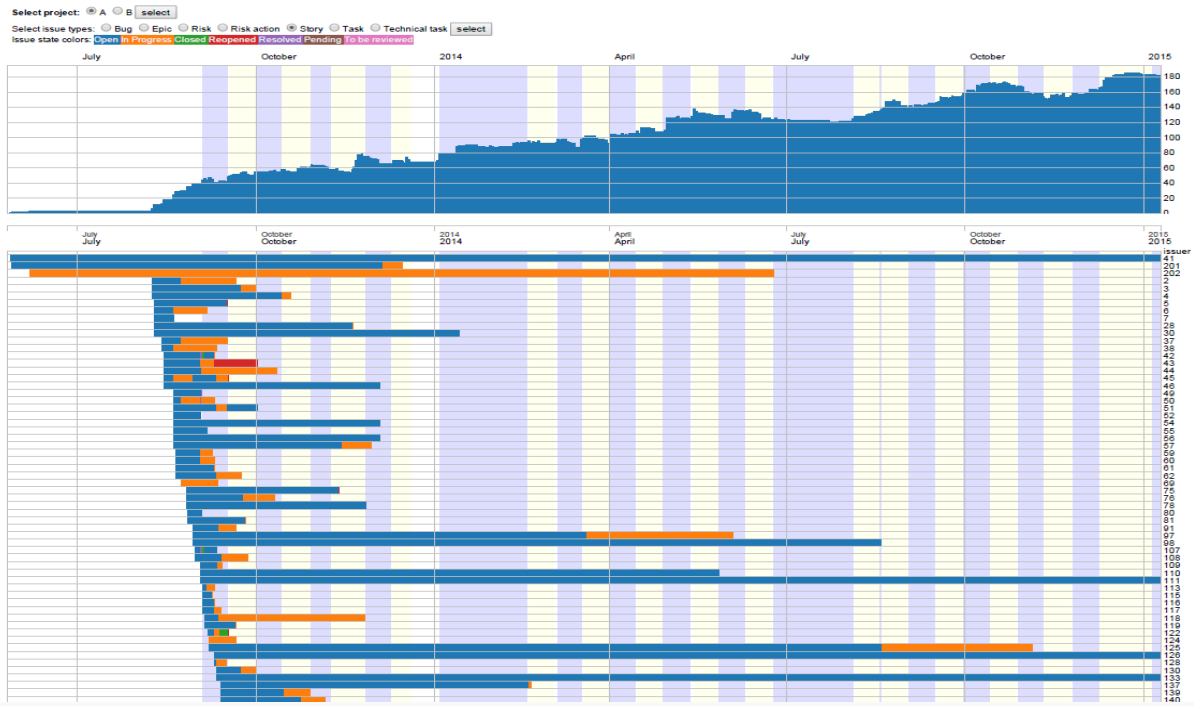


Figure 2.2: A partial segment of a 'Story' data visualisation as featured in (Anna-Liisa Mattila et al. 2017). Complete figure is large, but can be viewed at: https://github.com/pervcomp/DSPDUV.

Another study conducted by Brandt et al. that same year focused on monitoring project progress at a higher level. In their paper titled "*A Dashboard for Visualizing Software Engineering Processes Based on ESSENCE*", they presented a "*Process Visualization*" tool—as the authors referred to it—that was aimed at supporting project managers by focusing on a meta-model level of abstraction rather than the lower level tasks of a software project (Brandt et al. 2017). The approach is based on the ESSENCE standard for project monitoring, which is motivated by concepts drawn from the Kanban agile methodology. The standard defines generic and abstract states called *alphas,* through which project progress is monitored. The final product presented by the authors comes in the form of an interactive web-based dashboard that should look familiar to most Kanban users (see Figure 2.3). Their study shares elements of similarity with our work only in very broad terms—that is, the general goal of utilising visual representations to support the software development process, albeit the focus here is on the management side. Otherwise, the scope and the nature of the process aspects being monitored are very different.

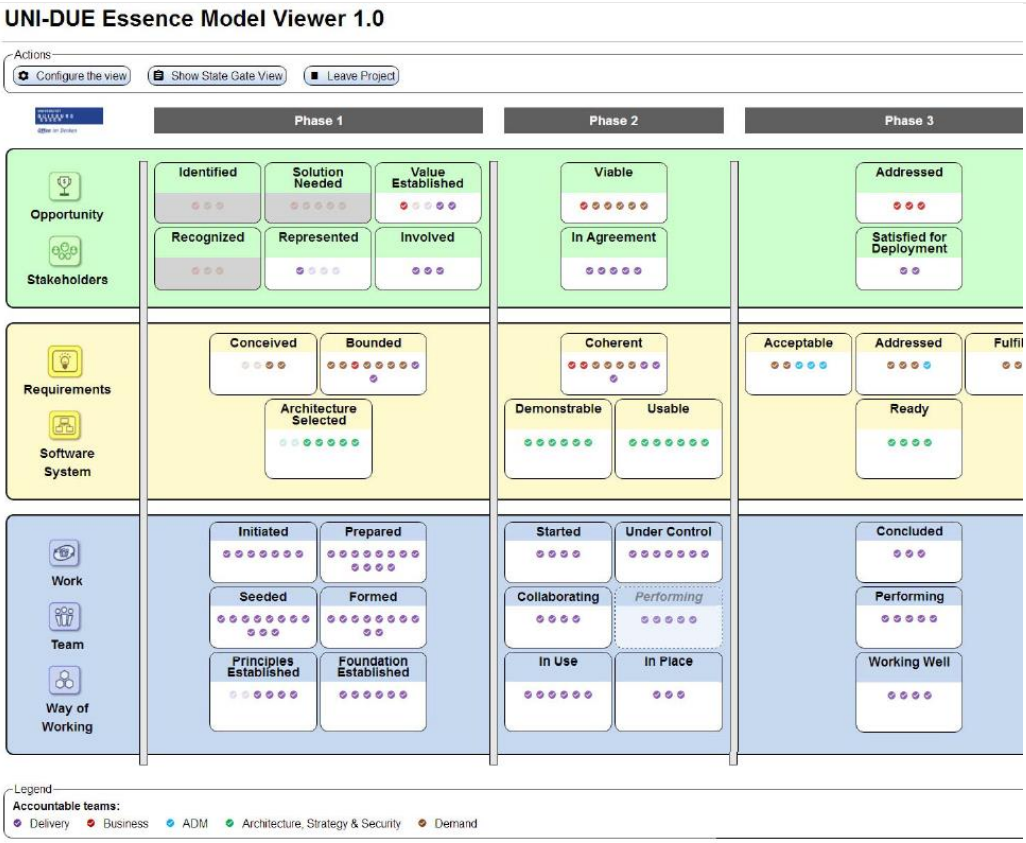


Figure 2.3: The ESSENCE dashboard illustrating a "Process Visualization" concept for supporting project managers (Brandt et al. 2017). Live demo is available at: http://www.s3.uni-duisburg-essen.de/pub/essence/demo/Wobbleboard.html

While the work of Mattila et al. and that of Brandt et al. have interesting intersections with ours, their actual visualisation technique is perceived as a traditional approach that fails to exploit the core elements promoted by modern software visualisation techniques. For example, the use of a human-familiar metaphor to represent software entities (or artefacts)—which is lacking in their approach—is a key principle behind the theory of modern software visualisation, and it constitutes the foundation from which cognitive scaffolding is drawn (Storey, Fracchia, and Muller 1999; Petre, Blackwell, and Green 1998; Petre 2002; 2010). Another key characteristic of software visualisation that is also lacking in their work is the exploitation of spatial glyphs and layouts, which brings significant cognitive advantage by stimulating users' natural spatial memory, as suggested by a number of researchers (Bacher, Mac Namee, and Kelleher 2017; Burgess, Maguire, and O'Keefe 2002; Khaloo et al. 2017). This criticism is in fact not limited to the above two works but is found to apply to the majority of visualisation works that addressed the software development process—as will come to be seen shortly.

> Remark. While admittedly we are not aware of any such distinction in the software visualisation literature that differentiates between the nature of approaches based on the employed visualisation technique, we believe that distinguishing between approaches that employ advanced metaphors and those that employ rather generic information visualisation techniques, such as charts, diagrams, and node graphs, is important.

First, the former modern techniques normally aim to expose actual software artefacts—non-physical entities that actually exist but are nonetheless intangible. These artefacts could be structural components that make up the source code itself, including higher level structural elements such as modules and packages. But they could also be artefacts that represent the design of the software, such as a piece of requirement or a user story. We perceive these as actual software entities, and they are often regarded by software stakeholders as objects with their own perimeter. Modern software visualisation normally aims to visually represent these artefacts using metaphorical objects that are naturally familiar to humans. The objective is to aid users in creating a mental model of those artefacts. The metaphorical objects—which now expose the artefacts and their structure visually—are then used to map additional information in the context of those artefacts.

On the other hand, the other approaches use generic and well-known information visualisation techniques to visualise information 'about software'. They do not visualise actual software artefacts themselves.

Second, those techniques that employ advanced metaphors tend to capitalise on the cognitive qualities inherent in those metaphors, and this is what gives them their natural and familiar feeling. Such techniques are supported by cognitive research to scaffold user understanding and aid in construction of their internal mental models (as indicated above, and as discussed later in Chapter 4). This is by no means undermining the importance of the other approaches—on the contrary, they undoubtedly outperform the metaphorical approaches in vast number of contexts. However, the approaches capitalising on advanced and natural metaphors bring their own advantages for certain contexts, especially—we believe—for users who work closely with the building blocks of software and its construction. This is thanks to the fact that such metaphorical approaches expose those structures and help users to visualise the information they need in-context. Hence, it is important to emphasise and recognise the different potential that these metaphorical techniques present.

Nonetheless, we acknowledge that many researchers would not necessarily agree with this position, and would hold that all software visualisation is but information visualisation 'about software'. Indeed, the above paper and many other non-metaphorical approaches regularly appear in 'software visualisation' venues, and 'software visualisation' is predominantly defined by the renowned three aspects introduced by Stephan Diehl (Diehl 2007) with no regard to the actual techniques employed. However, we perceive the difference between the generic information visualisation techniques and those employing advanced metaphors is akin to a visualisation of—for example—the actual structure of the human DNA with useful information mapped onto it, to a simple chart or abstract diagram that depicts information 'about the human DNA'. Each could have its unique advantage, purpose, and context of use. Admittedly, software itself does not have a shape. Nonetheless, its building blocks (both design and implementation) represent actual objects with well-demarcated boundaries, and they often have a structural composition. This structure tends to strongly dominate and influence the way that developers approach and think about the software they build (see Chapter 4).

Lastly, support for greater utilisation of advanced natural metaphors is found in a good number of other software visualisation works that emphasise the benefits of amplifying human cognition. For instance, in their literature review findings Bedu et al. call for the "usage of more advanced visualization techniques" and recommend to "rely on visualization techniques which are able to amplify the human cognition process". They further add, "the trend in recent papers is to use real

> metaphors (e.g. city metaphor) instead of abstract ones, to represent software artifacts…" (Bedu, Tinh, and Petrillo 2019).

In consideration of the above discussion, another aspect that could be noted about the two works discussed already is the detachment of the visualisation from the actual working context. While this is not necessarily seen as a negative aspect (as it is acknowledged that each visualisation should be designed to better fit its purpose), we believe that contextualisation–representing the information to users in their actual working environment where they need it most–offers greater advantage both cognitively, as well as with respect to revealing new insights. Nonetheless, given the targeted audience are of leading or managing roles, such techniques are undoubtedly better suited to the purpose.

Visualisation to Support Awareness of Software Artefacts. A relevant research effort conducted in this area that lends support to our work (in terms of its findings and motivations), is the systematic mapping study conducted by (Paredes, Anslow, and Maurer 2014). The authors were interested to explore and identify existing visualisation techniques that particularly supported *knowledge sharing* among Agile practicing software teams. Since *software visualisation* tools that serve this purpose are hardly present, the survey targeted instead the body of literature of the broader discipline of *information visualisation*. Interestingly, the authors frequently associate 'software visualisation' with raising awareness of software artefacts—a characteristic that we believe is particularly enabled by the metaphorical techniques. The authors describe knowledge sharing about artifacts to be particularly challenging among agile practitioners, and repeatedly call for 'software visualisation' tools to enable 'knowledge sharing' and 'awareness of artefacts'. Furthermore, the authors specifically called upon researchers to support "software development tasks based on the needs of real software practitioners and teams" by working directly with those practitioners to understand their needs and gain insight about their environment. These elements share strong parallels with our motivations and approach. Our work is motivated by the objective of capturing the original design knowledge, and its implementation knowledge, and then presenting it in synchronised fashion to users in their actual working environment. We also collaborate with real agile practitioners from the industry to guide and inspire our work, as detailed in Chapter 8. The authors report that while 'information visualisation' tools are widely adopted among agile practitioners, 'software visualisation' tools are almost absent. They attribute this to the lack of empirical work with such tools. While this is true, the situation has since been gradually improving with regard to empirical evaluation, as reported recently by (Chotisarn et al. 2020). Moreover, in Chapter 8 we also argue that there are other factors contributing to this lack of empirical study, mainly the disconnection between software visualisation research in academia, and the practitioners and their environment contexts in the real world.

Visualising Development Activities. Another work that shares some common ground with our research in terms of objectives (but not the approach) is that of Saito et al. in their 2018 paper titled *"Discovering undocumented knowledge through visualization of agile software development activities"*. The authors draw attention to the nature of agile software development practice that tends to place lower importance on maintaining and keeping track of requirements documentation—resulting in the loss of knowledge that could potentially be vital for the future sustainability of the software product (Saito et al. 2018). They then introduce an approach that links 'commit activities' from versioning control systems to 'tickets' in issue tracking systems, and produce a time-series visualisation of those activities in the form of charts to help uncover otherwise undocumented knowledge. By 'undocumented knowledge', the authors mainly refer to exposing commit actions that are unlinked to tickets—representing a loss of a knowledge link that is potentially important for future maintenance activities. In their approach, the unlinked commits are depicted alongside the linked commits in a time-based chart that serves to reveal the frequency and the date of occurrence of these events (see Figure 2.4). Meanwhile, a spreadsheet tool provides detailed projection of both types of events where each is individually hyperlinked to its commit details. Exploratory case studies were conducted on two industrial projects where researchers were able to track down and retrieve the *undocumented knowledge* from the developers and the requirement engineers—which was later deemed by these users as being *required* for future development. While this work does not identify itself as 'software visualisation' and is in fact published in a requirements engineering research venue, it is however presented here as the authors label their work as 'visualising agile software development activities', and the work's motivation converges well with the objectives of this work (in terms of capturing important software activity knowledge that is otherwise lost).

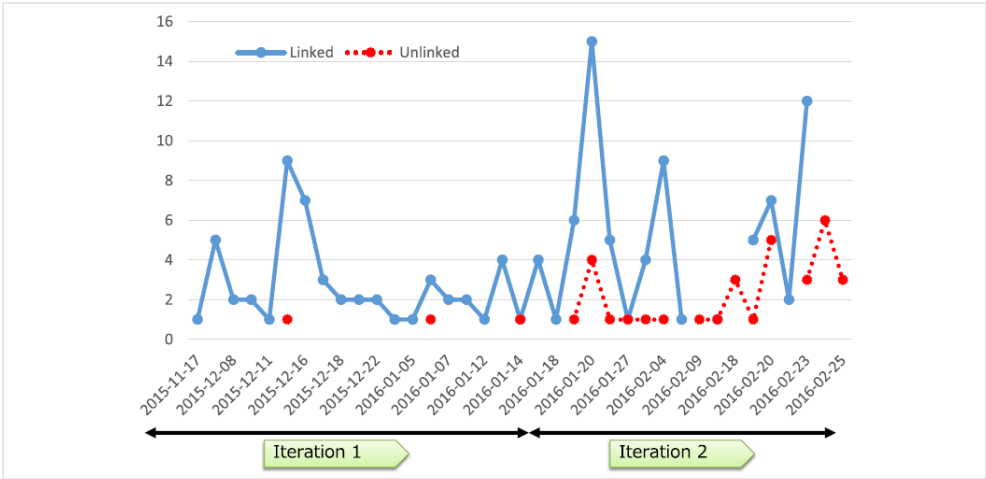


Figure 2.4: View from Saito et al. work depicting a timeline of repository commit operations being linked/unlinked to original issue tickets (Saito et al. 2018).

Visualisation Dashboards. Research to support the development process through visualisation dashboards has been steadily increasing in the literature (including in the specific venues of software visualisation). For example, a recent similar work to those presented above is that of (Scott-Hill et al. 2020) in which they attempt to analyse information and development activities from git repositories and Jira, and then present it to users in the form of graph visualisations to reveal new insights and knowledge. However, in this work, the authors focus specifically on the development progress and the use of large touch displays to facilitate team collaboration. They also offer a varying number of visualisation formats that can be added and organised on the dashboard dynamically (see Figure 2.5).

Figure 2.5: A dashboard visualisation of development progress to support team collaboration. Source: (Scott-Hill et al. 2020).

In a similar recent work, Schreiber et al. report on a web-based interactive dashboard through which they mine software repositories and then present insight and knowledge about the development process through a number of visualisations (Schreiber et al. 2021). An interesting element in their approach is that the visualisations are generated based on predefined questions of interest that are queried into a database, where the mined data is stored. While this implies a level of tailored visualisation, it appears that such visualisations need to be pre-prepared for each project, and then assembled into a web dashboard specific to that project. Nonetheless, their approach is able to generate a range of visualisation techniques, and to utilise a standardised data model, giving good potential for interoperability with other tools. However, unlike the works presented earlier, the visualisations here capture their information from git repositories only, and do not integrate it with knowledge obtained from issue tracking systems or other requirement management tools.

Also in 2021, Genfer et al. presented a similar visualisation dashboard that mines information from git repositories as well, but in that work, the authors target quality engineers in particular, offering them

a varying number of quality metrics about their software development projects (Genfer et al. 2021). Figure 2.6 presents an example of their dashboard output.

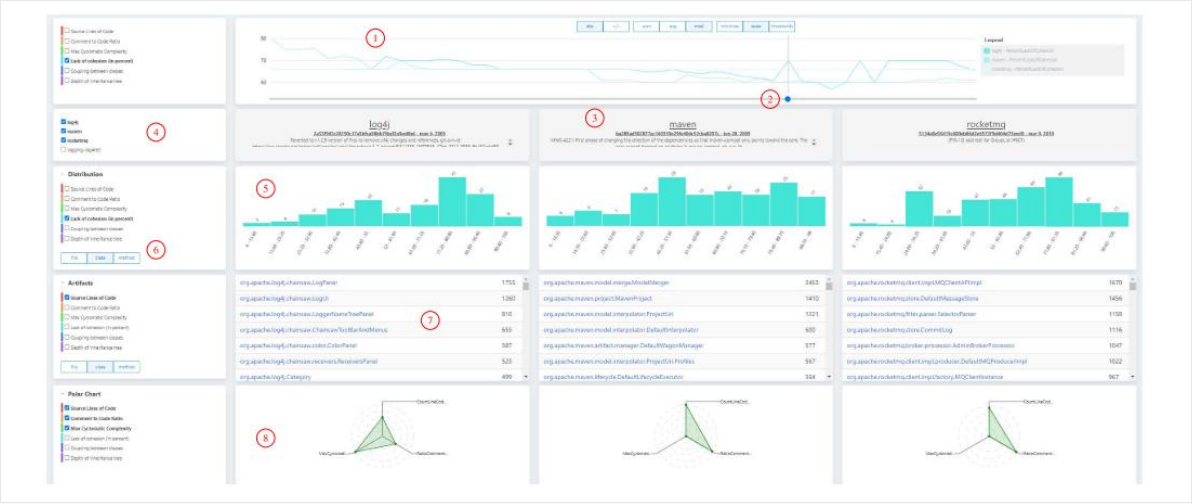


Figure 2.6: A dashboard visualisation of software quality metrics. Source: (Genfer et al. 2021).

Remark. Similar to the works covered earlier, we can readily observe that the above also predominantly use generic and non-metaphorical visualisation techniques that lack the advantages enjoyed by the natural metaphors used in other software visualisation works. Aside from missing the cognitive benefits, these techniques also lack the dimensional depths that allow the metaphorical techniques to encode and map richer information. In fact, we believe that a notable feature of natural metaphorical techniques is their versatile expression power in that they allow data from multiple sources (and multiple aspects) to be combined and presented in a natural form that users can readily relate to. In other words, they seem to have a natural ability to assimilate varying information and present users with new knowledge—in succinct form and in context—that is otherwise harder to expose. This quality is supported by other researchers as indicated earlier, but has also been acknowledged by expert users during our evaluation sessions, who voluntarily and without prompting expressed their approval of this characteristic. Nonetheless, and as expressed earlier, the use of simpler and non-metaphorical visualisation techniques is by no means being presented as a negative aspect. Such techniques appear indeed better suited to the particular context, the usage, and the audience they are designed for.

Another important observation is that almost all of the above works rely on extracting (or mining) the data from its original source, and then storing it in either local databases or middleware files, in order to carry out the necessary logic operations before generating the visualisations[10]. By resorting to storing

[10] This observation can be perceived from the discussion context in the papers. However, some has explicitly stated this such as (Genfer et al. 2021) and (Schreiber et al. 2021). Moreover, no paper claimed to feature an on-the-fly or a 'live' approach.

the information in a middleware medium, a number of adverse effects are created. In terms of practicality, the tool becomes harder to adopt and the entry barrier is increased. Many organisations are increasingly concerned about the privacy and security of their data, and are reluctant to have it stored outside of their premises. Indeed, during our work with the expert users, a number of participants were quick to ask if we were storing the information somewhere, and expressed the rising similar concern in industry. Of course, a solution could have the database stored locally, but that only introduces a further barrier by having to set up and maintain additional technology. More importantly though, by having such middleware files, a solution will normally lose the timeliness of the information being presented. Offering an on-the-fly solution that presents information in a live and constantly up to date manner is certainly advantageous and more desirable. It also happens to avoid a number of technical intricacies with having to regularly synchronise the data. If an approach involves data manipulation, the situation is then only exacerbated. This aspect is discussed in more detail in Chapter 6, when we present the design and implementation of the research tool. Nonetheless, we were able to find at least one work where the authors employed a 'live' solution to their approach.

Live Visualisation. Unlike the above works that relied on intermediary files or databases to store the mined data, Fernandes et al. report on a 'live' approach in which they utilise the modern APIs of Visual Studio Code to provide developers with quality metrics calculated in real-time (Fernandes et al. 2020). In this particular aspect, this work shares a similarity with ours in terms of utilising the modern technologies offered by Vscode—the technology aspect of this work is covered in detail in Chapter 6. Their short paper did not offer any views of their visualisation, but they report on presenting users with small bar-chart graphs in a side menu. In this regard, this work still shares similarities with the previous works in employing basic graph-based visualisations.

Using Metaphorical Visualisations. In fact, the only works we were able to find where advanced metaphorical software visualisation techniques were employed to support the development process (other than the generic purpose of comprehension), were that of (Bohnet and Döllner 2011) in which they used their 2.5D software maps metaphor, and that of (Bacchelli et al. 2011) in which they used the city metaphor. However, these works will not be discussed here as they are well-beyond the date range we committed ourselves to. Nonetheless, discussion of both works appears in our paper published in the early stages of this research (Alshakhouri, Buchan, and MacDonell 2018). The work of (Limberger et al. 2013) is also another important work that falls in this category, employing the software maps technique to deliver a good range of analysis activities that support the development process.

Visualisation to Support Code Review. The above discussions show that targeted applications of use appear to be increasing in the software visualisation literature, as well as other venues—which is aligned with recent calls from researchers to focus on more applied research in the field. In the same spirit of the above works, a paper of interest is that of (Balci et al. 2021) where they focus their attention on supporting Code Reviews. The authors introduce a GitHub plugin-tool that presents developers with a UML-like diagrammatic visualisation of commits (see Figure 2.7, top). The visualisation aims to expose the structural relationships between GitHub issues and the commits, as well as other relevant information such as the users associated with these items and percentages of changes made. Their approach is interesting in that it enables good interactivity levels, allowing users to dynamically navigate the diagram nodes through expand/collapse actions. The authors also bring good context in this case to their visualisation by offering what they named 'Artefact Map' view, where they present GitHub issues, commits, affected files, and associated developers in one view (see Figure 2.7, bottom).

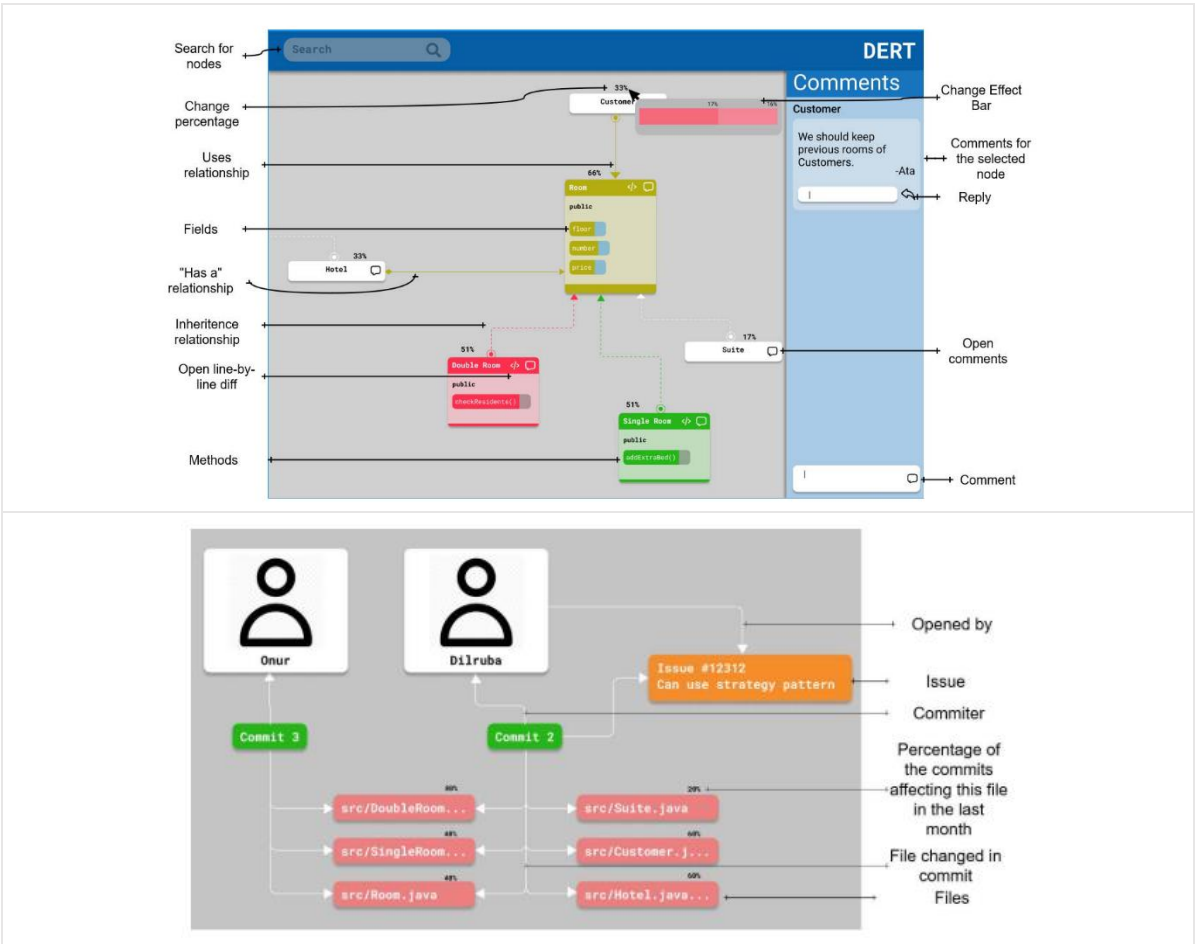


Figure 2.7: Visualisation to Support the code review process. Source: (Balci et al. 2021).

Visualising Development History. In a work that is related to some extent, Kim et al. report on another GitHub tool aimed at supporting users in overcoming the density and complexity involved in understanding the development history (Kim et al. 2021). They present a highly interactive and sophisticated visualisation dashboard, offering their users various types of visualisation techniques to explore Git metadata. An interesting part of their work is the techniques they had to develop to overcome the dense and large-scale graphs to make them practical to work with—some of which can inspire similar approaches applied to metaphorical visualisations.

Visualising Design Knowledge. An interesting work that shares similarities with ours on different levels is that of (Kleebaum et al. 2021). In their paper titled "Continuous Rationale Visualisation", the authors present an umbrella of extensions across Jira, GitHub, and other tools to support the decision-making process in the context of rationale management. A key similarity of their work (in a broad sense) is the fact that they aim to capture the design knowledge from different sources (mainly Jira and git repository) and assimilate it with the other relevant software artefacts, mainly source code. Indeed, the authors emphasise here the importance of presenting the knowledge in context of the specific software artefacts under examination, be it a requirement, source code, or other. From a motivational and philosophical perspective, this aligns strongly with our work and shares similar lines of thought to those expressed earlier. For example, the extensions introduced allow users to inspect 'visualisation views' and 'list views', presenting important decision knowledge in the context of a work item (e.g., in Jira) as well as the ability to navigate to such views from the context of source code (through IDE extension). Overall, the authors introduce 7 views that include interactive and customisable visualisations. On the other side though, the work employs node graph visualisations and other traditional chart techniques (see Figure 2.8, lower part)—hence differing greatly from our work in terms of the nature of the visualisation employed. Moreover, the tasks supported, and the purpose of the tool, are also very different. In fact, at this point, and by examining the node graph in Figure 2.8, it is worth emphasising again that advanced metaphorical visualisations offer a richness of dimensions that is more capable of exposing the context of knowledge presented.

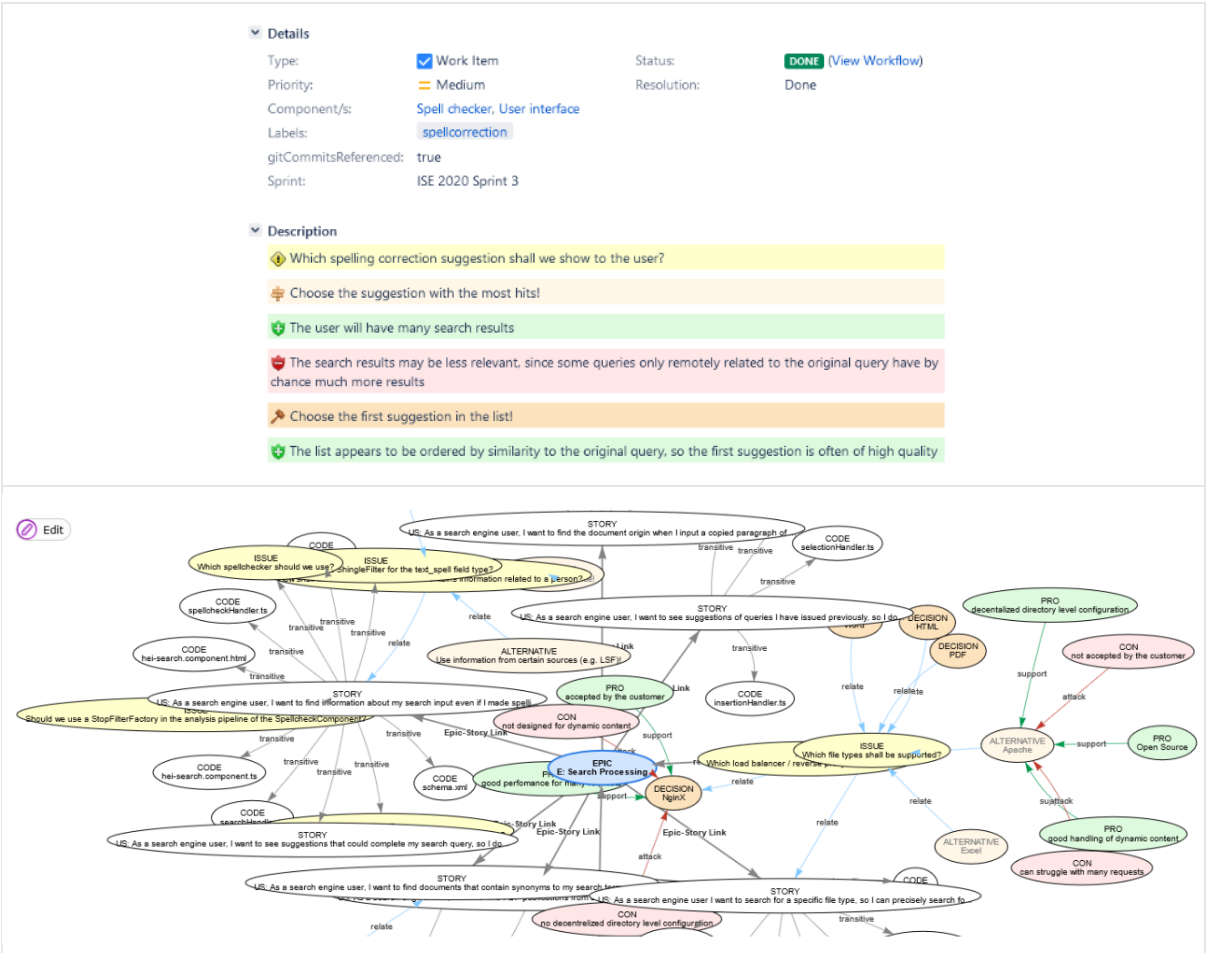


Figure 2.8: top: Contextual decision knowledge presented in description of Jira work item. Bottom: A node graph showing various artefacts in the context of an epic. Source: (Kleebaum et al. 2021)

Visualisation for Software Product Lines. One last relevant work that is worth some attention before closing this section is that of Software Product Lines (SPL) engineering. Given the sheer complexity and the large number of components that are commonly involved in this domain, visualisation techniques were an immediate candidate to turn to in search of cognitive alleviation (Pleuss, Rabiser, and Botterweck 2011). Indeed, it was observed that research exploiting visualisation techniques for the support of the SPL software process and its activities has recently become an emerging trend, and works in this regard started to appear among the body of literature of *Information Visualisation* in its broad spectrum, as well as among venues specialised in Software Visualisation. For instance, looking only at IEEE's Working Conference on Software Visualization (VISSOFT), we could find two papers specialised on the SPL process in 2016, with one paper having appeared yearly in each conference between 2013 and 2015. A comprehensive and interesting systematic mapping study on the subject was also conducted by (Lopez-Herrejon, Illescas, and Egyed 2018) and reported in their paper titled "*A systematic mapping study of information visualization for software product line engineering*". The authors identified 37 papers published between 2007 and 2016 that specifically

addressed the utilisation of visualisation techniques for the benefit of SPL practices. Their study focused on the *visualisation techniques* used, the type of SPL *activities being supported*, the state of *empirical evaluation*, and lastly, to identify the *gaps and opportunities* for future research. Given the fundamental role played by software features in SPL engineering practice, activities such as feature models, feature location, feature traceability, and identification of feature relationships are reported as crucial aspects of the practice and account for the vast number of activities commonly targeted by these visualisation techniques (as the authors conclude in their study). More recently, the work of (Bedu, Tinh, and Petrillo 2019) on software visualisation literature also identified the increasing interest of the SPL domain in utilising software visualisation techniques. They describe the utilised visualisation techniques in this field as mainly aiming to expose 'feature models' through the use of 'trees', 'graphs', and 'bar diagrams', and proceed to describe those techniques as "basic visualizations" that rarely offer the user the ability to navigate between the artefacts. They recommend instead the use of "advanced visualization techniques and tools" and call for "the integration of tools developed in popular IDEs", concluding that the full potential of visualization in the context of SPL has not yet been explored. The authors then draw attention to the potential of the city metaphor and other natural metaphors describing them to amplify human cognition. While our work does not address the domain of SPL in particular, nonetheless the recommendations, as well as the tasks to be supported by visualisation, find strong parallels between this work and ours. Moreover, the arguments of Bedu et al. appear also to align with the position we elaborated on with regard to the distinct potential that metaphorical visualisations offer to certain contexts.

Closing. The above reviews serve to demonstrate the rising interest, awareness, and receptibility of utilising visualisation tools to support the software development process in all its aspects. This interest is not confined to the academic research, but reaches also to the development community and industry, as indicated by many of the evaluations reported by the above studies.

2.2.2 Applications in Requirements Engineering (REV)

History. Another application area that has attracted strong interest in utilising visualisation techniques is that of *Requirements Engineering*. Research into this domain appears to be in its early days with very few publications available on the topic. The field appears also to be less recognisable among the broader visualisation research community. This is interesting given that IEEE had held five international workshops on the topic between 2006 and 2010, named the *International Workshop on Requirements Engineering Visualization*—from which the acronym REV is derived. But as evident from the nature of publication venues that hosted those studies on REV, this rising interest in *Requirement Visualisation* appears to have escaped the attention of the Software Visualisation research community. Out of 61 papers that constituted our initial effort to survey the topic at the beginning of this research, none was ever found to be published in a venue specialising in Software Visualisation, nor under those specialising in the wider field of Information Visualisation. They are exclusively published under either IEEE's less known venue of REV, or under some other general Requirements Engineering venues. Nonetheless, interest in visualisation applications in this area is gradually—albeit slowly—growing as is evident from the few specialised works that are discussed next. Also, there are at least two papers published recently under the specialised VISSOFT conference that we identify as addressing the subject, despite the works not specifically presenting themselves as such. These are: the work of Kritzinger et al. (2019) titled: 'A User Study on the Usefulness of Visualization Support for Requirements Monitoring', and that of Fiechter et al. (2021) titled: 'Visualizing GitHub Issues'.

Highlights. Given the relatively immature state of the field and the relative scarcity of a body of literature, it is beneficial at this stage to identify the key motivations behind it, the key problems that it tries to solve, and the particular techniques that it applies. This is also helpful to inform our work by uncovering common points of interests.

By examining key papers found on the topic, we identified five particular aspects motivating the use of visualisation techniques in requirements engineering. These are:

1) Understanding and effective communication	2) Enabling traceability
3) Identification of relationships	4) Modelling
5) Enabling exploration	

For example, in their 2014 paper, (Savio and Poothiyot 2014) mentioned dependency identification, multi-directional traceability, and 'increased cognitive amplification' (i.e., understanding) as being the key objectives behind their proposed *Pyramid* visualisation technique (a concept introduced by the first

author and his colleagues in 2011), and behind its implementation tool named ReBlock. Their technique depicts requirement artefacts as small cubes that are arranged hierarchically and presented in a wireframe 3D structure resembling a multi-face pyramid (see Figure 2.9). Each face of the pyramid is meant to present a different perspective corresponding to the different stakeholders. Connection lines are then displayed on demand to show the dependencies and hierarchies.

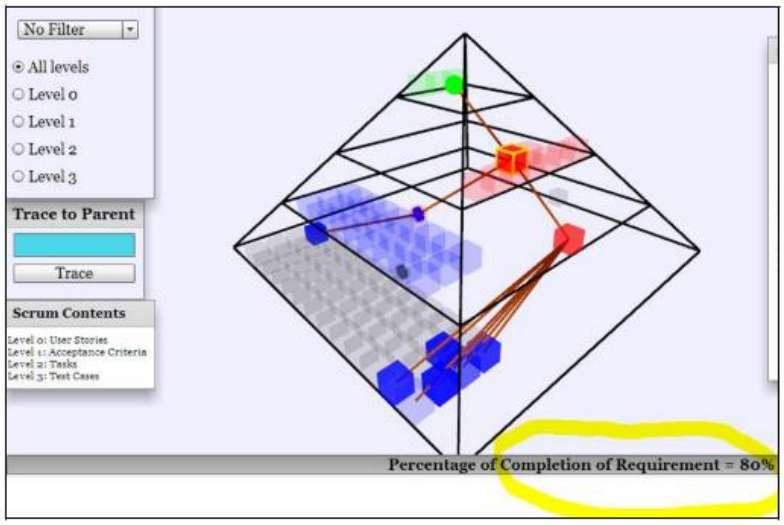


Figure 2.9: ReBlock visualisation of software requirements. Source: (Savio and Poothiyot 2014) (highlight is original)

Laying the Groundwork for the Field. In one of the key early works on the topic—as perceived from and limited by the extent of our literature review—(Gotel, Marchese, and Morris 2007) presented a motivating foundational paper in 2007 titled "On Requirements Visualization" in which they stimulated research in the field and drew researchers' attention towards the opportunities, the practical needs, and the theoretical justifications behind its importance and foreseen advantages. The authors' motivating arguments find parallels with those invoked by our work. For example, they cite the empowering cognitive advantages of *natural metaphors* and their *mappings* in fostering *mental models* of software artefacts in the minds of users, which are otherwise complex to explore in their original textual forms. Other motivations they recount includes metaphors' expressive power in making information readily "pop out" to the user, and the importance of *visual representations* in enabling "shared communication" and "provoking insight" (Gotel, Marchese, and Morris 2007). The paper plays a prominent role in laying out the foundational and conceptual basis for requirements visualisation as a field, with the authors questioning the lack of interest in visualisation technology despite its promising potential and despite its rising utilisation and success in other software engineering fields (quoting in particular, the software visualisation discipline). The authors identify the key areas typically targeted by requirements visualisation research to be: ***1) the effective communication*** of the structure, relationship, and evolution of requirements, *2) elicitation* of requirements, and *3) modelling of requirements (as in*

UML)—giving examples from literature for each category. They also provide prospective visualisation designers with guidance and advice on how to develop effective techniques and mappings. The ultimate goal behind their work—as put by the authors—was to drive the development of visualisation approaches that would enable stakeholders to "actually 'see' the requirements" by transforming textual data into visual artefacts that *promote awareness, facilitate high-level decision making*, and that *support diagnostic activities* (Gotel, Marchese, and Morris 2007).

The authors also discuss some criteria for designing (or choosing) efficient and familiar metaphors, and come to describe the *cartographic approach*—to which the city metaphor belongs—as one that satisfies these criteria[11]. They conclude that, while other areas of application have successfully benefited from information visualisation techniques, requirements engineering falls a long way behind in this regard. This paper represents one of the early founding works motivating research in the REV area. Many of its arguments are also found to intersect with motivations behind this research. For example, effective and shared communication of requirements, fostering mental conceptual models, provoking insights, promoting awareness, and supporting high-level decision making—are all strongly emphasised concepts in the authors' paper, and which appear across this work in support of different arguments. However, and as detailed in Chapter 4, rather than treating requirements as isolated artefacts, this work strives to take a more holistic approach, looking at requirements as part of a greater set of design artefacts, and then seeks to integrate them with their resulting implementation artefacts.

Current Position on REV Research. With the lapse of the REV workshop in 2010, it remains unclear as to the future of this research area, or whether it has merged under another requirements engineering research venue. In their 2016 paper titled "*Requirements Engineering Visualization: A Systematic Literature Review*", (Abad, Noaeen, and Ruhe 2016) identified no other specialised venue for the field, and out of 26 selected papers, 8 came from the IEEE REV workshops while the remaining works came from other non-specialised venues, making it difficult for researchers from outside the field to discover them. No clarification is provided by the authors as to the postulated future path of the field. Nonetheless, the authors placed a strong emphasis on the clear limitation of studies on requirements visualisation, as well as the "clear deficit" of visualisation support to the various RE activities (Abad, Noaeen, and Ruhe 2016). Citing a number of earlier studies, they presented a strong motivation for the field and called upon researchers to exploit the potential of visualisation techniques to support overarching problems in requirements engineering. In their arguments and motivations, they mostly relied on the ***communicative*** and ***expressive*** power of the visual artefacts, as opposed to

[11] It is worth noting that this remark is made just before the rising popularity of the city metaphor in software structure visualisation applications.

traditional and textual communication means. Interestingly, and in contrast to the traditional definition of 'Software Visualisation' that is commonly quoted by other researchers—which places a specific highlight on the Structure, Behaviour, and Evolution of software—the authors here quoted Stephan Diehl's less popular definition that highlights instead the '*software development process'* and *its 'artefacts'*—a stance that is more encompassing and that aligns with the adopted definition in this research, as discussed in Chapter 1. Of particular interest, though, is their focus and promotion of the "*Knowledge Visualisation*" concept as opposed to *Data* and *Information* Visualisation, respectively. Elaborated by (Burkhard 2005), the concept attempts to place emphasis on those visual techniques that particularly amplify the '*creation'* and the '*transfer'* of knowledge between multiple users; in other words, it promotes shared communication of 'mental concepts' as an aggregated component of *knowledge*, rather than disintegrated pieces of data or information. However, out of their 26 selected studies, only 5 were reported to qualify as promoting knowledge visualisation (Abad, Noaeen, and Ruhe 2016).

Another important contribution of their work is their detailed analysis of the existing visualisation techniques used in requirements engineering, their purposes, the particular activities being supported, and the identification of gaps to direct potential future research. Their findings align very well with the five key motivations behind REV research that were identified and stated earlier (i.e., communication & understanding, traceability, exploration, relationships, and modelling), albeit they adopt a slightly different classification structure (for example, traceability and relationships are placed under their higher category of 'Evolution') and they include 'Elicitation' as an additional type of activity. They reported that *communication* and *evolution* (which fall under traceability in our list) constitute the two activities that mostly need visualisation support, while at the same time being the least supported by existing approaches (with elicitation receiving more support). Lastly, they called for solutions that specifically incorporate the presentational techniques of '*storytelling'* and '*maps' (hierarchy & structure)*, and that supported better user '*interaction'*. This lends particular support to our research as these elements find parallels in our work. Unfortunately, while the authors identified the activities that are perceived to be mostly in need of visualisation support, the study makes no attempt to identify *the most popular (or commonly practiced) activities* in requirements engineering. A preliminary assessment based on our literature review reveals that *communication* appears to be at the forefront of the list, with modelling coming next. This should be an important factor to take into account for future researchers seeking to address the gaps in REV, as a sound correlation must be first made between *those commonly practiced* and *those least supported* by visualisation to arrive eventually at those activities that truly deserve attention. Lastly, as for the particular visualisation techniques that were so far employed in REV work, the authors identified five, with the most relevant being the visual tree

layouts and tree-maps that depict requirements hierarchies, relationships, and dependencies. Other techniques included conventional diagrams and charts, and pictorial sketches of actual products to help validate and verify the requirements.

Graphical Vs Textual. Before going further in our examination of the requirements visualisation literature, it is worth drawing attention to some empirical support. In a 2013 paper, Sharafi et al. reported the conduct of an empirical study to assess the accuracy and efficiency of requirement comprehension tasks when using structured textual representations versus 2D graphical representations (Sharafi et al. 2013). Interestingly, their findings suggest that accuracy was found to be the same, while time spent (hence efficiency) on tasks increased when using the graphical representations. Yet, they still reported that 82% of participants preferred the graphical representations over their structured textual counterparts. Moreover, when presented with both options, 96% of participants started with the graphical representations to address their tasks. We argue that these results are probably suggestive of the natural affinity that humans have towards visual consumption of knowledge—which is well-documented and well-known in literature. The increased time spent in performing the tasks could simply be due to the fact that participants were at that stage unfamiliar with the graphical representations, and that they needed to first learn and gain sufficient familiarity in interpreting those representations, before advantages could be reaped. Indeed, the authors arrived at a similar conclusion, stating that "… in agreement with previous works, developers must learn how to use a graphical representation before using it", and that "training is required before the benefits of a graphical representation can materialise". The authors finally state that 'time' and 'effort' have improved 'significantly' in the later sessions, after the participants had had some exposure to the models in the earlier sessions. Considering this point, we hence argue that using natural metaphors, and especially those with spatial dimensions and structures most familiar to the human environment, could probably increase the initial familiarity and affinity of the visual representation to a new user, reducing the time required to interpret those representations. Also, while the authors' finding could seem initially as a granted and matter-of fact-presumption, it nonetheless highlights the importance of conducting participant training sessions in such experiments, before arriving at meaningful conclusions from such empirical studies. This is particularly true in the context of assessing visualisations' efficiency for the various applications they are developed for. As a side note, during our expert interviews, we opted for the tasks to be conducted by the researcher (and having participants as observers) in order to circumvent this issue, in particular.

Source Code Artefacts. Also very relevant to this work is a later extended empirical study co-authored by Sharafi where they assessed the importance of 'source code entities' in the role they play when developers engage in requirements traceability, as part of their program comprehension and maintenance tasks (Ali et al. 2015). Their results conclude that the naming of those entities, as well as their semantics, is the most important to developers during these tasks. They particularly conclude that 'method names' and 'comments' were the most relevant and most frequently inspected by participants—even when they enlarged class names to make them more visible—stating that developers "search/comprehend source code by reading method names and comments mostly", and that "as soon as developers read a requirement, they paid immediately (sic) attention to method names and then comments". Moreover, the authors report that within those naming and terminologies that developers seek to spot, they mostly look for 'domain-level source code terms'—indicating (as we perceive) that developers are attempting, whether knowingly or not, to reconstruct domain-level concepts through whatever they could find of relevant terminologies in the source code. These findings are supportive of the importance of **design concepts** to developers as well as to the importance of **connecting** these concepts to their actual source code manifestations. This topic is discussed more in Chapter 4. They also support our decision to pay particular attention to supporting method-level granularity in our work (including other similarly scoped levels, such as functions).

REV in Recent Literature. While we did not identify any current venue specialising in requirements visualisation, works targeting this application area keep appearing sporadically among wider RE research circles. Given the present rising popularity of agile software practices, it is unsurprising that almost all these works appear to afford their focus to this particular scope. A prominent work in this regard, and one that shares good relevance to this work, is that of Lucassen, Dalpiaz, and co-authors, across several publications between 2016 and 2018. Their earliest paper involved no visualisation considerations and was primarily focused on investigating the effectiveness of user stories as artefacts of software requirements in an industrial survey-based study, and has reportedly led to substantially positive results (Lucassen, Dalpiaz, Werf, et al. 2016). They concluded their study by motivating researchers to dedicate more resources towards the investigation and design of *"advanced methods and tools"* for supporting user stories—complaining at the same time about the "little to no" support from academia addressing this area. Their call was generic, without specification of any particular context or aspect.

That same year, recognising the practical difficulty in constructing a mental image of user stories and their relationships, the team embarked on developing a visualisation approach to alleviate the problem (Lucassen, Dalpiaz, van der Werf, et al. 2016). Rather than re-representing each user story as a visual

artefact, in their paper titled *"Visualizing User Story Requirements at Multiple Granularity Levels via Semantic Relatedness"* the team aimed instead to focus on the general and *higher-level concepts* embodied by those user stories. They applied 'semantic relatedness algorithms' to extract the core concepts from the textual user stories, identifying relationships among those concepts, and then clustering them into varying levels of granularity to arrive at higher level concepts representing the entire system. The resulting high-level concepts are eventually interrelated with the relationships between the original user stories to finally produce a *graph-based* visualisation depicting the entire hierarchical network of concepts. The approach is multi-granular in that the user can zoom into each of the higher-level concepts to uncover the underlying inner concepts. The preceding description should become more comprehensible after inspecting the illustrations in Figure 2.10. The authors described their visualisation as being preliminarily capable of highlighting 'information classifications' at a very high level. However, applying their technique to four real-world data sets, the authors discovered potential problems in their approach that could be summarised in three major points; 1) the resulting complexity of the network of relationships that seriously affected the practicality of the generated graph, 2) the need to implement better algorithms to enhance the accuracy of identified concepts and their relationships, and 3) recognising the potential to apply "*state of the art visualisation techniques"* to replace their "*graph-based"* one (Lucassen, Dalpiaz, van der Werf, et al. 2016). These findings have aptly informed their next work, described shortly.

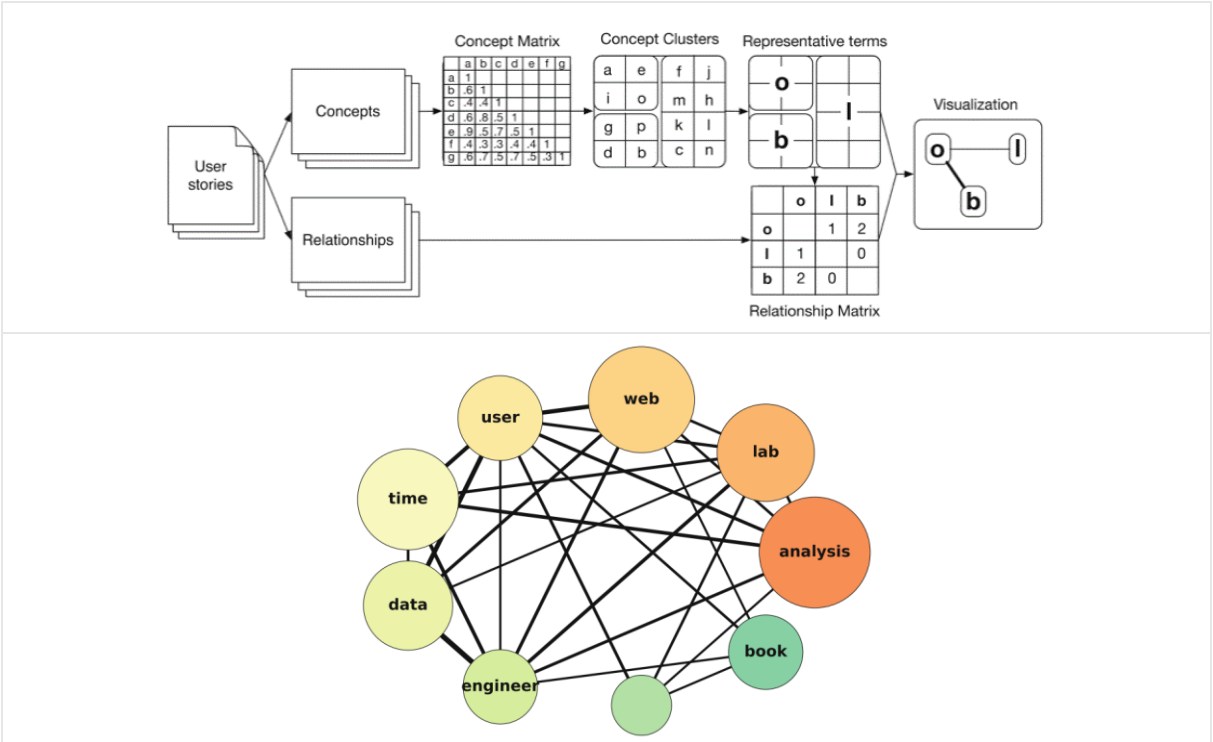


Figure 2.10: Figures from Lucassen's et al. 2016 paper on employing semantic relatedness algorithms to extract concepts from user stories. Top figure illustrates the model of their approach, while the lower figure shows an example result.

Before examining the team's next work, however, it is relevant to draw attention here to an interesting observation that this research has brought to surface. Extracting concepts (or conceptual models) from user stories appears to be receiving rising interest among SE researchers. The topic has particular relevance to our research motivation of capturing and representing design concepts, and then making them readily accessible to developers within the context of implementation artefacts. More discussion on this appears in Chapter 4. What is important to highlight here is the fact that this team's work has seemingly originated from initial requirements engineering interest—as perceived from the above two studies—to then progressively transform to requirements visualisation (employing conceptual modelling in the process). We find this of interest because it seems to reaffirm the argued cognitive potential behind metaphorical visualisation. This perception should become more evident after discussing the team's subsequent papers as follows.

In a following paper, the team presented an improved attempt to extract concepts from user story requirements using a heuristic NLP algorithm to automatically build conceptual models (Robeer et al. 2016). The concepts and their relationships are then visualised by their 'Visual Narrator' tool to create a holistic view of a domain's concepts (see Figure 2.11). The authors report a considerable accuracy and recall improvement of 80%-92%, with the resulting models being described by the original user stories' creators as useful artefacts to communicate and discuss requirements.

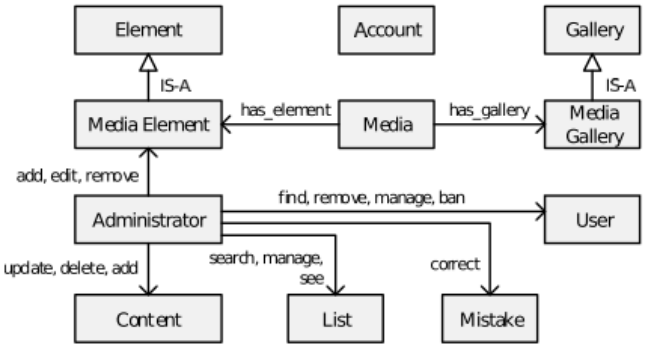


Figure 2.11: Partial view of the conceptual model generated by Visual Narrator. Source: (Robeer et al. 2016)

In 2017, the team presented an extended study of their work and introduce an improved visualisation of their conceptual models—now as interactive node graphs where users can zoom and filter on demand (Lucassen et al. 2017). In this paper the authors place particular emphasis on their motivation of bringing to the surface the key *entities* and *relationships* out of "lengthy textual specifications", presenting the user with an easy and accessible view of dependencies, redundancies, and inconsistencies in a holistic visual representation of system requirements (see Figure 2.12). An interesting remark to point out is that graph node representations do not scale well when visualising a

large number of artefacts as they quickly turn into dense and cluttered views, increasing the cognitive load rather than reducing it. This is evident from the view appearing in the lower part of Figure 2.10, as well as from other visualisations presented earlier from other works. This issue can be alleviated, however, by implementing aggregation and other dynamic techniques to control the level of detail presented to the user (Limberger, Scheibel, et al. 2017; Trapp et al. 2008; Balzer and Deussen 2007). Indeed, the authors report that in their earlier work, the generated conceptual models "quickly become too large for human analysts", and hence they resorted in this work to implement their interactive techniques to overcome the complexity by allowing the user to reveal details on demand. Interestingly, in concluding their paper, the authors implore on connecting their generated conceptual models to software architecture views in future work. This aligns well with the overall objective of our work, as detailed later.

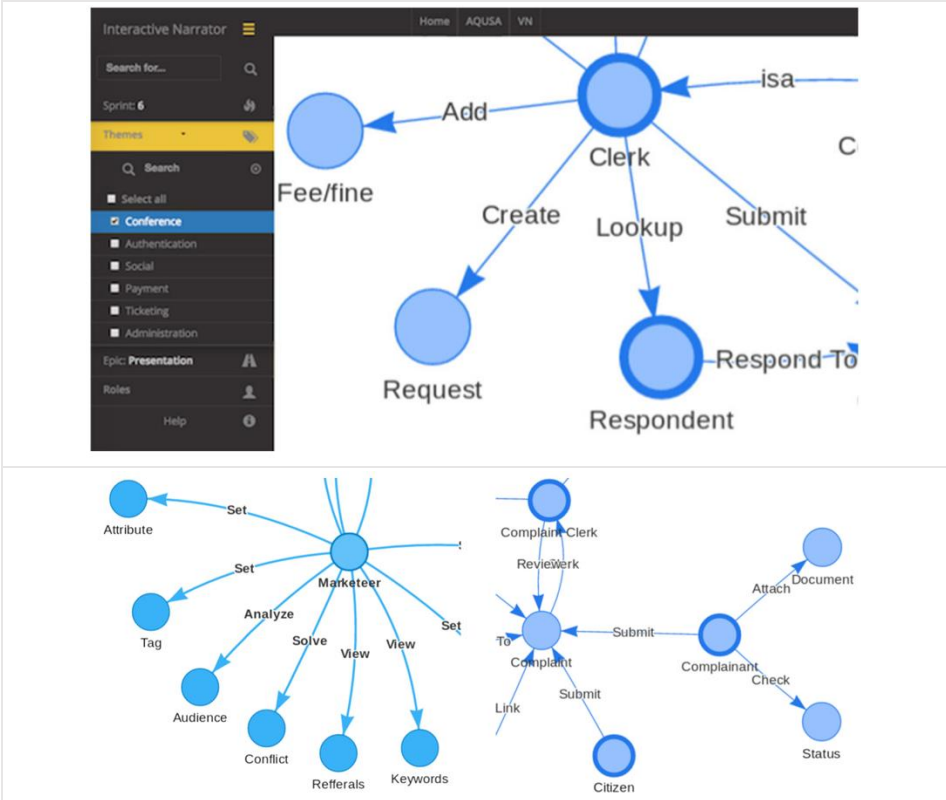


Figure 2.12: Visualised conceptual models as featured in (Lucassen et al. 2017). Top figure shows filtered concepts belonging to a selected Sprint, while bottom figure shows conceptual models generated from two real-world datasets.

In their 2018 paper, the team articulate their journey very well, summarising their original objective and the path it took them to arrive at a practical solution—a path that has strong parallels with the motivations and objectives of this work (although ours differs in approach, application context, and technology used). From a desire to help developers in forming comprehensible and accessible mental models of their agile requirements, the authors sought to abstract user stories into higher conceptual

models, and then informed by the REV literature, concluded by presenting them in interactive visualised forms (Slob et al. 2018). In this paper, they also introduce improved interactive functionalities to their tool that includes advanced filtering, using greying-out techniques to reduce visual clutter, and searching mechanisms (see Figure 2.13). In Chapter 4, we discuss our generalised approach to capturing design concepts irrespective of granularity, and synchronising them with their actual source code implementations.

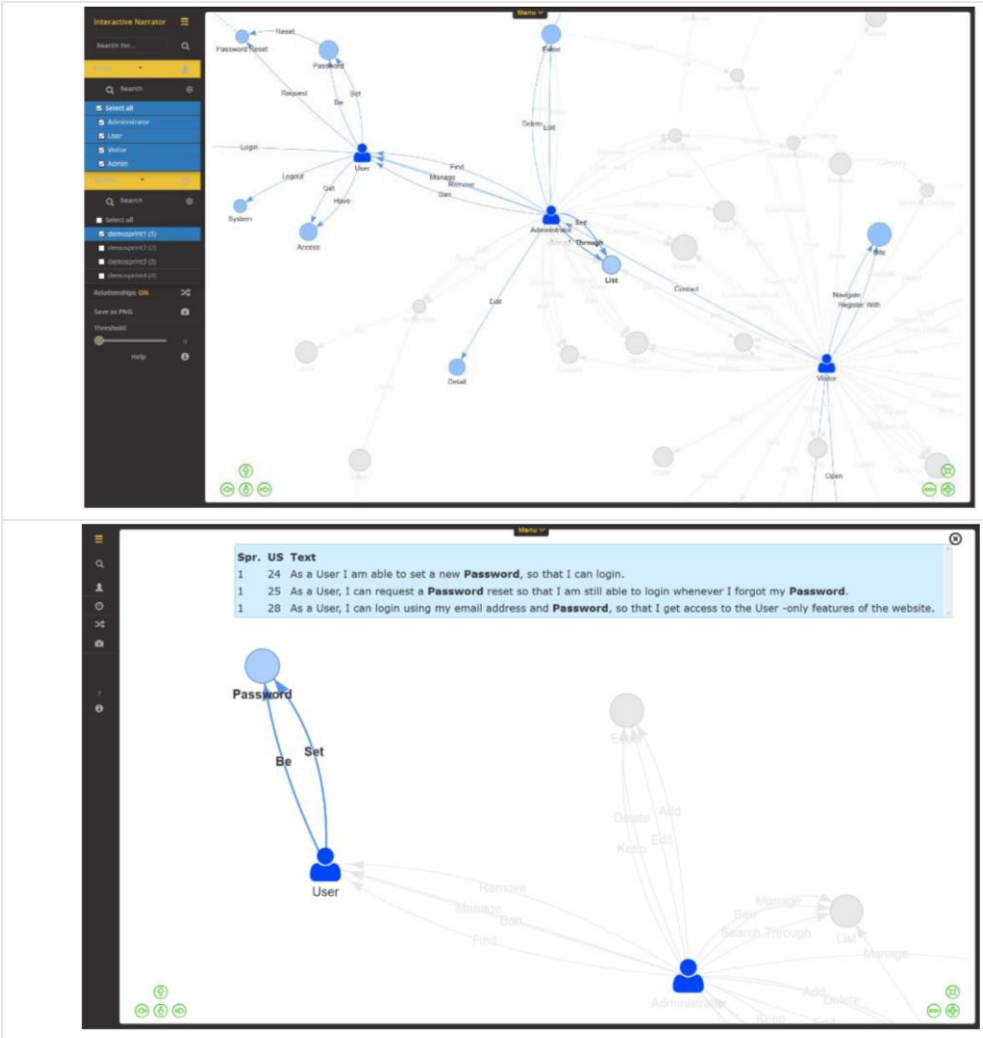
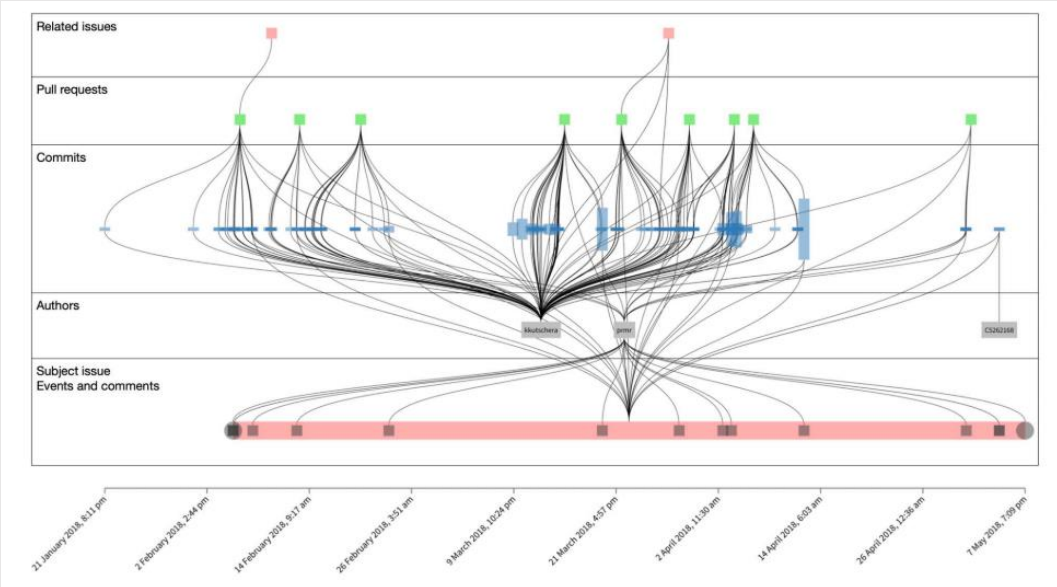


Figure 2.13: Enhanced visualisation functionalities of Visual Narrator as featured with concepts filtered for a particular sprint (top) and for a specific user role (bottom). Source (Slob et al. 2018)

Visualising Knowledge on GitHub. Research works in requirements engineering visualisation are not abundant, as indicated earlier in this section, and despite the fact that a few papers keep appearing slowly in literature, we had difficulty identifying recent publications that are of close relevance to this work. Nonetheless, there is at least one very recent work that is of particular interest. Concerned by the overwhelmingly textual nature of concepts and development artefacts found on GitHub (and other version control platforms), Fiechter et al. report in 2021 on the development of a visual analytical tool to help developers navigate through this dense and complex data (Fiechter et al. 2021). They introduce an approach to transform GitHub Issues into 'visual narratives', exposing in the process key information and event entities behind each issue (e.g., commits, actors, pull requests, timings, and relationships to other issues). The approach offers a global overall view of issues in a project, as well as a detailed timeline view of each individual issue, where users can browse and navigate interactively (see Figure 2.14). This work is found relevant because, under our generalised concept of Design Artefacts (see Chapter 4), we look at GitHub issues as only one form (among others) in which software requirements are expressed in today's development community.



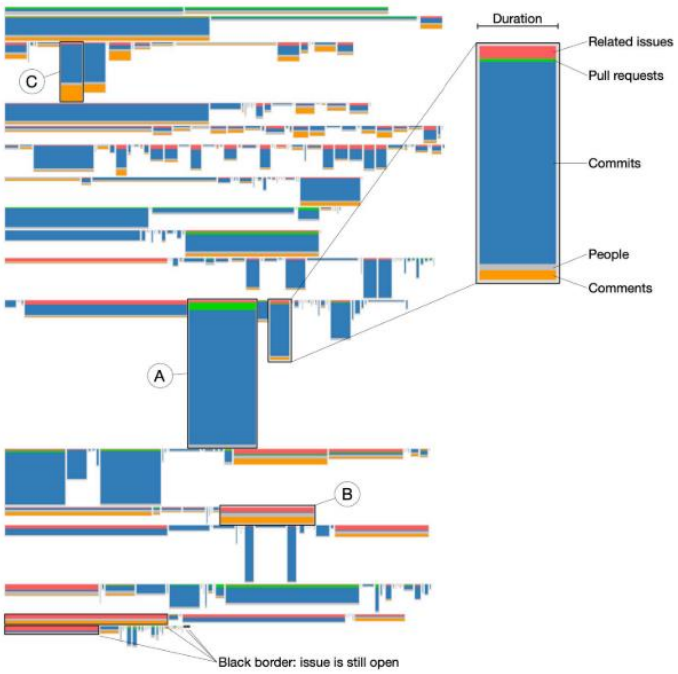


Figure 2.14: Detailed timeline view of a GitHub issue from a real-world open source system (top), with a global overall view of all issues (bottom). Source: (Fiechter et al. 2021).

2.2.3 Applications in Artefact Traceability (Traceability Visualisation)

Context. When discussing software requirements, requirement traceability is often a topic of importance in that general context. Moreover, just as visualisation techniques were studied to benefit the requirements engineering area, the same technology has naturally received parallel interest from researchers to utilise it for the benefit of requirement traceability. Traceability research can sometimes be about as simple as tracing functional user requirements to their eventual source code. More often and increasingly so, however, it involves traceability within various levels and hierarchies of requirement and design artefacts themselves, and it also extends to include various implementation artefacts. For example, (Cleland-Huang and Vierhauser 2018) studied traceability between source code and a specific family of hierarchical design artefacts in the context of safety-critical systems, while (Chen et al. 2018) studied traceability between source code and documentation. However, our perspective of interest here is in the techniques used to visualise those traceability links, regardless of the nature and context of artefacts between which traceability is being represented. Moreover, this work is also interested in visually representing traceability between multiple and varying software artefacts, irrespective of their nature. Hence, in this section we adopt the generic term 'artefact traceability'.

Focus. As indicated earlier, this work is not grounded in traceability research, which is a relatively old, extensively studied, and well-established domain. Thus, our literature discussion here is primarily focused on visualisation techniques as an area of research applied in the context of the traceability problem.

Background. Utilising visualisation in artefact traceability research appears—unsurprisingly—to have gained momentum around the same time that requirements engineering did. For example, in an early work published under the second REV workshop, Cleland-Huang and Habrat discussed visualisation in the context of evaluating traceability links that were automatedly reconstructed (Cleland-Huang and Habrat 2007). A common issue with automatedly reconstructed trace links is their varying levels of correctness, and the need for human input to manually accept or reject each calculated result. To assist in this task, the authors presented different visualisation techniques to cognitively aid the user. For example, they used an interactive node graph to depict the hierarchical structure of requirements, utilising techniques such as greying out, node sizes, shading, and color-coding to support an analyst in reviewing tracelinks' accuracy (see Figure 2.15). An interesting remark from the paper is the authors' statement about the inherent hierarchy in 'almost all documents' (in apparent reference to traditional user requirement documents), which they acknowledged as a presumption in using the hierarchical node graph to capture those requirements and expose their context. As will come to be evident shortly,

the way requirement artefacts are structured in modern agile practices makes hierarchical structures a straightforward candidate in capturing and representing those artefacts. Chapter 4 also treats this subject with more elaboration.

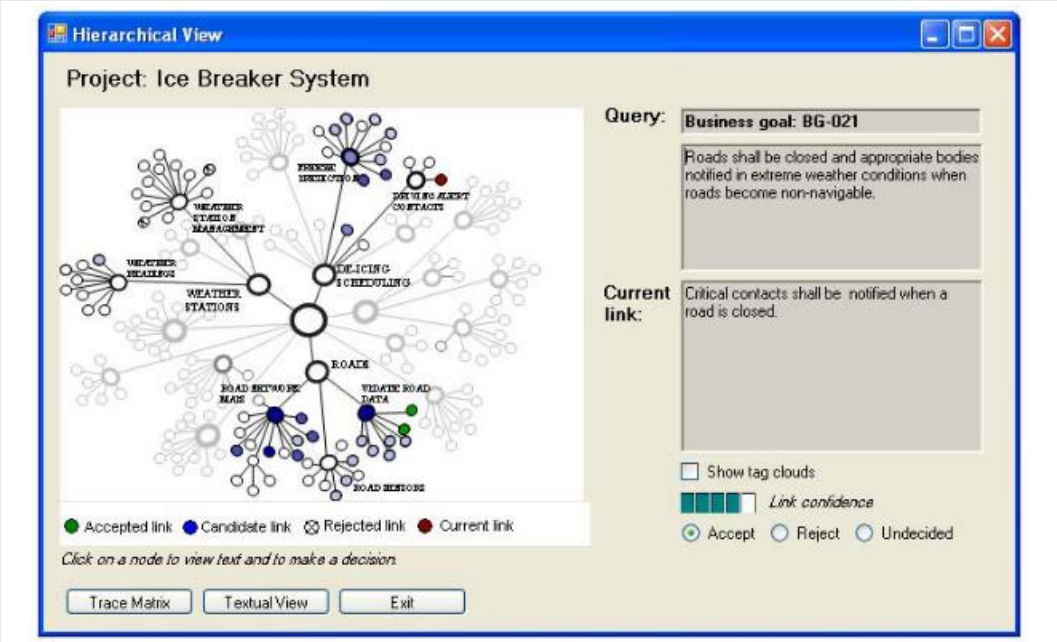


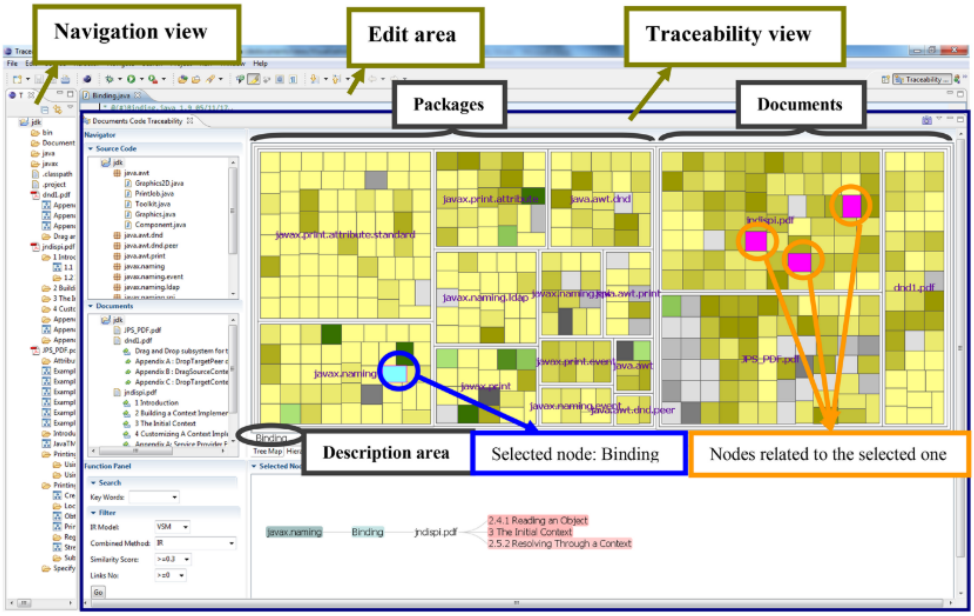
Figure 2.15: Hierarchical representation of traceability links between requirements. Source: (Cleland-Huang and Habrat 2007).

Modern Works. Moving forward to focus on more recent works, Chen et al. in their 2018 paper introduce an Eclipse-based traceability link recovery tool, in which they integrated two visualisation techniques to help their users in browsing, locating, and maintaining (editing) traceability links between Java classes and documentation files (broken down into specific sections). Instead of node graphs with large numbers of edges connecting them, the authors resort instead to utilising a treemap technique to visualise the artefacts as squares (or blocks) and then using colours to depict the relationships once a user selects an object of interest. The authors avoided drawing edges on top of the blocks to represent the traceability links, describing the commonly used approach as “giv[ing] rise to scalability and visual clutter issues”. This aligns with concerns raised earlier, with regard to node graphs leading quickly to overloaded scenes and tangled lines. To augment the treemap view, the authors provided a detailed view that further displays the relevant linked artefacts in small hierarchical trees (see the section labelled ‘Detailed HT View’ in the lower part of Figure 2.16).

As an alternative technique to the treemap, the authors implemented an interactive hierarchical tree visualisation to depict the structure of the artefacts (classes and document files). They specifically cite preserving of ‘context’ and reduction of visual clutter as motivations behind their adoption of the

treemap and the hierarchical tree structure, indicating that both mechanisms allow their users to locate and identify the artefacts of concern, while at the same time maintaining an overview of the overall system context (see Figure 2.16). However, by inspecting their treemap structure, it does not appear to provide particularly good context, as all blocks are of equal sizes, and are unlabelled. The hierarchical tree, on the other hand, appears to be more effective in representing the structural context. Results from their evaluation appear to indicate that the treemap view was more preferred overall. Nonetheless, participants found the hierarchical tree to be an especially useful supplement to the treemap view, stating that four participants found them to be "easier to use and easy to browse links", and two others describing them as "more intuitive and comprehensive".

Another interesting highlight from the work is the authors reporting that some participants found simply highlighting artefacts in a particular colour (to depict traceability) was not enough to make them noticeable to them. Another pointed out the unsuitability of this method to colour-blind users. The authors then contemplate using other techniques (such as enlarging the selected artefacts) to make them further stand out. This is interesting because in our work, we implement visual animation techniques (in addition to colour) to help the selected artefacts of concern visually stand out. Lastly, the paper was found to present a good historical overview of visualisation techniques used in the early works of artefact traceability visualisation.



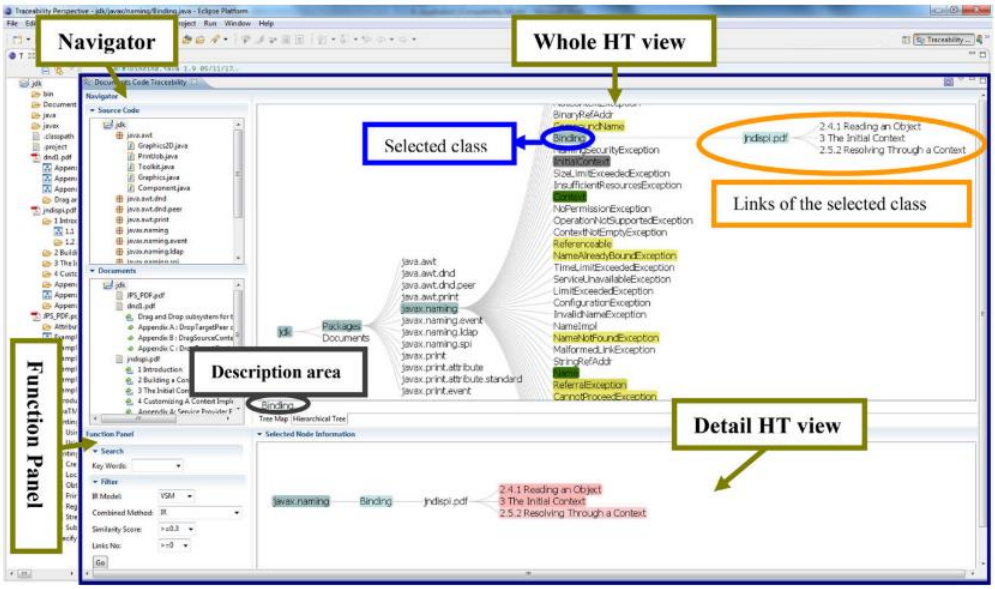


Figure 2.16: Treemap view (top) and hierarchical tree view (bottom) of the visualisation techniques used by the authors to depict tracelinks between source code and documentation artefacts while maintaining context in both cases. Source:(Chen et al. 2018).

Synchronising Artefacts. Rubasinghe et al. introduce in a 2018 paper a proposal to establish traceability across key phases of the software development lifecycle (I. D. Rubasinghe, Meedeniya, and Perera 2018). In their approach, they discuss three types of artefacts, 'Requirement Artefacts', 'Design Artefacts', and 'Source Code Artefacts', which are automatedly extracted via different algorithmic techniques and then stored into an XML-based artefact relationship model. Traceability links are computed based on a 'similarity algorithm'—and hence are subject to the precision issue—and then visualised using a node graph technique. Colours and labels are used to annotate and distinguish the various nodes and edges (see Figure 2.17). While the visualisation technique is considered to fall among the simpler approaches that lack the ability to represent context, do not support richer dimensions, and suffer from the same issues discussed earlier with regard to node graphs, nonetheless, the authors' motivations and objectives are similar to ours. As is evident from their design model (on the left side of Figure 2.17), the authors are clearly attempting to bring the various artefacts of the development process into a common unified model in order to "ensure the proper synchronization among software artefacts during the software development process"—a statement that strikes parallels with our motivating arguments.

In a following paper the same year, the authors extend their approach to support the DevOps practice by including 'test artefacts' and 'configuration artefacts' to their artefact traceability model (I. Rubasinghe, Meedeniya, and Perera 2018). Unfortunately, no attempt is made or discussed with regard to improving their visualisation technique, with the resulting node graph appearing now to clearly show its visually overloaded and cluttered nature—despite only part of it is being shown (see Figure 2.18)

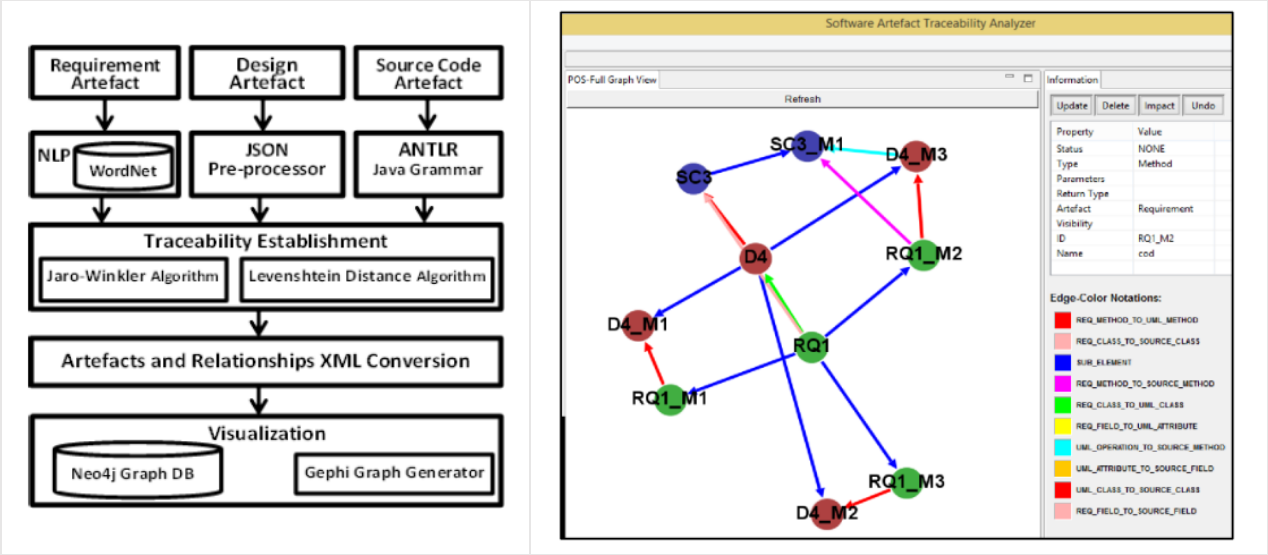


Figure 2.17: An ontological abstraction model used to establish a heterogeneous set of artefacts (left), with a node graph visualisation used to depict traceability between the various artefacts. Source: (I. D. Rubasinghe, Meedeniya, and Perera 2018).

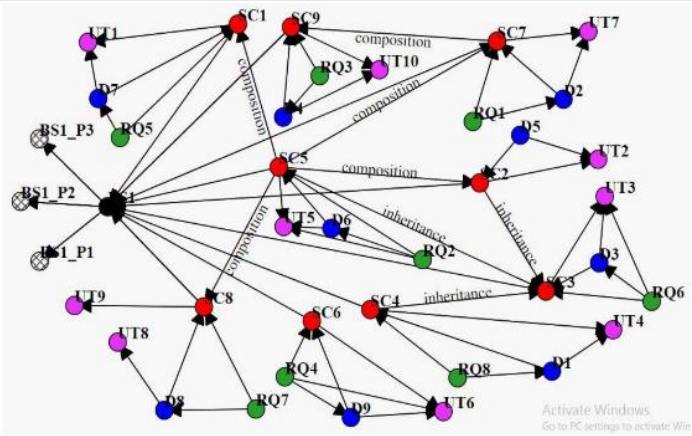


Figure 2.18: A partial node graph showing traceability links between various types of artefacts (requirements, source code, unit tests, and design) as implemented by Rubasinghe et al. It illustrates the visual overload that node graphs typically lead to, especially when not aided by an interactive mechanism to control the degree of information being displayed. Source: (I. Rubasinghe, Meedeniya, and Perera 2018).

Introducing the Third Dimension. In a short 2019 position paper, Aung et al. complain about the limitations of the two traditional approaches to representing traceability links, referring to them as either a bland textual tabular format (in reference to the 'Requirement Traceability Matrix'), or two-dimensional node graphs, stating that both formats make it challenging for software engineers to explore the possibly large number of traceability relations in a system (Aung, Huo, and Sui 2019). As an alternative, they report on a conceptual tool in which they attempt to utilise the third dimension and a form of hierarchical map to alleviate the issue. Their approach focused on depicting traceability links from a single type of 'source artefact' to two types of 'target artefacts'[12], where 'use cases' represent

[12] The words *source* and *target* here refers to the common terminologies used in traceability research with 'source' representing the starting origin of the link and target representing its endpoint.

the source artefact here, while 'test cases' and 'source code' represent the target artefacts. This trio relation is what they describe as a 'hierarchical trace map visualisation'. Each type of artefact is then represented as nodes grouped in one plane, creating three planes in total, with the source artefact placed on top. Nodes are identified with colours and labels, and tracelinks are depicted by drawing lines between the related artefacts (see the left side of Figure 2.19). Searching, filtering and other interactivity is implemented to help the user focus their view on individual artefacts (see the right side of Figure 2.19). This work is of interest in that the authors express similar concerns about the traditional node graph visualisation techniques that we raised earlier. It is also pertinent that they acknowledge that bringing a hierarchical structure and using the third dimension could be a possible way forward to reduce the cognitive overload of visualising traceability links. Unfortunately, despite attempting to use both features, their approach does not seem to have overcome the visual clutter and visual overload problems, and it appears to fall back to the same node graph technique (more views can be seen on their accompanying GitHub page). We argue that one reason for this could be the fact that the authors did not use an actual artefact structure as a baseline for their visualisation and hierarchy, forcing them to fall back inevitably to the structure of the 'traceability network' itself, which is inherently disorganised and non-hierarchical. More explanation and discussion on this point appears at the end of this section. Another issue with their approach is that labels of the artefact nodes could be improved to reflect meaningful names to better guide the user.

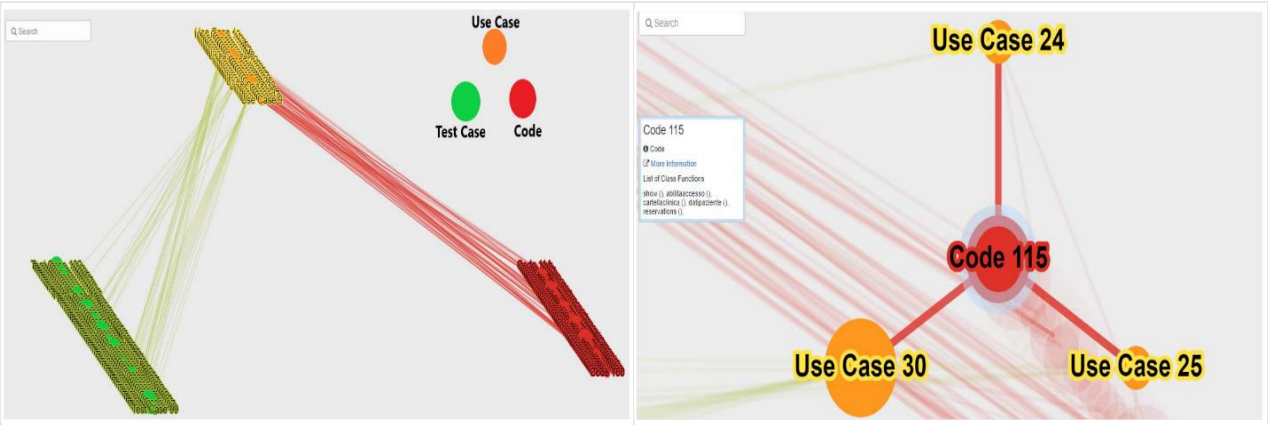


Figure 2.19: An overall visualisation of traceability links between use cases and target code or test artefacts (left), and a filtered view focusing on a particular code artefact (right). Source: (Aung, Huo, and Sui 2019).

Visualising Traceability Within Issues. Traceability sometimes becomes relevant between the same family of artefacts. Lüders et al. proposed in a short paper an approach to visualise the links between 'Issues' in a Jira system, creating an 'Issue Link Map' (Lüders et al. 2019). Motivated by the fact that issues in Jira can have nested sub-artefacts, as well as relationships to other issues, the authors implemented a node graph visualisation to allow their users to "see all linked issues at a glance" rather

than having to click through the links one after another on the Jira site to explore the relationships (see Figure 2.20). Their Jira-plugin is said to help users manage their issues and detect, missing, unknown, or inconsistent links. This work illustrates again the popularity of the node graph technique in artefact traceability visualisation works, despite its limitations.

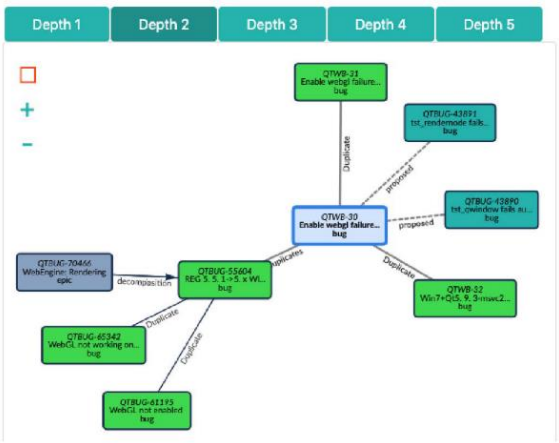


Figure 2.20: Visualisation of Links between Jira Issues, as appearing in (Lüders et al. 2019).

Utilising Eye-Tracking. In a different and so rather interesting approach, Ahrens reported in 2020 on a work-in-progress project where he attempts to utilise eye tracking software to automatically capture tracelinks between different artefacts (in this case, documentation and source code) based on detecting the artefacts being focused on and interacted with (Ahrens 2020). The author then uses overlayed heatmaps to visualise areas of interest for a certain task or user role, with colour scales indicating the level of relevance. The technique is applied to a source code file open in an Eclipse editor, to file names in Eclipse's file explorer, as well as to requirement files (see Figure 2.21).

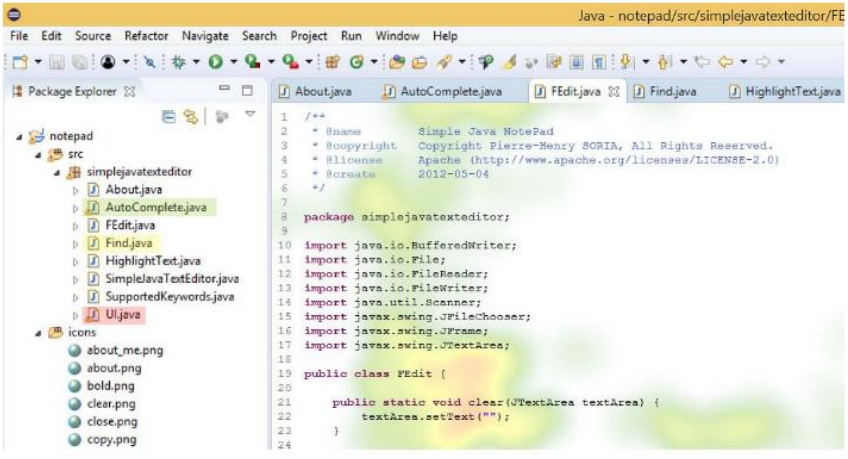


Figure 2.21: Visualising tracelinks between source code artefacts and certain user tasks or role through color-scaled heatmaps. Source: (Ahrens 2020).

Using eye tracking tools to automatically capture tracelinks between artefacts is well-known in the traceability literature, with the research tool 'iTrace' being a popular research tool that is commonly

used (Abid, Maletic, and Sharif 2019). However, this work is of particular interest in that it attempts to utilise the collected data to present a visualisation of the trace links and associate it with particular user tasks and user roles.

Supporting Code Review. In a somewhat different context, Sülün et al. have used node graphs to supplement their Reviewer Recommendation system (Sülün, Tüzün, and Doğrusöz 2021). Utilising the information available on GitHub, they augment their recommendation system to present a visualisation of source code artefacts linked to GitHub *changesets* and *issues*, which is then used to recommend particular reviewers for specific artefacts based on reviewers' knowledge of that artefact as computed by the system. Information from the generated graph (such as distance between nodes) is utilised in computing the candidate reviewers. The authors also describe their node graph visualisation to help potential reviewers in examining possible impacts to other artefacts (see Figure 2.22). While this paper does not hold strong relevance to the context of our work, it serves to demonstrate the predominance of node graphs in the field—apparently facilitated by popular graph databases (e.g. Neo4j) that makes it relatively easy to produce these visualisations.

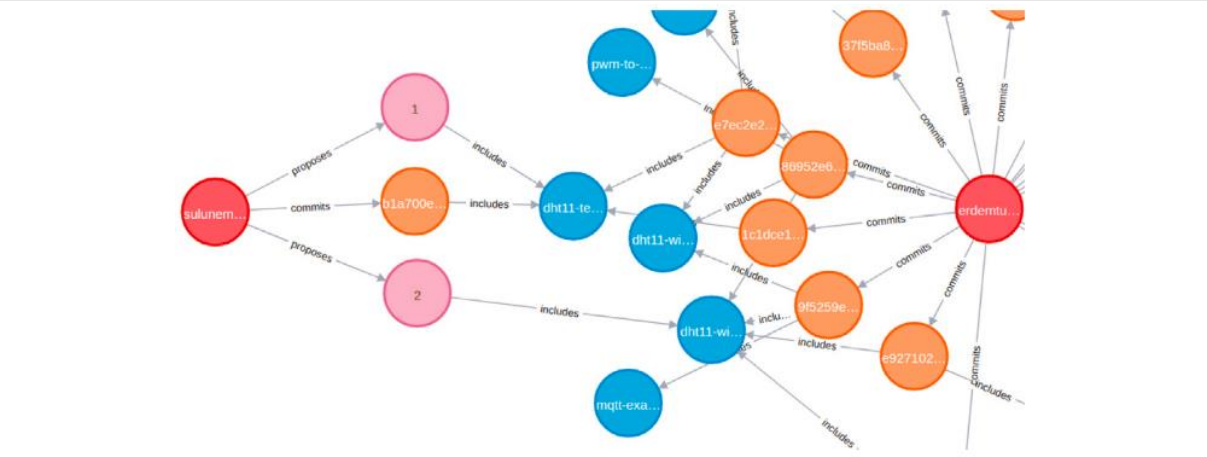


Figure 2.22: Visualisation of GitHub changesets, issues, files, and users as part of an artefact reviewer recommendation system. It serves to illustrate the predominance of node graphs in traceability visualisation research. Source: (Sülün, Tüzün, and Doğrusöz 2021).

Hierarchical Node-Graphs. In extended research work featured across a few publications, Cleland-Huang and colleagues report on a traceability analysis framework that particularly addresses traceability requirements in safety-critical systems. In their 2021 paper, they focus on presenting the visualisation aspect adopted in their tool (Cleland-Huang et al. 2021). Their traceability information model (TIM) consists of requirement and safety artefacts (e.g., user stories, safety stories, design constraints, hazards, design rationale), as well as implementation and test artefacts (e.g., source code, acceptance tests, test logs), as retrieved from Jira and GitHub. The framework is described to suit

traditional safety-critical systems as well as other agile projects that need to incorporate safety-assurance practices. The authors utilise node graph visualisation techniques to provide their users with clear visual depictions of traceability links for a particular artefact of interest. Instead of visualising the artefacts and their links in the typical 'nodes and edges' format, and instead of visualising all artefacts and their relationships at once, they use rather a semi-hierarchical tree (named a slice tree) to only visualise the traceability links around one 'entity of interest' at a time. For example, in the left side of Figure 2.23, a visualisation is automatically generated for a selected hazard (UAV-1364), where its related artefacts, such as safety stories, design constraints, and implementation source files, have been produced based on the rules defined in their TIM model. Such a view is said to allow users to monitor the progress and the extent of realisation of a particular hazard. Other views are also implemented to support different activities—for example, the view on the right side of Figure 2.23 allows users to focus on a specific code artefact and then visually examine all related artefacts from design up to safety and hazard artefacts that are impacted by that code artefact—hence facilitating refactoring and change impact analysis. Users can also interact with the visualisations to display additional information or confirm a suggested artefact to be linked.

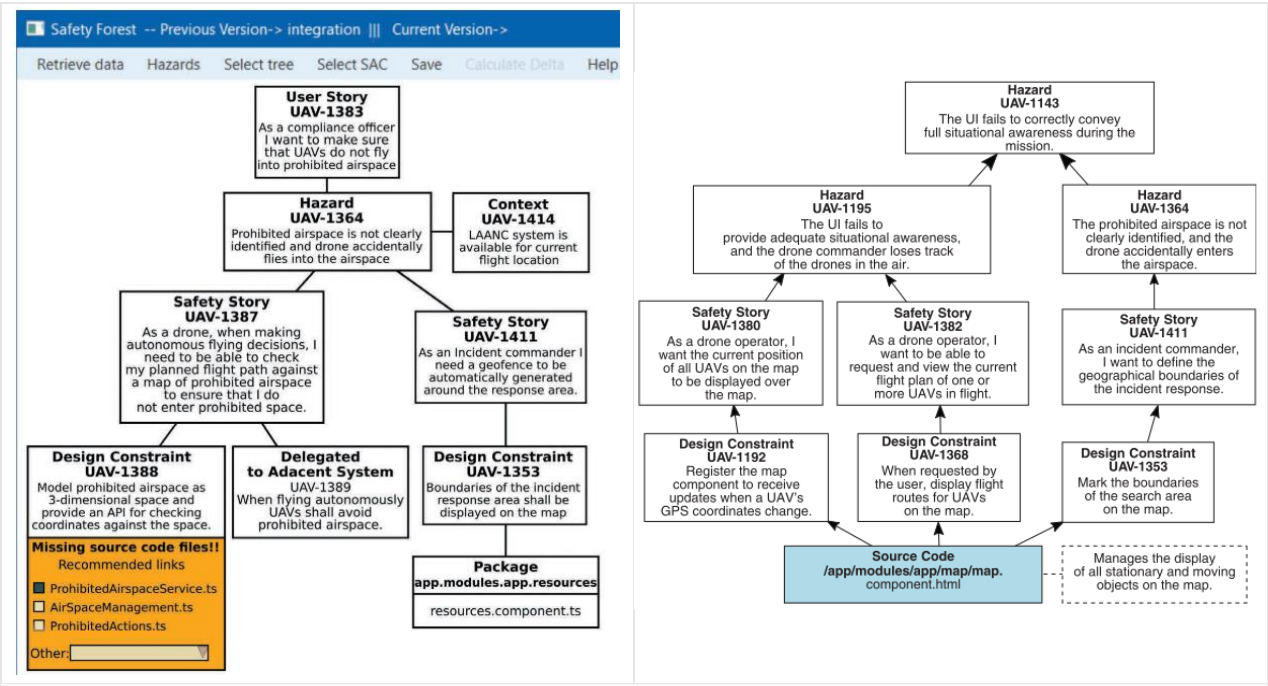


Figure 2.23: Visualisation of safety artefacts as implement in the SAFA tool with traceability links depicted for a selected hazard 'UAV-1364'(left), and another for a selected code artefact (right). Source: (Cleland-Huang et al. 2021).

In contrast to many other node graph visualisations that we have seen earlier, the visualisations here are remarkably uncluttered and appear to communicate their message clearly. We believe this is attributed to two factors referred to previously. First, the visualisation depicts the related artefacts' traceability links around one particular item of interest at a time—which in terms of practicality, aligns

well with how users would typically use such a tool to answer a real-world question—but more importantly, it avoids overloading the user with a forest of nodes and links. However, context is lost here as the user is unable to see the overall picture (both at design, as well as from the perspective of implementation). The second factor is that the node graph here is of a semi-hierarchical nature, giving it a familiar and more comprehensible quality as opposed to the structure-free instances seen in earlier works.

Hierarchical Node-Graphs with Method Level Traceability. Automated link recovery (or rediscovery) appears to be predominant in traceability research, and similar to almost all previously discussed works, Aljawabrah et al. presented in 2021 their automated approach that focused on recovering tracelinks between code artefacts and unit tests in particular (Aljawabrah et al. 2021). The authors describe that in practice, such links are often manually identified and maintained, requiring time-consuming and lengthy comprehension efforts from developers. Of particular interest to our context here, is that they resort to utilising visualisation techniques to present the reconstructed tracelinks to their users—both to better comprehend the relationships and to maintain them. They use a node graph technique similar to other works reviewed above, but they adopt a strict hierarchical tree structure, resulting in a clean and well-organised visualisation. The hierarchical structure is based on the actual structure of the unit test artefacts (essentially the source code structure). Also of interest is that the authors establish their tracelinks at the class and method levels of their artefacts—an approach that is empirically validated as offering users better granularity and accuracy (Ali et al. 2015; White, Krinke, and Tan 2020), but yet is seldom adopted in the literature. In this regard, our generalised abstraction model discussed in Chapter 4 describes our adopted approach to enabling multi-granularity tracelinks between any type of design and implementation artefacts, without restricting users to a particular paradigm or model. The authors present a number of visualisation views to support different development tasks (see Figure 2.24). Their tool is implemented as a stand-alone .NET application and is currently working for C#-based programs, with future plans to extend to other languages. The approach works by accepting a .NET 'solution' file and appears to process its extraction and computations without resorting to middleware storage mediums—however, this could not be confirmed as the authors do not explicitly discuss it. Lastly, it must be highlighted that, similar to the above work of Cleland-Huang et al., the fact that the authors opted to design their visualisation technique to be focused on one particular artefact of interest at a time, rather than displaying the tracelinks for the entire system all at once, appears to have significantly contributed to the clean and visually comprehensible results that both teams obtained. Indeed, both works share a similar visual appearance in their resulting visualisation techniques. Also, being able to focus on a small subset of

artefacts at once seems to have enabled both teams to design their nodes as information bubbles, resulting in a graph that is self-explanatory and easier to interpret by users.

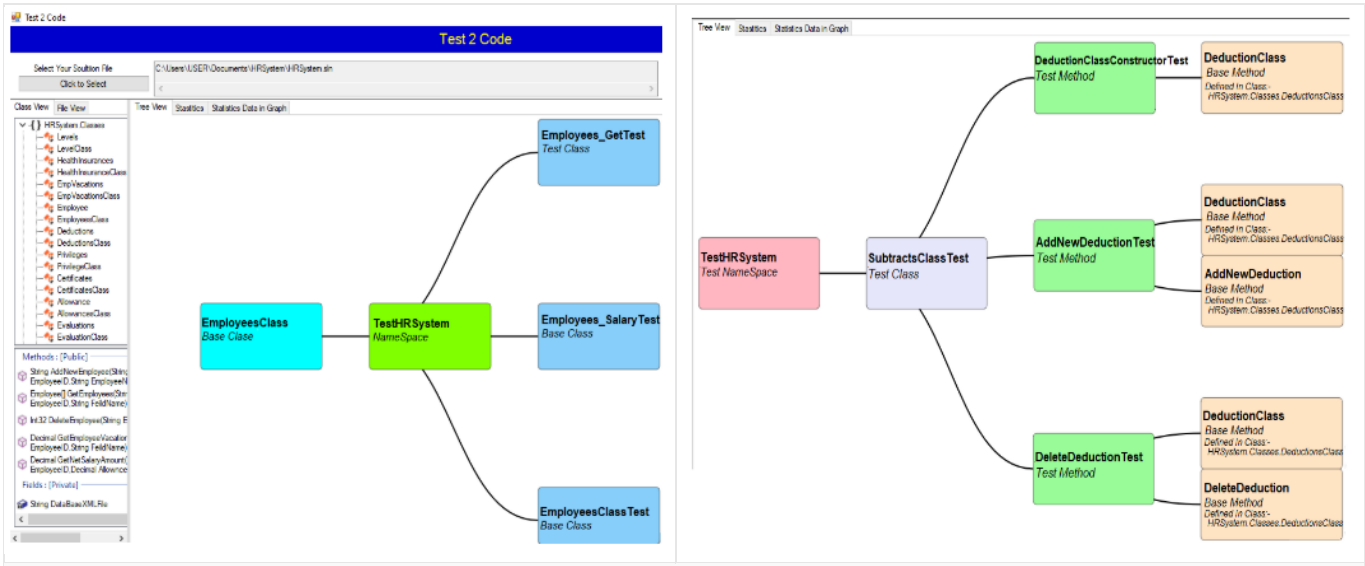


Figure 2.24: Two views of test-to-code tracelink visualisations. On the left the view starts with a particular implementation class and shows its related test classes (class-level view). On the right the view starts with test class, and shows its related implementation artefacts (tested artefacts) at method and class levels. Source: (Aljawabrah et al. 2021).

Visualisation in Product Service Systems. Before closing this section, it is worth noting that visualising artefact traceability has also seen interest in relation to embedded and physical systems. For example, Wolfenstetter et al. discuss visualising artefact traceability in the context of Product Service Systems, where they introduce a high-level abstraction model they name 'Model Integration Ontology' to collect a heterogeneous set of artefacts into a unified common model (Wolfenstetter et al. 2018). They then use this model to present node graph visualisations depicting the traceability links between the various artefacts (see the right side of Figure 2.25). The work is found to be of interest because the ontological model introduced is an attempt to capture artefacts of a very different nature and domains (e.g., physical system components, service component, environment resources, as well as software artefacts such as requirements specifications and diagrams) into a single abstract representation so that tracelinks can be practically established between them (see the left side of Figure 2.25). This is found to share parallels (in terms of objectives) with our generalised abstraction model of the design and implementation artefacts as discussed in Chapter 4—admittedly, the context is much greater here though. Nonetheless, the authors rely on an XML-based format to capture and store the data, whereas the abstraction we introduce is ephemeral in that it is used to capture the artefacts (from their original resource) during runtime only, avoiding the need to store them in any intermediary media. This study also serves to demonstrate that visualisations based on node graphs (with edges linking the nodes) appears to be a predominant technique in representing traceability links across the greater 'systems development' disciplines, and not just the traditional software development domain—despite the cognitive and scalability challenges that this technique presents, as we have discussed earlier.

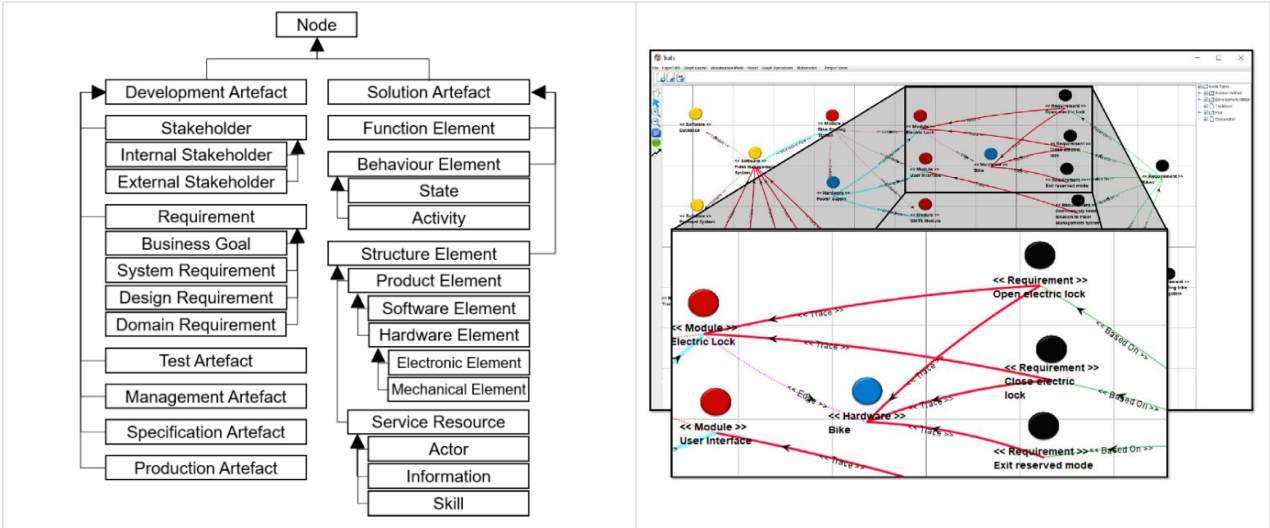


Figure 2.25: An ontological abstraction model used to capture a heterogeneous set of artefacts (left), with a node graph visualisation used to depict traceability between the various artefacts. Source: (Wolfenstetter et al. 2018).

Closing and Lessons Learned. Below are a few reflective remarks outlining overall insights, findings, and lessons learned from the above literature reviews.

- Node graphs appear to dominate the artefact traceability literature, despite their apparent (and academically reported) limitations, particularly in terms of scalability and limited dimensional depths. They also often result in visually overloaded scenes and tangled edges that defy their intended cognitive scaffolding purpose.
- Using a form of hierarchical structure to guide and model a node graph appears to significantly improve their effectiveness and comprehensibility. Instead of a chaotic and unstructured appearance, it gives them a familiar and organised look, making it easier for users to follow and interpret the visualisation.
- Interactive techniques and other creative ways to control the number and size of artefacts displayed at one time can also alleviate the visual overload issue, and the practicality of the visualisation. This can include filtering, aggregation, on-demand details, and so on.
- The majority of works discussed in this section appear to apply the node graph directly to their traceability links network, inadvertently exposing in the process the structure of the Traceability Network itself. The problem with this approach is that this network is inherently not governed by any meaningful structure that the user could relate to, and is largely chaotic in nature. It also does not represent any actual artefact structure that is meaningful to the user.
- There appears to be hardly any practical use of exposing the entire traceability network of all artefacts all at once. Developers and other stakeholders are normally interested in the traceability links of a specific item (or set of items) at one time.
- Visualisations appear to be more meaningful and comprehensible when they are based on an actual artefact structure that carries some meaning to the user and in the real world. Traceability can then be visualised by choosing a mapping technique to complement this structure. This mapping could be represented using colours, distinguished borders, shapes, and the like.
- An advantage of the above method is that it exposes the traceability while retaining the context within the artefact medium it is mapped onto.
- The work of (Cleland-Huang and Habrat 2007) and that of (Chen et al. 2018), which were discussed earlier, appear to attest to this argument. For example, Cleland-Huang and Habrat used the structure of the documents as a baseline and exposed it through a hierarchical node graph, resulting in a clean and relatable structure. Tracelinks were then simply 'mapped' using colouring and shading schemes across those artefacts. Similarly, Chen et al. used the structure of the documentation files in one view and the structure of the Java packages in another (which are both hierarchical in nature) to give a meaningful baseline structure to their visualisations.
- Our work uses the actual structure of the source code not just at the package level, but it also exposes the structure of individual code elements inside each file. It then uses this structure, which is familiar and relatable to users, as a baseline on which to map the traceability links. It further visualises this structure using a metaphorical and dimensionally rich representation that is empirically supported by literature for its cognitive advantages. All three techniques are thought to be novel.

2.3 Other Relevant Work—Traceability Research (Non-SV)

In this last section, we discuss a number of related traceability research efforts that are not particularly centred around visualisation. The works that are discussed are also centred around two research groups. Interestingly, both groups did resort at some stage to utilising visualisation techniques to aid their users, and the works in which visualisation was introduced have been covered in the earlier sections. This section covers the other aspects of their work from a software traceability perspective.

Capturing Tracelinks from Developers. In a 2013 paper, Delater and Paech report on a study to "(semi-)automatically create links between requirements and code during development" (Delater and Paech 2013c). Their approach defines a Traceability Information Model (TIM) that identifies and binds artefacts across three areas: System Artefacts (Features and Functional Requirements), Project Artefacts (Sprints, Work Items, Developers), and Code Artefacts (Files and Revisions). Figure 2.26 illustrates this model. A key aspect of their approach is to capture these artefacts semi-automatically while a developer is working on a particular work item, and then use an algorithm to "infer" the links once the developer changes the status of the work item from "assigned" to "done". The inferred tracelink is established between the changed files (Java in this case) and the requirements (features and functional requirements). The approach is implemented as an extension to the UNICASE Eclipse plugin, which is used to facilitate the capturing of the tracelinks.

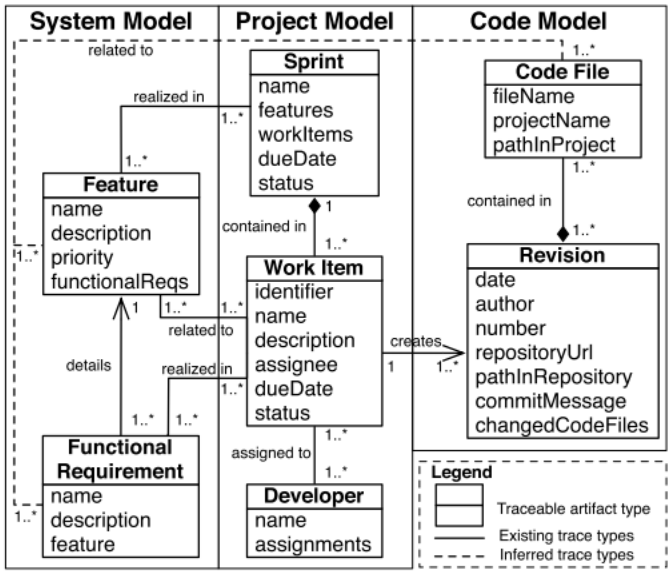


Figure 2.26: A Traceability Information Model as defined by Delater and Paech. Source: (Delater and Paech 2013c).

In a later paper that same year, the authors present further details on their work and report on an empirical study conducted with undergraduate students (Delater and Paech 2013d). In this paper, they described three ways in which a developer can select a work item in order for the process to capture

the code that they are working on—these are *before*, *during*, or *after* they start their implementation. The first method captures the *requirements* that the user has looked at during their work, and then links them to the selected work item[13], which is in turn linked to a revision in the version control system. The latter two methods cannot capture the requirements since the user has already started, and thus in both these cases the link is created between the work item and the revision only—i.e., no link is established to requirements. It is understood that a manager will need in the latter two cases to link that work item to the requirements it relates to. The authors appear to focus on the approach and no details are provided on how the process is implemented or how the work items are retrieved and presented to the user within Eclipse. Nonetheless, they described that they use the "EMFStore to access all artifacts per sprint and at the end of the project", which is facilitated through the UNICASE plugin on which their tool relies. The tool is validated over three undergraduate student projects with results indicating high precision and recall for the inferred traceability links. The results also showed that participants showed good interest in using the tool and found it easy to use. Interestingly, the first method was the least used (5-10%) meaning that users preferred to select a work item only after or during their implementation work, despite the first method resulting in higher accuracy.

Discussion. The above work shares parallels with ours in its goal of capturing tracelinks directly from developers during their work, and in utilising tool support in the process. In general, it also shares similarity in the notion of incorporating version control systems, and requirements defined in a management tool. However, the two works differ greatly in concept and actual approach. First, the authors above define a particular model for their artefact traceability model, which specifically relies on the concept of a Sprint and a Work Item. For example, in their approach a Work Item acts as an intermediary artefact that can implement multiple Features (or requirements), and a Feature can be implemented across multiple Work Items (see Figure 2.26). Our approach, on the other hand, adopts a simpler and more abstract view of software artefacts, and only defines a generalised concept of a Design Artefact and a Code Artefact, as covered in Chapter 4. It is designed to accommodate different development practices, with no presumptions or pre-requisites to a pre-defined model. In fact, the authors' approach ends up with three types of tracelinks as a result of their adopted model: work items-to-code, requirements-to-work items, and requirements-to-code—where the first two types are created manually by the developer, and the last being inferred automatically. On the other hand, our approach has a single type of tracelink, which is Design Item to Code Item, and it is captured/acknowledged directly from the developer. Second, their approach establishes tracelinks at file level only, and has been developed around Java[14], whereas our work enables tracelinks at multi-

[13] The user is asked to validate and edit the captured items.
[14] The authors do mention future plans to support other languages.

granularity levels (file, class, function, method, and so on) and is implemented to work with any source code, irrespective of language or paradigm. Third, despite capturing parts of the tracelink knowledge from the developers directly, the authors' implementation does not seem to capture the complete information required as the authors still resort to using an automated inference algorithm to arrive at (or complete) the final traceability links. Lastly, the approach above is designed around the UNICASE Eclipse plugin and requires the requirements and work items to be defined there, whereas our work is designed to accept any design items defined in modern agile dashboard management tools, as long as those dashboards have public facing APIs. More importantly, the authors do not discuss any mechanism for presenting the collected tracelinks to the users, or how they make them available to use and support software tasks.

Using Tags. In a 2017 research preview paper, and working from a feature management context, Seiler and Paech present a discussion on using tags to manage 'features' across issue tracking systems and version control systems, where they conducted eight expert interviews to understand current practice and to evaluate their idea (Seiler and Paech 2017). The authors propose an early concept of their model, in which they define three key requirement artefacts: 'Feature Description', 'Requirement', and a 'Work Item' that are stored in Jira. They propose then to 'relate' these artefacts by tagging them with a label that corresponds to a high level 'Feature' of the system. That label is then also used to tag implementation code files at either 'class' or 'method' level. As the work was still at the development stage, the authors did not elaborate on where the high-level system 'Feature' should be stored nor how its label tag would be obtained. Figure 2.27 depicts their conceived approach.

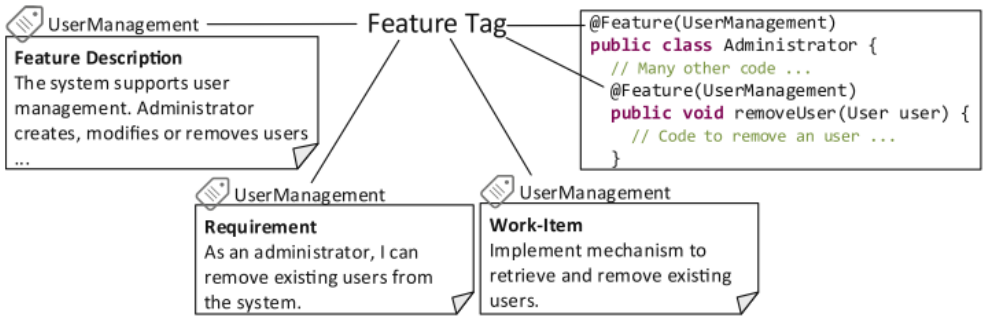


Figure 2.27: A concept proposal of using 'Feature Tags' to establish traceability as described by Seiler and Paech. Source: (Seiler and Paech 2017).

In 2019, Seiler et al. presented further progress on their work reporting a case study designed specifically to investigate how their earlier proposed approach compares to two other existing approaches of establishing traceability—an interaction logs approach, and a commit-based approach (Seiler, Hubner, and Paech 2019). The commit-based approach is common among the development

community, where developers mention the id of the issue they are working on in their commit message. The interaction logs approach is based on capturing user events during development. The feature tagging approach—the one under study—was facilitated by a plugin for Jira and another plugin for Eclipse. The authors describe their Jira plugin as providing "a recommendation engine to suggest feature tags for issues by processing the issue description and presenting the most likely feature tag for an issue to the students", while the Eclipse plugin is described to "complete(s) a feature tag in the code when typing parts of the tag name". Moreover, they add that a tag "summarizes the feature in a short and concise manner with a descriptive term" and that "tags are manually defined during development, prior to their application". No further implementation or technical details are provided with regard to the Jira or Eclipse plugin. More importantly, however, the authors appear to have dropped their earlier proposed class- and method-level tagging in this study, applying it only at file-level instead. This is evident from their statement: "we created a trace link between a requirement and a code file if both the requirement and the code file have the same tag applied".

Overall, their results indicated that the interaction logs approach had the best precision and recall (albeit describing it as intrusive), followed by the commit-based approach. However, of more interest to this work is their finding that the tag-based approach required the least effort. Interestingly, the authors do not correlate or comment on the reduced accuracy in their tag-based approach to the fact that their tagging was only applied at the file levels. Also, of particular interest is their finding that the type of code file had a direct influence on the precision and recall, as it seemed "easier for developers to provide commit ids for java files than for xml files"—an issue that our approach (see Chapter 6) avoids through automation.

Discussion. The above two works share similarity with our research in terms of the common notion of using tags to facilitate traceability. However, the actual approaches differ in concept and application. Moreover, while the authors do propose a method- and class-level concept, this does not seem to be realised in implementation. Our approach, on the other hand, implements tagging at any code structure level regardless of the source code language or its paradigm. It also does not require a pre-existing model for the requirements structure, and does not require the tags to be manually defined. It instead leverages the identifiers that already exist, and that are in common practice among agile practitioners. Most importantly, the tagging in our approach is not done manually, but as part of a three-legged operation that is performed automatically, and that covers the issue tracking system, the source code, and the versioning repository. While the authors mention that their tags are auto-completed once the user starts typing directly in the source code, the completion does not seem to be retrieved from the issue tracking system, as they added that "only feature tags, which have been applied to the code

previously are completed". In our approach the issue tracking system is connected live to the IDE, which auto-completes the tags in real-time, along with displaying other supplementary and contextual information. Lastly, we could not unfortunately establish confidently whether the high-level system features the authors referred to were defined or stored somewhere, or were simply an abstract concept (the authors appear, particularly in their earlier work, to refer to them as separate from those artefacts stored on Jira). As for the Feature Tags, they are said to be created in advance by developers to summarise a feature—indicating an added effort—but no example was provided of such tags.

Lightweight Trustworthy Traceability. In a 2014 vision paper, Cleland-Huang et al. described the current state of available trace data as 'neither trusted nor trustworthy', attributing it to a 'heavy-weight' process that produces untrusted tracelinks (Cleland-Huang, Rahimi, and Mäder 2014)[15]. They proposed instead a 'philosophical change' to the traceability landscape by encouraging 'light-weight' solutions that are more agile, favouring 'disciplined' and 'just-in-time' approaches over post-event heavyweight approaches. While the authors' proposed approach is very different in nature from that introduced here[16], nonetheless we find their motivational statements to align very well with our approach and the arguments behind it. In our approach we strive to encourage and enable accurate and real-time documentation of tracelinks from day one. It is accurate because it captures (and confirms) the traceability knowledge from the original developer just as it occurs in real-time, thanks to an automated process with a minimal operational footprint. We also seek to make the benefits available to users right away, encouraging them to develop a disciplined practice. Interestingly, the authors recognise the importance of the 'human component' as a distinguishing characteristic of their proposed approach. Our approach looks at the human as the original source of the traceability knowledge—e.g., a developer takes a requirement and produces a set of artefacts that realise that requirement—and hence it is only sensible to capture that knowledge immediately from its source, rather than working to rediscover it afterwards. We argue that such disciplined practice can be made feasible with proper tool support, and thus has promise in avoiding the flawed, missing, and incomplete trace data that the authors ascribed at that time to many existing safety-critical systems.

Method and Class level Traceability. Recognising the importance of finer-granularity tracelinks, Rahimi et al. introduced an algorithmic automated approach in 2016 to maintain tracelinks at class and method level across multiple versions of software (Rahimi, Goss, and Cleland-Huang 2017). The authors

[15] Admittedly, the tracelink recover approaches based on machine learning or similar (to which the 'heavy-weight' attribute refers) have since advanced considerably, and the same author do refer to this fact in their latest works, such as in (Cleland-Huang et al. 2021).

[16] The approach still relies on recovery algorithms, but introduce the concept of total separation between the trusted and non-trusted links produced from those algorithms, which are then constantly updated through different processes that also involve the developer.

focus on the evolutionary aspect of maintaining such tracelinks with regard to addition, modification, and deletion. The approach relates to our work with regard to the finer granularity involved, however it assumes the pre-existence of such tracelinks and does not address the subject of capturing or facilitating them in a concrete process or a practical tool.

Understanding Practitioners Needs. To better understand the needs of practitioners when engaging in software change activities, Goodrum et al. conducted a study in 2017 involving 19 experts—including five from safety-critical systems backgrounds (Goodrum et al. 2017). The purpose was to collect relevant data to guide and inform the design of a prospective traceability tool. The study bears relevance to our research in terms of conducting expert interviews *before* the development of a solution so that the tool can be better aligned with users' practical needs—an approach that does not seem to be common among the software visualisation literature. The authors were also interested in understanding how the artefacts and their traceability links could be organised and presented in a visualisation that served the purpose of those users. Interestingly, the authors remark that, in practice, such artefacts and their links are neither readily accessible to developers, nor integrated into IDEs to support user tasks. This remark resonates strongly with our earlier expressed motivations, and an objective of this research is to work towards meeting this need. An interesting finding reported in this regard was the participants expressing that they usually faced particular challenges to understand "the context of the code change", and not being able to "see the big picture and understand interconnections among the code to be changed and other related software artifacts." Others emphasised the importance of establishing and maintaining dependencies among the various software artefacts in order to ease their code change tasks. Lastly, the authors also found that "Overview + Detail" visualisation approaches of the artefacts and their tracelinks gave good results. Indeed, these user needs, and the idea of creating tool support to aid developers in performing such tasks, come to constitute key areas that shape our tool design and development, and the design science behind it. The authors' paper did not cover tool design or implementation at this stage.

Agile-compatible Traceability. Driven by the motivation to facilitate greater uptake of agile practices in the development of safety-critical applications, Cleland-Huang and Vierhauser explain in their 2018 paper that there is indeed rising interest in the safety-critical domain to adopt agile development practices, but that this rise seems to be hindered by the fact that agile practices do not have a robust mechanism to enforce thorough traceability—a factor that is critical to these domains (Cleland-Huang and Vierhauser 2018). The authors explain that present traceability approaches seem to be not agile enough for the agile practices, and are often perceived as "burdensome overhead" by the agile practitioners. The authors proceed then to introduce a framework—named SafetyScrum—in

order to integrate agile methods with safety-critical systems. The approach involves a model where traceability is established from source code (class level) to 'Design Definitions', and then to 'Safety Stories', which are in turn traced to 'Hazards'. This same model was further developed and featured in their later work (Cleland-Huang et al. 2021), which was discussed in an earlier section. However, of particular interest to this work is the authors' remark that existing capabilities of agile tools such as Jira can be extended to create a "recommender system" to facilitate tool-supported tracelink creation. They further remark that version control systems such as GitHub "can be used to establish links between source code and design definitions and/or requirements, simply by tagging each commit with the ID of the design definition or requirement". The overall message was that recent advances "create viable options for automated tracing support within an agile project environment". This research closely embraces the above recommendations and works towards creating a practical and an operational mechanism, integrated in a popular IDE.

Integrating Agile and Versioning Tools. In 2019, Cleland-Huang's group reported on an implementation of their further developed traceability approach—now named Safety Artifact Forest Analysis (SAFA)—which is built around supporting safety-critical environments and their safety artefacts (Agrawal et al. 2019). The Java-based toolkit is described to interface with Jira and GitHub (with potential to integrate with other repositories) to retrieve the relevant artefacts and their links. The tool also supported a node-graph visualisation—an earlier version of that presented in their 2021 work, which was covered in an earlier section—to present the artefacts and their tracelinks to the user. The authors add that tracelinks to source code were made possible by "tagging GitHub commits with the respective Jira issue ID", but it was not made clear if this was done manually or facilitated by their plugins. In fact, it is understood from the paper context that the toolkit interfaces with Jira and GitHub primarily to retrieve and build the traceability data, and is not intended to facilitate or enable the creation/capturing of those tracelinks. Our research shares similarity with this work in the concept of tagging commit messages as well as with the idea of integrating with an agile dashboard and a version control system. In contrast, however, and in addition to visualising the artefacts and their traceability, our research is focused on enabling the real-time capturing of the tracelinks. It works towards providing a practical and operational tool that can be made accessible to the wider development community. In this regard, our tool also implements a different concept of approaching design and implementation artefacts, which is again particularly focused on making it applicable to the wider development community. Chapter 4 discusses this generalised artefact concept, while Chapter 6 discusses the tool design and implementation.

Explicit Traceability. Our work promotes explicit traceability, enabling developers—through an automated mechanism—to have design artefacts explicitly tagged with their code artefacts (and vice versa) at any desired level of granularity (e.g., file-level, class-level, namespace-level, function-level, method-level, and so on). While some researchers expressed that such an approach is difficult or infeasible as it is often done manually (Aljawabrah et al. 2021), we argue that, with proper tool support, it becomes practical and can be made low-footprint (i.e., affordable in terms of effort and intrusion). Moreover, recent researchers in the field have expressed calls and support for enabling such explicit traceability to be recorded at the early stage of development. Examples of this include (Gauerhof et al. 2022), (White, Krinke, and Tan 2020) and (Unterkalmsteiner 2020).

Closing. It is important to highlight that all of the works discussed in this section adopt an automated recovery approach to reconstructing tracelinks in a post-event manner. The only exception was that of (Delater and Paech 2013c) in which they partially captured traceability artefacts from developers directly. It is predominant in traceability research to build automated algorithms to rediscover lost tracelinks from a corpus of data. Our work (the part of it that deals with traceability) approaches the problem differently in that it works towards creating tool support and a mechanism that enables the capturing of traceability knowledge in real-time from the developers directly—hence sharing similarity with the Delater and Paech work. It does not incorporate any post-processing mechanism to retrieve and rebuild tracelinks from some existing corpus. Hence, we do not discuss *precision* and *recall* in our evaluation later, as it is not applicable to our case. In this regard, a *completeness* factor might be a more applicable evaluation indicator. However, this would only be meaningful on a project case where the tool was used by developers from day one. Still, it would not measure a characteristic of the tool itself, but rather, would measure its effectiveness in either encouraging (or enforcing) the capturing of the connections (tracelinks) from the developers. This discussion should become clearer when more details are covered in Chapter 6 and Chapter 8.

Yet another important point to highlight is that almost all works encountered in the traceability research field appear to treat traceability links as *concrete* and *actual artefacts*. This brings a significant disadvantage in the form of the overhead of having to maintain and keep up to date those trace link artefacts. In contrast, in this work we do not recognise or treat traceability links as actual artefacts. Instead, they are treated only as ephemeral phenomena that are computed in real-time and on-demand based on datapoints stored in the Design Artefacts themselves (covered in detail in Chapter 6). Hence, when knowledge of a tracelink is presented to the user, it is guaranteed to be always up to

date as it is fetched from its source in real-time, and no additional objects exist in the background that need to be maintained.

Lastly, it is also worth emphasising that our work did not originate from a desire to address the traceability issue. It is motivated in fact by a desire to help developers 'see' into their code visually, and how to put that into good use to support various development tasks. Synchronisation of design artefacts with implementation artefacts was necessary to realise that objective. To enable this, a mechanism was needed that encouraged a disciplined practice where developers see it as part of their work to identify whatever they contributed in terms of code and to relate it to the specific requirement it implements. The traceability advantage simply emerged as a result.

2.4 List of Notable Recent Software Visualisation Works

Table 10.1 in Appendix I presents some of the recent software visualisation works that are found interesting but were not covered in our literature review. It serves to highlight current research directions in the field, but also to give credit to some of the works that inspired and informed our research.

Chapter 3
RESEARCH METHODOLOGY & DESIGN

3.1 A Design Science-Based Study

Prelude. This work is largely motivated by the objective of bringing software visualisation technology into the world of the everyday developer. It has argued for the potential of this technology, but at the same time has acknowledged the current lack of its reachability to the end user. This lack of reaching the end user is caused by difficulties accessing the technology (see Chapter 5), but also by a considerable lack of availability of applied uses that take advantage of the technology potential (Leonel Merino, Ghafari, and Nierstrasz 2016; Leonel Merino et al. 2016)[17]. Moreover, making the technology accessible has intrinsic difficulties from a technical aspect, which is considered to be a significant factor that has hampered many earlier efforts (see Chapter 5). This research is designed to navigate these problems, and to contribute to the field through a robust construction process of a research artefact that address these issues. Described as such, the work has a strong research and development nature, and thus a Design Science methodology is adopted to guide the work.

However, a special characteristic of the work is that it embeds the end user and a study of their work environment into the research and development process from the outset and throughout its course. The overall goal is to collaborate with the end user to better understand their needs and their work environment. In needing to address the accessibility and availability issues, however, the work must first navigate and study a number of technical and technological aspects that have been hindering the delivery of those characteristics. Those technical and technological aspects represent the infrastructure that would facilitate attaining such goals—Chapter 5 reports on and addresses these in detail. In consideration of these properties of the research, and its intended goals, the work has specifically adopted the **Hevner et al. model** (A. R. Hevner et al. 2004) that sits under the design science research umbrella of methodologies, as it is found to best align with the research goals.

The remainder of this chapter elaborates further on this selection process, and then presents the design of the study.

Navigating the space of design science research methodologies. There are a number of recognised frameworks within which one can execute design science research. Some are extensions of others to add strengths to specific areas, while others differ in their philosophical drives and

[17] As reported earlier, recent issues of research publications specialised on the topic also show clear focus on 'applying' visualisation technology to software engineering, as well as interest in practicality issues such as integration into development environments, supporting development activities, and industrial experiences using the technology. See CFPs of VISSOFT 2022 (https://vissoft.info/2022/submission.html) and IST's 2022 Special Issue on the topic (https://www.journals.elsevier.com/information-and-software-technology/call-for-papers/visualization-applied-to-software-engineering).

dispositions. Some are found to place more attention on guiding the construction process (for example, Action Design Research), while others provide more guidelines around evaluating the artefact. With regard to philosophical drives, early methodologies seems to structure themselves around a recognised and well-established problem that the research then aims to systematically resolve (Nunamaker, Chen, and Purdin 1990; Burstein 2002). More recently, potentiality and opportunity began to be recognised as providing a valid research drive, empowering innovation in the discipline and allowing for 'proactive' research—rather than limiting efforts to remedial and reactive responses to existing problems (Iivari 2007; A. R. Hevner 2007). Frameworks were thus adapted to suit research endeavours where a recognised new opportunity plays a central role in the study (Cole et al. 2005; Iivari 2007; Sein et al. 2011). Overall, design science remains strongly positioned as a problem-solving effort—in line with other research paradigms—but is distinguished from the other paradigms in that it is centred around the construction of innovative artefacts (Cronholm, Göbel, and Hjalmarsson 2016). Whether a design science effort was driven to solve an existing problem, or by a recognised opportunity, both share the construction of an artefact as a key activity, and both eventually aim to provide better and improved service to the end user. Both also strive to contribute to the discipline, not just through the introduction of an innovative artefact, but as importantly, through the knowledge generated around the construction process itself. In summary, given these variations in how the research is conducted, and how it is philosophically justified, it is important to choose a research framework that best aligns with those elements of the research itself.

To illustrate further, the Nunamaker et al. framework (see Figure 3.1) is found to work well and suffice where the research objective is to deliver a prototype or a proof-of-concept artefact. It offers a simple and straightforward—yet systematic—way to approach the research. Indeed, this model was adopted in our earlier work (Alshakhouri 2013) and was instrumental in guiding and successfully delivering the research. However, the model falls short in supporting this specific research, as it offers little guidance around the inclusion of the end user or their environment in the research process.

In this work collaboration with practitioners has a key role throughout the process, as it is believed to increase the likelihood that the artefact will indeed address the real-world needs of the user, and will address them in a way that actually suits the user and their way of working—hence leading to a better chance of adoption. Looking at Nunamaker's et al. model shown in the figure, one can see that *Systems Development* is the central activity of their research model, with the three other components positioned to support, facilitate, and inform that central activity. Other important elements, such as the application domain and the environment context, are not represented anywhere in their model. While it can be argued that the application context is an implicit part of the Observation activity—

facilitating the evaluation and validation efforts—it appears that indeed this is the extent of its role. That is, the application context does not play a role in the research and development activity itself, but rather is confined to test and validate the resulting artefact. The model makes no attempt to situate the application domain and environment context as an active component that informs and shapes the research. Closer examination of the model further confirms this position. In summary, this framework is primarily centred around the artefact itself (or the instantiation), and how to rigorously approach its building process, to ensure the research and development effort is well-informed by existing literature, and to help in that effort leading to an academic contribution to the discipline. It does employ a feedback loop to evaluate and inform the work, but it does not explicitly address the end user, their actual needs, or their environment in the process itself.

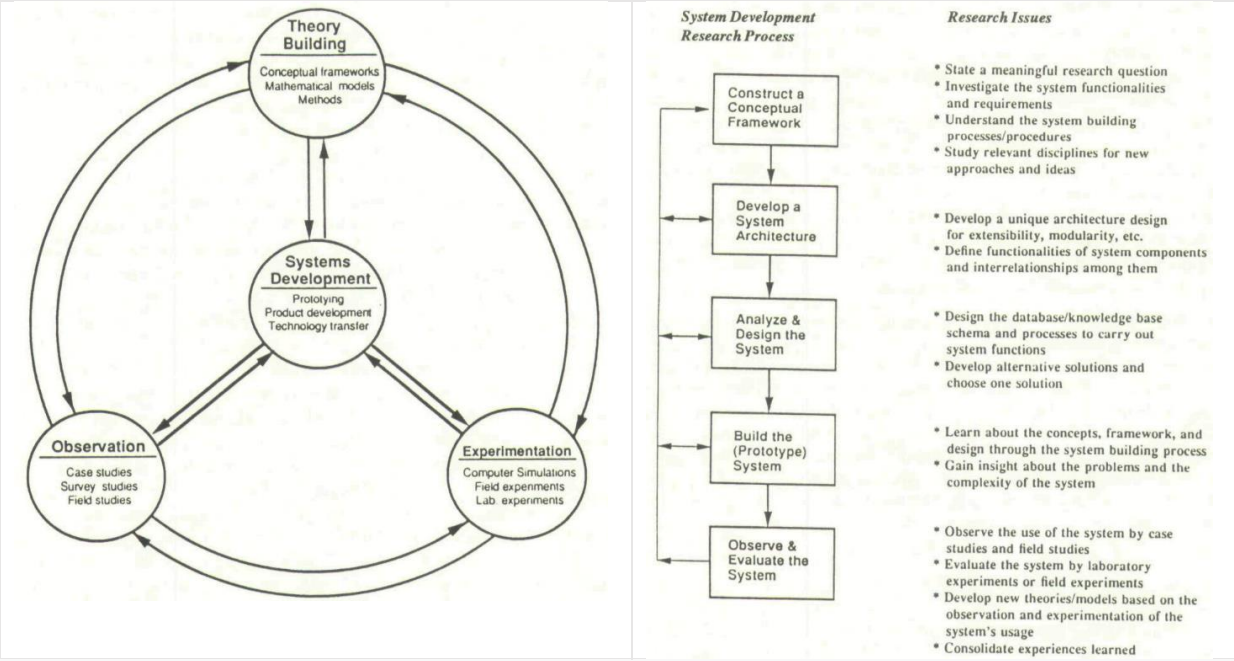


Figure 3.1: To the left, Nunamaker's et al. multi-methodological framework of Information Systems Research, and to the right, appears their execution process model where environment role is apparently limited to testing and validating the research artefact. Both figures are adapted from (Nunamaker, Chen, and Purdin 1990).

Frada's (Burstein 2002) article presented an updated and generalised version of the original Nunamaker et al. framework (Burstein 2002). The new model is seen as more inclusive—as well as empowering—in that it places the *Conceptualisation process* as the central activity that is then continuously improved via feedback loops from the other activities (see Figure 3.2). Here, construction of a research tool or prototype is placed as a separate activity, that in itself helps feed and improve the conceptualisation activity. In this model, the research artefact is not confined to the resulting prototype tool. Instead, both the prototype and the new introduced concept are given valid grounds as research artefacts, with the concept receiving the central focus. This adaptation would suit this research better than the original Nunamaker et al. model. This is because, in this work, significant effort is attended to building and

refining a novel concept, and construction of the tool is seen as the practical implementation of that theoretical concept. It works as a validation mechanism of the concept, but at the same time, as a mechanism to improve and refine the concept (see Figure 4.4 and Figure 4.5). However, in a similar fashion, this model still lacks sufficient representation and incorporation of the overall user environment in the design process itself.

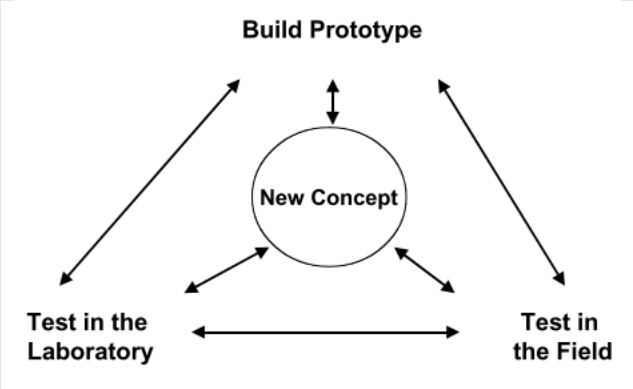


Figure 3.2: The generalised CMI Arizona Research Model as appearing in (Burstein 2002).

Efforts to find a framework that is more closely aligned with the philosophy and objectives of this research have eventually led to the Hevner et al. model, which was originally introduced in 2004, and refined over a series of later works (A. R. Hevner et al. 2004). In contrast to the two models discussed earlier, this research framework *explicitly articulates* the role of the application, the environment, and the organisational context in shaping and steering the research—a characteristic that strongly aligns with and serves the goals of this work.

In the Hevner et al. model, the environment is not seen as merely a testing ground to validate and refine the new concept (being embodied in its instantiation), but is rather placed to actively influence, inform, and shape the research via informative feedback loops, and constructive interaction and collaboration with its users. This thinking is strongly promoted via what they refer to as the *Relevance Cycle*.

The next section presents this model in more detail, in order to better explain how it supports the research objectives.

3.2 Hevner et al. Model—Our Adopted Research Framework

Figure 3.3 presents the overall framework of Hevner et al., where three predominant components can be readily identified. The **Environment** is set to represent the intended or targeted context, as well as the problems and/or opportunities that the research aims to address. The **IS Research** encapsulates the actual research activity. The **Knowledge Base** captures the entire corpus of past research, grounding theories, experiences, and expertise as identified by the researcher to support and inform their work, as well as the knowledge contribution that is generated and fed back into the discipline's knowledge base. The diagram also shows the interactions and feedback loops between these three components. Three cycles of research activity can be further identified; a **Cycle of Relevance**, a **Cycle of Design and Evaluation**, and a **Cycle of Rigor**—these are illustrated more effectively in Figure 3.4. Considered collectively, the three cycles come to capture the key activities that should shape any rigorous instance of Design Science Research (DSR).

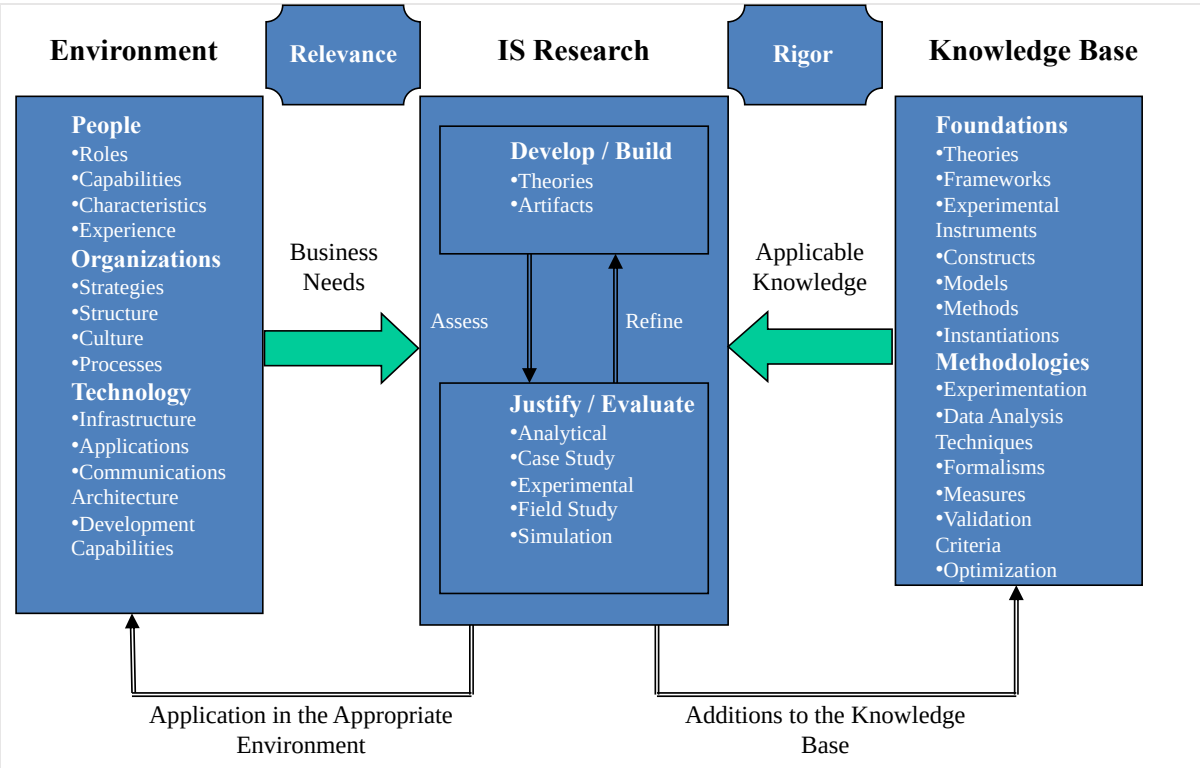


Figure 3.3: Hevner et al. Information Systems Research Framework as first appearing in their 2004 seminal paper (A. R. Hevner et al. 2004). Diagram is adapted from Hevner's 2015 Presentation (Alan. R Hevner 2015).

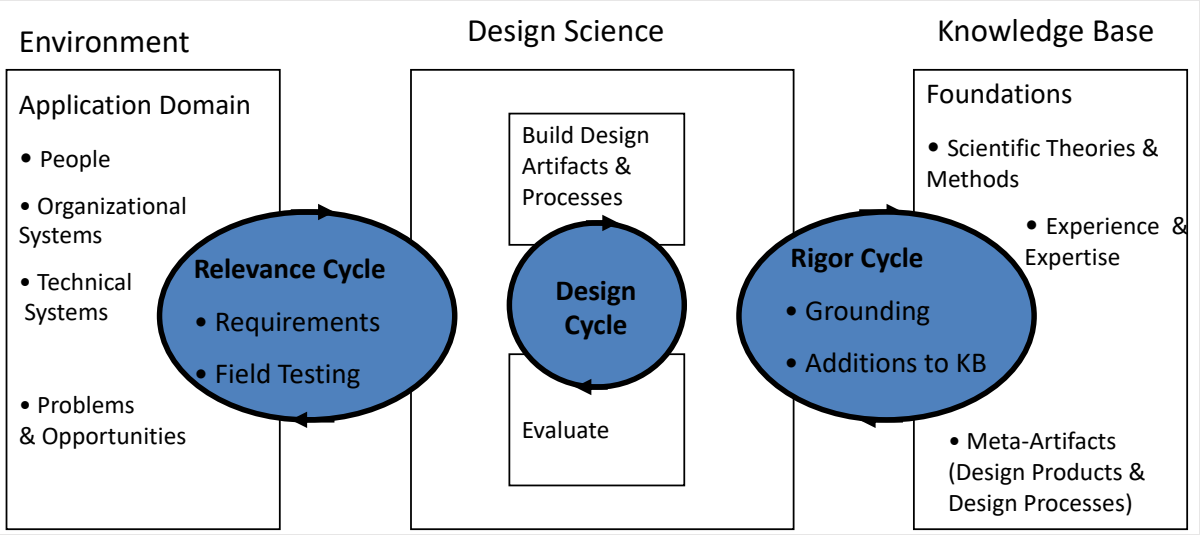


Figure 3.4: Hevner's Three Cycles of Design Science Research as first appearing in his 2007 paper (A. R. Hevner 2007). Diagram is adapted from Hevner's 2015 Presentation (Alan. R Hevner 2015).

The Relevance Cycle captures the ***research context*** in terms of the targeted environment. It includes the *people*, the *organisational factors*, and the existing *technology constraints* as key influencing players that shape the environment. They should work to continuously inform and guide the design research. However, they also present a context or sound basis for conducting rigorous empirical evaluation of the resulting design artefact in a real-world setting. Hevner and Chatterjee further recommend that such activity should include the prior definition of the perceived improvement that the research promises, and how this improvement and its effectiveness are going to be measured (A. Hevner and Chatterjee 2010).

Informing the research can start with the identification of legitimate research problems, or the identification of research opportunities and potentiality. The informing process should not be limited to this initial formulation of the work context, however. In Hevner's model, the environment plays an active and continuous role to constantly steer the research to ensure its output better serves and suits the environment, including its users. This could mean regular evaluation or verification activities to test the artefact in various stages of its development. The framework also advocates collecting information and studying the environment before the artefact construction even begins, and before the research concept is fully developed. For instance, some researchers recommend conducting user surveys and expert interviews in the early stages of a research endeavour in order to better understand the environment and the real-world needs of the user (L. Merino et al. 2018). This is likely to equip the researcher with more relevant knowledge during the conceptualisation stage, and increase the chance of the artefact coming to be more meaningful to the user, and suitable to their actual environment. In summary, the interaction process between the actual application environment, and the research's design and development activity, is a crucial aspect of this model in steering and conducting the

research. However, depending on a researcher's interests, skills, and goals, the level of utilisation and incorporation of the environment can vary significantly. In fact, Hevner even suggests that elements of Action Research could be integrated to conduct this particular cycle, if the researcher seeks extended involvement of the environment and its users (or stakeholders) into the research (A. R. Hevner 2007).

The Design and Evaluation Cycle captures the core research and development activity, including any laboratory experimentation and validation of the design artefact. The Relevance and Rigor Cycles work to keep the design activity continuously informed by input from the environment and its players, and from the body of literature. However, results and outcomes from the design activity are also expected to be fed back into both other components—e.g., the product artefact into the environment, and contributed experiences, findings, concepts, and artefacts back into literature.

The Rigor Cycle is where design science research gains its position as an academic effort, building on extensive examination of the discipline's knowledge base, foundations, theories, and methodologies—and then contributing back new knowledge generated by the overall research activities. In fact, aside from the generated contributions to the body of literature, what truly distinguishes academic design research from a routine design and the industrial practice of systems design, is the *amalgamation* of the Rigor Cycle and the Relevance Cycle, and how these two are put to effective use by the researcher to execute their design and development activity—as remarked by Hevner and Chatterjee (A. Hevner and Chatterjee 2010). Iivari's 2007 article titled '*A Paradigmatic Analysis of Information Systems As a Design Science*', Iivari shared a similar emphasis noting that it is "the rigor of constructing IT artifacts that distinguishes Information Systems as design science from the practice of building IT artifacts" (Iivari 2007). According to Hevner et al., this rigor is acquired through the effective use of prior research—i.e., the domain's body of knowledge (A. R. Hevner et al. 2004). This also includes the selection of the methods, theories, and techniques that are employed by the researcher to design, construct, and evaluate the research artefact.

Another remark by Iivari that is of interest here, is his acknowledgement that the legitimate drive and motivation of design science research should not be restricted to identified and outstanding domain problems, but design science research could also be legitimately motivated by the recognition of a new opportunity that may improve the practice before any problem actually arises, or is 'officially' recognised. This is found to be in close alignment with our original research motivation. To clarify, our research identifies and is motivated by two well-known problems—the lack of adoption (and accessibility) of software visualisation by practitioners, and the long-outstanding problem of artefact traceability. However, our research is also driven by an identified new opportunity—the potentiality of

employing visualisation technology to improve and support a well-defined application environment—that is, the traceability of original design features and development activity in agile practice environments. The reason we acknowledge this work as a perusal of 'new potentiality' more than a targeted effort to solve a particular problem, is because while artefact traceability is a well-known and legitimate issue in software development practice, it is certainly less apparent and less reported by agile practitioners due to the nature of this paradigm that places lesser importance on maintaining original requirements (e.g., user stories), let alone be interested in their traceability. This point has been well-discussed in previous chapters, including indications of a shift in interest among the agile community. In fact, coming to address the artefact traceability application using software visualisation technology could be seen as a testament to the legitimacy of undertaking research based on well-established 'potential', as it is indeed a by-product to the original claim that the technology holds potential to support practitioners in a number of contexts. Artefact traceability came to be recognised as one of those contexts, and research was afterwards carried out to further develop the concept. However, the technology seems to also hold potential (and interest by practitioners) for other contexts (or applications of use) as our evaluation activities have revealed (see Chapter 8). Needless to say, identification of a new potential to form a solid and legitimate research endeavour requires an elaborate study and analysis of both the existing body of literature (Rigor Cycle), and the actual application environment (Relevance Cycle), and should be recognised to stand on par with the activity of establishing an outstanding domain problem.

Lastly, it is worthy to note that the Hevner et al. framework provides an accompanying toolkit that presents a good range of support to the overall research process. This includes guidelines on how to properly execute the research, advice for conducting rigorous empirical evaluation, and lastly, guidelines on effective presentation of the research aimed at highlighting its *contribution component* to the academic community and its *utility and efficacy* to the practicing community. These guidelines present rich and valuable support towards rigorous execution and delivery of the research.

Closing. This research identifies the potential of software visualisation technology to support the practicing community in a number of applications. However, the technology is hardly accessible or available to the practicing community, and appears to be exclusively confined to academic domains. Moreover, in order to demonstrate its potential, the technology must be applied and put to use to support targeted application contexts, so the user could begin to reap its benefits. There is, however, considerable difficulty in bringing the technology closer to the everyday developer, which is manifested mostly in technical and technological intricacies. To work toward addressing these issues, this research embarks on studying prior literature in the field, on conducting extensive study and examination of

relevant technologies, and on regular collaboration with the end user. The research framework of Hevner et al. with its three cycles of rigor, design and relevance is found to offer strong support and alignment to execute this research.

The next section introduces the design employed by this research, which is based on the Hevner et al. framework.

3.3 Design of this Research

This section presents the overall design of this research and offers some discussion around its execution. A diagram capturing the overall process is also provided. However, a detailed description of the evaluation design is left for Chapter 8, which is dedicated to addressing that particular aspect.

3.3.1 Three Phases of Design Science

This work implements three phases (iterations) of Hevner's research cycles—that is, it conducts the design and development activity over three sequential phases, where in each phase it seeks input and insight from the environment and the body of literature, to refine the research concept and guide the development of the design artefact. The first iteration was completed at a very early stage of the research, where input from industry practitioners was sought and helped to verify and establish the potential utility of the concept. Prior to that, literature was examined and reviewed to establish the status quo of the discipline, and confirm the research motivation and objectives. This was facilitated by the outcome of our earlier work, and a journal paper was published that outlined the position of the research and its future goals (Alshakhouri, Buchan, and MacDonell 2018).

The research concept was then further refined by a second collaboration with experts from industry, through the use of a wireframe prototype and structured interviews. That phase produced valuable insights, recommendations, and directions that were instrumental in undertaking the third and final phase of the research, where a functioning tool was constructed that implemented the refined concept. In a similar fashion to the prior two phases, an evaluation was conducted in collaboration with experts from the industry to verify the utility, usability and benefits of the constructed artefact, and with this activity the research was brought to a conclusion.

The reader is referred to Chapter 8, which presents a detailed discussion and a diagrammatic representation of the three phases (also reproduced at the end of this chapter for reader's convenience).

3.3.2 Addressing Domain and Environment Relevance, and Research Rigor

A significant differentiating aspect of this research is that it is based around understanding the existing user environment, the tools they use, their way of working, and the potential areas where this research could offer some help. Another equally important aspect is the examining and studying of the technological infrastructure that is necessary to construct the artefact, in a way that ensures its accessibility and availability to the everyday developer.

Relevance in the first aspect was largely addressed through direct collaborations with expert users and analysing the generated feedback and recommendations. Nonetheless, literature also played a considerable role in understanding the existing practice and needs, especially around the artefact traceability domain. As for the second aspect, research rigor was addressed through studying the existing literature with regard to the technologies used, but also by examining the technologies, libraries, and tools that are presently available and used by the software industry and community. This is to ensure relevance to the targeted environment, while at the same time maintaining solidity and support through the discipline's body of knowledge.

Chapter 5 covers the latter aspect in detail, while the first aspect (relevance with the environment) is addressed in Chapter 8.

3.3.3 Research Path and Design

This research has been inspired by the gradual emergence of a new concept based on careful study of the body of literature—specifically, that dealing with the application of software visualisation technology to agile practice. Recognising the potential of software visualisation, a range of landscapes within software engineering practice were explored where this technology could potentially be put into effective use. This led to the adoption of agile practice as a high-potential candidate to benefit from this technology. Integrating design artefacts (requirements in generic form) with the resulting software artefacts (code in generic form) came then as a more specific focus. Eventually, the research was set to focus on the problem space of artefact traceability within the agile software development domain, as a targeted application area to benefit from software visualisation. Reflecting on the adopted research model, the above activity could be seen to correlate to the Rigor Cycle where the conceptualisation of the research is established and informed by the discipline's knowledge base. While aspects of the application environment have necessarily influenced this stage, it was primarily based on theoretical examination of practice as obtained from the literature. The Relevance Cycle was then utilised to verify, refine, and improve the concept, as detailed in Chapter 8.

Examining Hevner's model as illustrated in earlier figures shows that the Design Cycle is itself composed of an inner iteration of artefact design, build, and evaluation activities—hence the reference to it in this work as a 'Design and Evaluation Cycle'. This cycle allows for the design artefact to be iteratively built, assessed, and refined, until the desired result is achieved. Hevner and Iivari recommend the conduct of laboratory evaluations of the design artefact before releasing it to the Relevance Cycle (A. R. Hevner 2007; Iivari 2007). Such laboratory evaluations may be conducted in a number of forms—e.g., a simulation, a case study, an experiment. This work employs a combination of laboratory evaluation

exercises (mini-case studies using open-source systems) but, most importantly, extends to the environment through collaboration with experts in each of the three evaluation exercises. This way, while falling short of field testing, it ensures connection with the end user and their environment, and to capture their valuable feedback and input.

The diagram below summarises and presents a visual depiction of our design study in its overall form. Chapter 8 presents a more detailed diagram focusing on the evaluation activities.

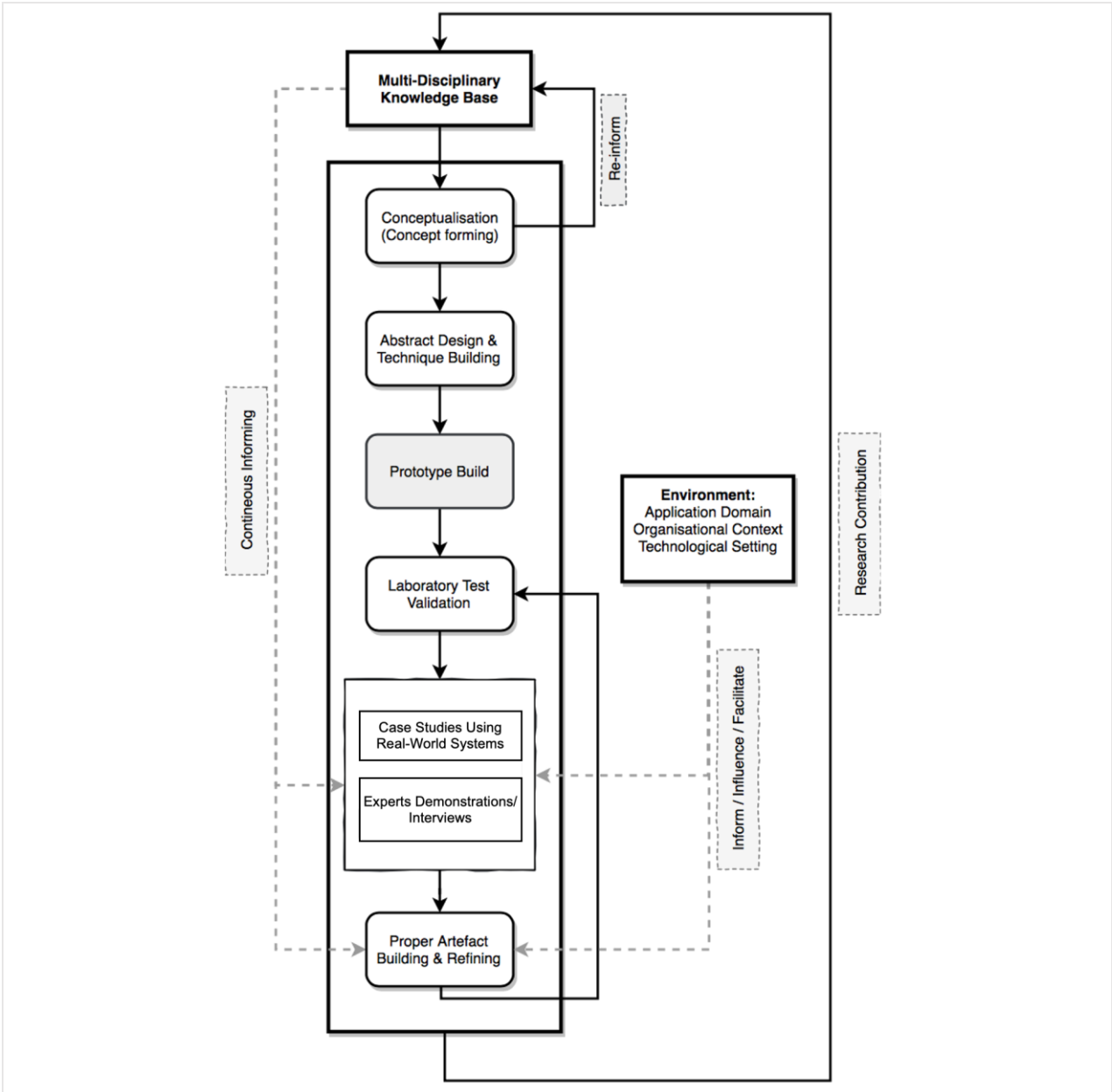


Figure 3.5: Diagram illustrating the overall design and methodology employed by this research, which is an adaptation of the Hevner et al. Design Research Framework.

3.3.4 Literature Guiding the Research Design and Evaluation

As reported in earlier chapters, the topic of empirical evaluation is often highlighted as a significantly lacking aspect in the software visualisation body of literature. This matter was discussed in detail in our earlier work, and an examination of the recent literature shows that the issue still largely holds, and calls for empirical studies remain outstanding. This research aims to extend attention to this issue while actively mitigating the present situation. Although conducting a controlled experiment was infeasible given the scope of the research, the decision to work closely with expert users over extended iterations of collaborative sessions is believed to bring new strengths to the field. As elaborated in Chapter 8, involving expert users in evaluation exercises in the software visualisation discipline is very rare—the majority of works have relied on independent laboratory simulations or case studies (L. Merino et al. 2018).

In consideration of the above matter, a number of prior and recent studies have guided this work, particularly around the design and evaluation process, providing invaluable insights and lessons. Following is a brief introduction to these works, covered later in more detail under the relevant sections of Chapter 8.

The software visualisation community has produced a number of works aimed at providing guidance and advice on conducting rigorous empirical work to evaluate visualisation artefacts. One example is the Müller et al. paper titled "A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization", where they introduce and describe an approach to conduct empirical work over a series of experiments to rule out influencing factors (Müller et al. 2014). A year later the authors reported on an implementation of their approach in a paper titled "How to master challenges in experimental evaluation of 2D versus 3D software visualizations" (Muller et al. 2015). Despite their focus on controlled experiments, the works offer good insight into the wider aspects of evaluating software visualisation tools.

Other studies in this regard that were of particular use to this work include the mapping study on validating software visualisation tools by Seriai and his colleagues (Seriai et al. 2014), and the more recent systematic literature review on the same topic by Merino and co-authors (L. Merino et al. 2018). Aside from providing an excellent status quo on the matter in the discipline, both works offered particular insights and lessons around the design of user tasks, which is again covered in due course in Chapter 8. In addition to those mainly descriptive studies, a number of exemplar works involving controlled experiments were also examined. Instances of this include the work of Baum et al. (Baum et al. 2017) and the earlier work of Fittkau and his colleagues titled "Hierarchical software landscape

visualization for system comprehension: A controlled experiment" (Fittkau, Krause, and Hasselbring 2015).

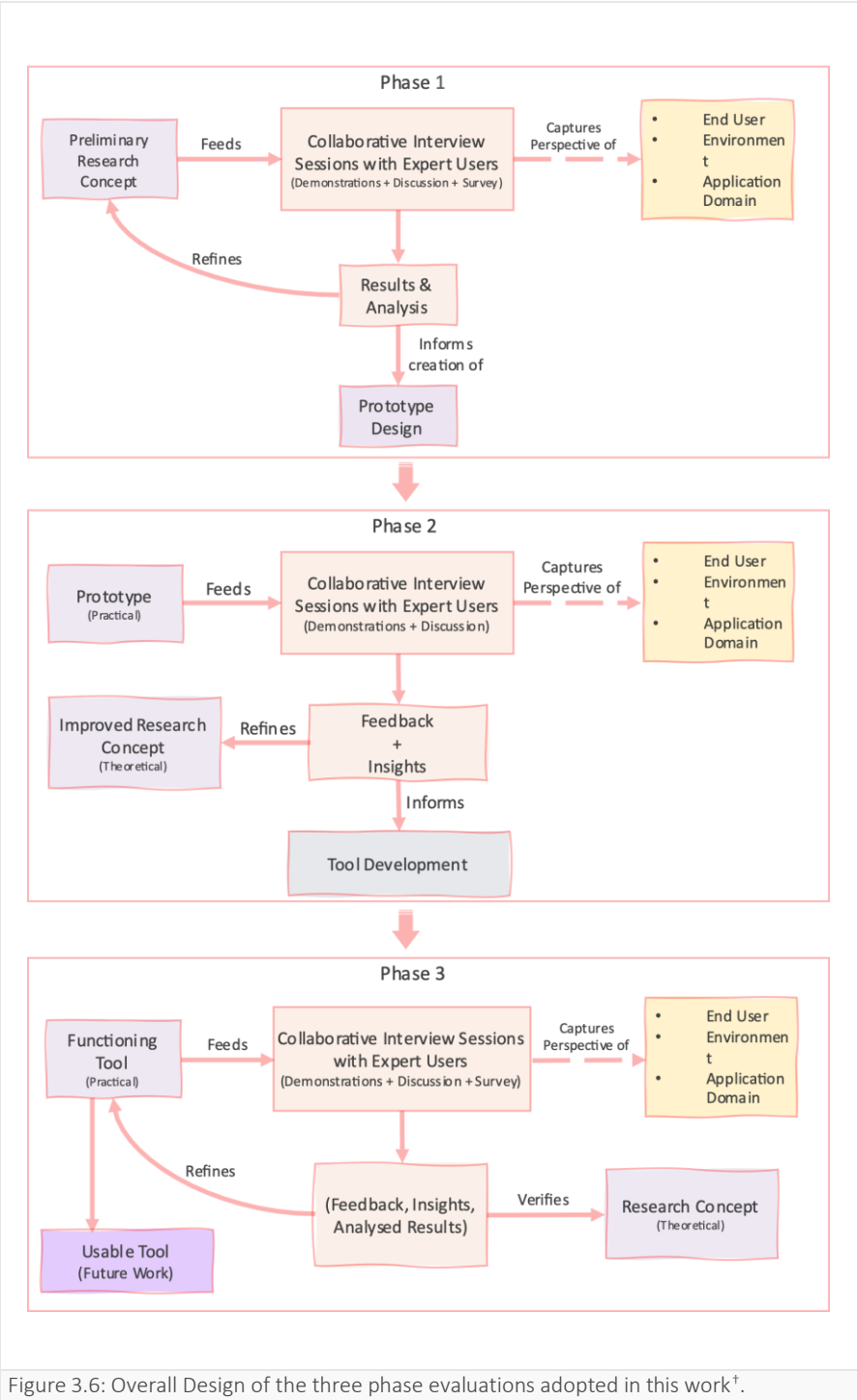


Figure 3.6: Overall Design of the three phase evaluations adopted in this work[†].

[†]Reproduced here for reader's convenience

Chapter 4
CONCEPT DEVELOPMENT

4.1 Prelude

This chapter introduces the concepts that underly this work, and which enabled the construction of AgileInsight—the research tool, covered in detail in Chapter 6. The work as presented in its current shape across the coming chapters is the culmination of early raw concepts that were developed and refined progressively over the course of this research—some had origins and roots in our earlier work. In fact, this chapter is intended to represent the theoretic side of the work—where the concepts came to be formulated, designed, and refined through a combination of literature examination and collaborative work with industry experts. The developed tool, AgileInsight, represents the practical implementation of those concepts that are introduced here in theory.

Throughout the development of this work, there was a keen drive to arrive at a generic concept that captured the key ideas and motivations of the research, rather than to base the work on a specific implementation. We believe this approach gave fruit as it allowed us to develop the concept independently from any implementation, and the actual tool development was indeed only carried out as the last stage of work. More importantly, the concept is indeed independent from any development methodology or a specific programming language—as discussed in the coming chapters.

The purpose of this chapter is twofold. First, it is to offer the necessary transparency behind the key elements that underpin this work and that made it possible. It is also meant to share the thinking and the research foundations that led to the emergence of the final concept—as we believe this could hold value to future researchers. More importantly, the chapter also serves to draw parallels with earlier relevant works, and to build support from existing literature. The second purpose is that, by presenting it here, the concept could be utilised by future researchers or inspire future work that may address completely different application contexts, but remain independent from a specific development methodology or programming language.

Lastly, it must be acknowledged that the research concept as presented has no complexity at all, and is rather surprisingly simple. However, we believe its simplicity could be a source of promising potential. In one aspect, it enabled us to abstract design and implementation in a highly generic form, and consequently, allowed us then to develop the concept—and later to build the tool—in complete independence from development methodologies or programming languages. In another aspect, the concept is believed to capture and enable a number of related ideas that have been raised independently by earlier researchers and are found in varying contexts, ranging from traceability, conceptual design, to visualisation works. It could thus be seen as an amalgamation of ideas that have echoed in one form or another in earlier research.

4.2 Concepts as Building Blocks that Enable Software Engineering

While this work is certainly not intended to address the subject of conceptual design in great detail, it nonetheless has been driven in some respects by literature advocating the topic of design concepts in software engineering. This section discusses a selection of the most relevant works.

Daniel Jackson has produced a series of works motivating the adoption of 'Concepts' in software engineering as a better approach to guide the development process. In 2013, he introduced his work in a research agenda paper, that treated the subject from a theoretical perspective. He described concepts as "constructs and notions" that already exist in the world (e.g., bank account) or invented specifically by the software system to better structure its functionalities (e.g., the concept of a 'layer' in graphic design applications). He argued that 'conceptual design' is about designing the behaviour of the collection of concepts that make a particular software instance, and that approaching the design of a system from the perspective of its concepts leads to greater ease of use of the product as well as better quality of the code (Jackson 2013).

In his 2015 paper titled "Towards a Theory of Conceptual Design for Software", the work was developed further, providing a practical and systematic treatment of the topic using examples from software design (Jackson 2015). His key message, and one which resonates with the original motivations behind this work, was that encouraging developers to think of the core 'concepts' underlying their system has promising potential to lead to better software design. The paper elaborates that concepts drive the design of software, and that in a good design, a concept should be motivated by one purpose, and that each purpose should in turn fulfil a single concept—promoting coherence and discouraging coupling. He illustrates this further with examples from practice, arguing that concepts provide developers with a constant reminder of end goals, discouraging them from careening off to other paths, and that the level of grasp of the core problem at hand is what matters most, more than a developer's skill at programming. Interestingly, he postulated that concepts could offer agile methodologies particular help by presenting a framework to ensure the integrity and coherence of features—as the incremental approach to delivering functionality, while proving invaluable at tackling complexity and minimising risk, could often lead to fragmentation of features. Another interesting remark that also resonates with this work is his description of *'concepts dependency graphs'* as offering a good map for exploring a system's design space with a 'reduced functionality'. This is to some extent in line with our objective of providing a medium to visually explore the design concepts of a system, as mapped across and as synchronised with their implementation.

In 2016, along with Santiago De Rosso, they applied their theory of conceptual design to the Git version control system, claiming that a major source of its perceived complexity is an underlying bad design of concepts. They used their framework to remodel Git at a conceptual level, and built a reworked version named 'Gitless' to counteract what they described as 'flaws' in the current Git system. An evaluation of their work appeared to show potential in terms of better usability of the end product, as well as improved ability to identify and fix design problems (De Rosso and Jackson 2016).

Gulden and Reijers adopt an aligned stance with regard to the important role of conceptual design models in delivering a better software engineering process, arguing specifically from the perspective of comprehension and communication of how a system is working (or intended to work), as well as comprehension of its requirement specification (Gulden and Reijers 2015). They argue that such models are useful to validate one's understanding of a system, but are also useful in guiding developers to build the system. However, rather than attending to the design of conceptual models, the authors focus on the visual presentation of those models, emphasising the role they play in stimulating the human cognition process. In fact, their position paper calls for the embracing of visualisation as a 'primary concern' to aid the software process, rather than treating it as a peripheral one—reiterating earlier researchers' call to harness theories from the disciplines of graphic design, cognitive science, and knowledge construction, to formulate a solid basis for designing and using effective visualisation. For example, they describe that, according to established knowledge construction theories, patterns and metaphorical abstractions are natural in terms of how the brain constructs knowledge and are essential for higher-level cognitive abilities to start to emerge. These lines of thought share useful common ground with the motivations behind our work, with regard to the notion of harnessing humans' natural visual cognitive abilities to communicate complex information and relations using simpler (and naturally familiar) patterns and metaphors. In particular, it echoes our interest in capturing the design concepts and presenting them visually to the user. Interestingly, while Jackson emphasised the importance of those concepts for a better software process, Gulden and Reijers called for the need of their 'effective visualisation'.

This importance of conceptual models in guiding developers is also supported by Ben-Ari and Yeshno, who in 2006 conducted an experiment in an education environment, in which they claimed that well-designed conceptual models enabled their participants to analyse and solve problems conceptually, whereas their control group engaged in aimless trial and error paths (Ben-Ari and Yeshno 2006). Their work is motivated by ideas from the constructivism theory of knowledge, which argues that knowledge emerges internally in humans through mental models constructed by each individual. Hence, in the absence of readily available and well-designed conceptual models, developers would eventually and

naturally resort to coming up with their own mental models, which in the case of students and new developers, face the risk of being flawed. They thus call for the embracing of conceptual models in these environments. This again shares similarity with our motivations, in that we have argued that developers working on a codebase would resort to constructing their own internal models of the codebase structure to be able to navigate through their problems and tasks. The same is also true with respect to constructing internal models of the design concepts of the application itself. For example, a developer could end up with a different perception of how a feature is supposed to work, or how it is interrelated to other features, from other colleagues. This idea was then used to motivate our drive for bringing the structure of a codebase out to the surface, and physically visualised, so it becomes a medium through which teams can begin to construct a common and shared mental model that is more accurate and always up to date. Unifying this visualisation with design concepts was then argued to bring to surface a visual representation of those concepts as realised and as distributed across the codebase—hence, the 'synchronised' descriptor. Such a view potentially enables a unified lens on 'design concepts and implementation' to start to emerge as shared internal models among developers.

Interest in bringing to the surface and visualising design concepts is also evident in the work of Garm Lucassen et al., who developed a tool called Visual Narrator aimed at extracting design concepts from user stories, and then visualising them using node graph diagrams (Lucassen et al. 2017; Lucassen, Dalpiaz, van der Werf, et al. 2016). This work was discussed in Chapter 2 but is brought up here to draw a connection to researchers' interest in capturing design concepts at varying levels of granularity for the purpose of supporting the varied tasks of the everyday developer.

Closing Remark. While this section is not intended to present an elaborate examination of design concepts or conceptual modelling, we contend that the above discussion illustrates sufficient evidence of software engineering researchers' interest in the topic, and for its perceived importance to the software process. In light of the above discussion, it is worth reiterating that an objective of this work is to enable the capturing of design concepts, and to present them as visually synchronised to their implementation. We particularly see concepts as sitting at the high level of a system's design, and that in practice, concepts are always broken down into smaller design intents represented by varying forms of requirements such as epic features, individual features, tasks, and the like. Those lower-level concepts are then transformed by developers into actual working code. Thus, the abstraction that follows in the next section is purposely intended to capture the design concepts at any level of the hierarchy, whether it is a top-level concept, or a lower-level user story or feature. It must also be noted that we acknowledge that not all methodologies practice conceptual design as part of their engineering process, particularly in the agile development community, where the focus tends to be placed on the

lower-level concepts manifested by user stories or features. Nonetheless, design concepts, whether practiced or not, will always be internally constructed by developers to guide their work. While the implementation in this work incorporates lower-level design items, such as features and user stories, to demonstrate the concept, it nonetheless is designed to capture and represent design concepts at any level of the design hierarchy and regardless of the development methodology practiced. The diagrams in (Figure 4.4 and Figure 4.5), and the upcoming discussion explain this further.

4.3 Generalisation of Design Concepts and Implementations

A key antecedent to realising the objective of this work is to be able to work with design concepts and their implementation in a generic manner, without being tied to a particular methodology or programming language. In other words, to be able to access and work with the building blocks of both aspects of the development process—the requirements and the source code—in a generic form.

In light of the above discussion of some of the literature on conceptual design, we sought to either utilise an existing model—if such a model exists—or to create one, for each of the two aspects. With respect to capturing the requirements in abstract form, the closest match we were able to find in literature was the Work Breakdown Structure (WBS) model. However, the model does not meet the level of abstraction required, and has rather more complexity than needed. Moreover, it is relatively old and structured primarily around the traditional development methodologies. In fact, similar effort to find a modern and abstract model to capture and represent requirements was also attempted in our earlier work, but failed to uncover any good results (Alshakhouri 2013). Also, no such model was identified or reported in any of the recently reviewed papers that shared some common grounds with our work, whether those studies were focused at the conceptual level such as (Lucassen et al. 2017; Jackson 2015; 2013; Vidya Sagar and Abirami 2014; Ben-Ari and Yeshno 2006) or those studies that were undertaking practical work such as (Agrawal et al. 2019; Cleland-Huang and Vierhauser 2018; Delater and Paech 2013b). Rather, those that featured practical work have always resorted to creating their own model, which was specifically suited to their purpose.

As for the implementation side, our preliminary examination of the literature was unsuccessful to reveal any model that readily met our goal of capturing the structural elements of source code in an abstract and generic form.

Considering the simplicity and basic requirements needed for each model, it was decided that it would be more efficient to create our own abstract model for each of the implementation and design concepts, rather than attempting further literature examination to find if such models existed.

4.3.1 Abstracting Design Concepts

Design concepts come in varying forms and sizes. For example, in current agile methodologies, design concepts could be captured by epic user stories or epic features, that are then broken down into individual features, user stories, and tasks. However, smaller features, issues, and user stories could also be used on their own—that is, the hierarchy or grouping is not necessary. Those design concepts are also intrinsically characterised by being either constructive or corrective in nature. Constructive

design concepts are those that are primarily intended to introduce new features and functionalities. Corrective design concepts are those that are primarily intended to fix the behaviour of existing functionality. This classification is in line with the well-established distinction between development and maintenance that held prior to the emergence of agile methods. The distinction is helpful but is not vital for the introduced concept. Also, *enhancements* or *improvements* can sit in either category according to the perceived extent of their contribution.

In order to make it possible to work with design concepts irrespective of the paradigm or development methodology, the concept had to be generalised and abstracted to a very basic skeleton. The following is how we defined a design concept:

Design Concept: a mental construct embodying a form of software requirement or specification that at a given point in time might be actioned by developers and turned into a working piece of source code.

Explanation and examples: the above definition is intended to capture functional requirements in their most generic form, and regardless of their level in the chain of implementation. Thus, the above definition could apply to high-level requirements such as formally defined specifications in the case of traditional approaches, and to epic features or user stories in the modern agile methodologies. But most importantly, it also includes lower-level requirements such as broken-down user stories, features, tasks, and specific design requirements. It also captures both forms of design requirements, the constructive and the corrective. Thus, a user story, a feature, or a task are considered to be design concepts, but so are bugs and issue fixes.

Object Model: a design concept is represented by a unique identifier and a payload representing the description of the design concept itself. Figure 4.1 illustrates this simple model. In actual implementations, the payload could consist of additional properties to suit the particular context—e.g., a title could be a helpful addition—however, the unique identifier is the only crucial element that must exist to enable the abstraction.

To account for the hierarchy in design concepts, an optional reference to parent—and children—could be incorporated into the skeleton, which would allow the model to capture such cases whenever applicable.

> Remark. In our practical work, design concepts are referred to as Design Items (DIs), as this terminology is seen to be friendlier and more familiar to practitioners and everyday developers.

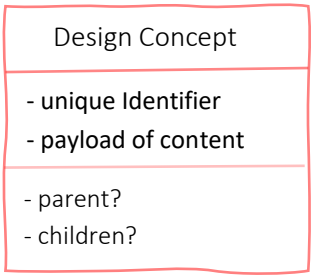


Figure 4.1: Abstract object model to capture and represent design concepts (requirements) irrespective of development methodology

4.3.2 Abstracting Implementations

In a similar fashion to how design concepts were generalised, a parallel abstraction was applied to the implementations—that is, the source code elements that are created to fulfil and realise those design concepts. Source code is typically organised into a collection of files and folders, and files contents are structured into elements according to an adopted programming paradigm (e.g., Object-Oriented). Some source code may also adopt a combination of paradigms simultaneously. Similar to design concepts, source code structural elements often involve hierarchies, such as classes that contain properties and methods, or files that contain functions and structs. At the lower end, source code is eventually written in a specific programming language.

The purpose of the abstraction is to be able to work with the structured elements of a codebase, irrespective of the size of the element, its level in the structural hierarchy, the programming paradigm it is structured according to, and irrespective of the programming language it is written in. The following is how we defined source code implementation for the purpose of the abstraction:

Code Item: a code item (CI) is any structured element of a source codebase.

Explanation and examples: As discussed above, source code is always composed of structured elements regardless of the paradigm adopted, or the programming language it is written in. These elements are always well-demarcated and named. Thus, a code item could be a method, a function, a class, a namespace, or a whole file.

Object Model: a code item is primarily represented by a **unique identifier** that is able to distinctly identify the item across the codebase. A **payload of optional properties** could be incorporated into the model to suit the particular context—e.g., a file's path or a range of line numbers—however, the unique identifier is the only essential component required to create the abstraction. This unique identifier

should not necessarily be contrived, rather it should ideally be intrinsically deduced from the code item structure itself[18].

To account for the hierarchy of the code item within the source code structure, **references to parent and children** are incorporated into the skeleton as optional elements. Figure 4.2 shows the skeleton of this object model.

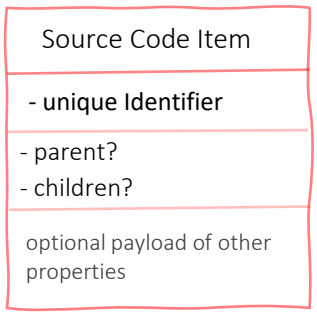


Figure 4.2: Abstract object model to capture and represent implementations (source code items) irrespective of programming paradigm and language.

Closing Remark. As is made evident by the above definitions, the key purpose of the abstraction is to be able to uniquely identify and designate any member of the design concepts, and any member of the structural code elements, within a software project. This is seen as the key enabling characteristic as it allows us to dissect both worlds, and to be able to uniquely point to each single building block and tell it apart. This does not undervalue other useful properties under either category. However, such properties are best left to the individual implementation and application context, as the value—and need—for such properties would vary accordingly. Speaking from a philosophical point of view, the abstraction is an attempt to establish a numbering and identification scheme for the building blocks of the two key aspects of the development process. It is akin to essential identification schemes adopted by other engineering disciplines—referred to later in this chapter—and is seen to allow for convenient navigation, browsing, and designation of concepts and their specific implementations. Lastly, it is worth noting that the presented abstraction should not necessarily need the creation and sustainment of new physical datatypes. On the contrary, in an ideal implementation, the abstraction model should work as a temporary encapsulation layer that only exists during runtime to represent and work with the already existing objects (e.g., issues on Jira, or a method in a source file). This way, unnecessary middleware objects are avoided, resulting in cleaner implementations. The above conceptual ideas should become clearer in Chapter 6, when a practical implementation is discussed in detail.

[18] In practice, a form of identification is often readily used by developers and could be easily formalised and structured to work as a viable unique identifier. An implementation of this is covered in detail in Chapter 6.

4.4 Traceability—an Emergent Benefit

Once the design concepts and the implementations—two primary representations of the development process—are abstracted and captured in a generic form, then a number of benefits emerge with regard to supporting the development process and the daily tasks undertaken by developers. This work is originally motivated by this notion of synchronising the two aspects and making them readily accessible to the user, due in particular to recognising the potential benefits it could bring. As expressed in Chapter 3, one of those recognised potentials was the area of artefact traceability, which the work has paid particular attention to.

Unique Identifiers as Tracelinks. The abstracted models of design concepts and implementations enable a relatively simple approach to realising bi-directional artefact traceability. To illustrate, we draw on a practical example of a developer working to implement a particular user story. The developer spends a few hours (or days) to produce working source code that implements the intended functionality of the user story. Their implementation is always translated into either newly contributed *Code Items* (see earlier section) or modifications of existing ones. Since the code items are now readily captured by a common abstract object—irrespective of their nature (being a class, a method, a file, and so on)—and each has its own unique identifier, then all that is needed to create a tracelink is to simply add a reference of each of those code items to the particular user story they are implementing. Additionally, a user story is also readily captured by the abstract Design Concept object, and has its own unique identifier. Thus, each code item could also be linked to (or tagged with) the user story that it is fulfilling (whether fully or partially)[19]. Thanks to the abstraction of both the design concepts and implementations, realising such bi-directional traceability becomes possible at any level or type of a design concept (for example, a similar scenario can be applied to a Jira issue, or to a bug fix) and at any level or type of a code item (for example, a class, a function, a namespace). Also, thanks to the abstraction, it becomes possible to build a tool that can facilitate such functionality irrespective of the nature of the design concepts or the implementations. Moreover, since the design concept model is generic, it is able to represent higher-level design concepts such as epic user stories or features, or even top-level design concepts that capture a system's key behaviours. However, those items are not actioned directly by developers, but rather, only their lower-level children or representations are actioned. Hence, an implementation tool should ensure that tracelinks are only created for these lower level design items that are directly actioned by developers. In subsequent chapters of this work, these items are referred to as '**Actionable Design Items**'. Dependencies and tracelinks to the higher-level design concepts can then be inferred through the parent-child hierarchy, facilitated via the parent and

[19] We are not referring to or encouraging the creation of duplicate tracelinks though, as will be shortly elaborated below.

child references. Indirect tracelinks from design concept to parent code items can also be similarly inferred. The diagrams in Figure 4.4 and Figure 4.5 should be helpful in further explaining this concept.

Remark. In principle, the traceability can be established by either referencing a design concept with the identifiers of its implementing Code Items, or vice versa—by referencing the Code Items with the identifier of the Design Concept (or concepts) that they implement. In other words, **the primary tracelink datapoints** could be stored in either of the design concepts or the code items to form the bond between the two. However, when contemplating the practicality of each approach during our implementation, it was found that the former approach is more feasible and leads to a cleaner implementation. This is elaborated further in Chapter 6, when the implementation is discussed. Regardless of where the primary tracelink datapoints are kept, once the bond is established, the implementing tool could then make use of the unique identifiers to append visible tags to both types of objects for user convenience and as part of a user-facing presentation—as is demonstrated later. The primary bond should always be established and maintained in one single origin, however, to ensure a cleaner and simpler implementation. Figure 4.3 illustrates an updated view of the Design Concept model, with the primary tracelink datapoint now added as a list of identifiers of related code items.

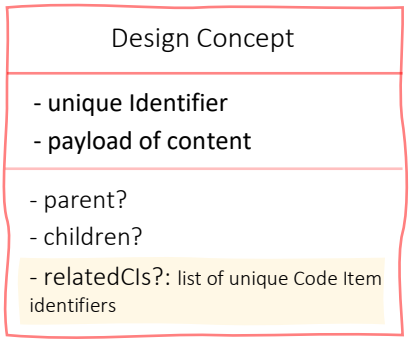


Figure 4.3: Design Concept model revisited, with tracelink datapoint now represented by a list of code item identifiers that are implementing the design concept

Reflections. The need to trace original design requirements to their end products is not unique to the software engineering discipline. In fact, other engineering disciplines, particularly construction and manufacturing share similarities with respect to the process. For example, in both fields, a building or a product's lifecycle always goes through a process of initial user requirements, then concept designs, to blueprints and models, down to final construction of the physical product. Quality requirements and design reviews are also observed through monitoring and approval processes that may result in multiple versions of design documents being produced. In fact, revisions could even occur after batches of a product have already been manufactured, resulting in updated designs being produced for future batches of the product. Yet, in these fields, tracing a particular product back to its original specification document, and vice versa, is rarely an issue. For example, a construction engineer could easily pull a blueprint of a dining room they built in a prior project, to reuse it in another. If a particular part of a building shows damage or failure a number of years in future, investigators and insurance people would normally have no problem in retrieving the relevant design requirements and blueprints to work on tracing back the origin of the fault. In fact, thanks to an enforced numbering and identification scheme, most construction organisations are able to pull the specific design requirements—and specific version if applicable—for the various stages and layers of a building in a systematic manner. For example, technical drawings and specification requirements are normally easily traceable for the various stages of the process, such as structural frame stage, electrical wiring stage, fixing and fit-off, final finishing layer, and so on. Similar arguments are also applicable for many manufacturing contexts, where a product is comprised of numerous parts that have their own interdependencies.

The above narrative represents certainly no more than a reflection drawn from experience in an attempt to elicit inspiration. Software engineering is admittedly very different in nature from other engineering disciplines and has its own and unique challenges. For one, the intangibility of the end products and the extent of interrelations and dependencies perhaps present an irreconcilable distinction. However, we believe there is still a space of similarities in the overall outline of the processes involved, that present an opportunity for lessons to be learned. One such lesson is the fact that strict adoption of identification and labelling—as well as versioning—of the various artefacts and building blocks involved in the engineering process (which includes the design artefacts and the eventual product artefacts) has allowed those disciplines to overcome the traceability issue in their domain to a large extent. The problem of retaining and making accessible the original design concepts for the various stages of the process appear to have been largely solved in these domains based on this simple—yet effective—principle.

Below are two diagrams that summarise and illustrate the discussion and concepts presented above.

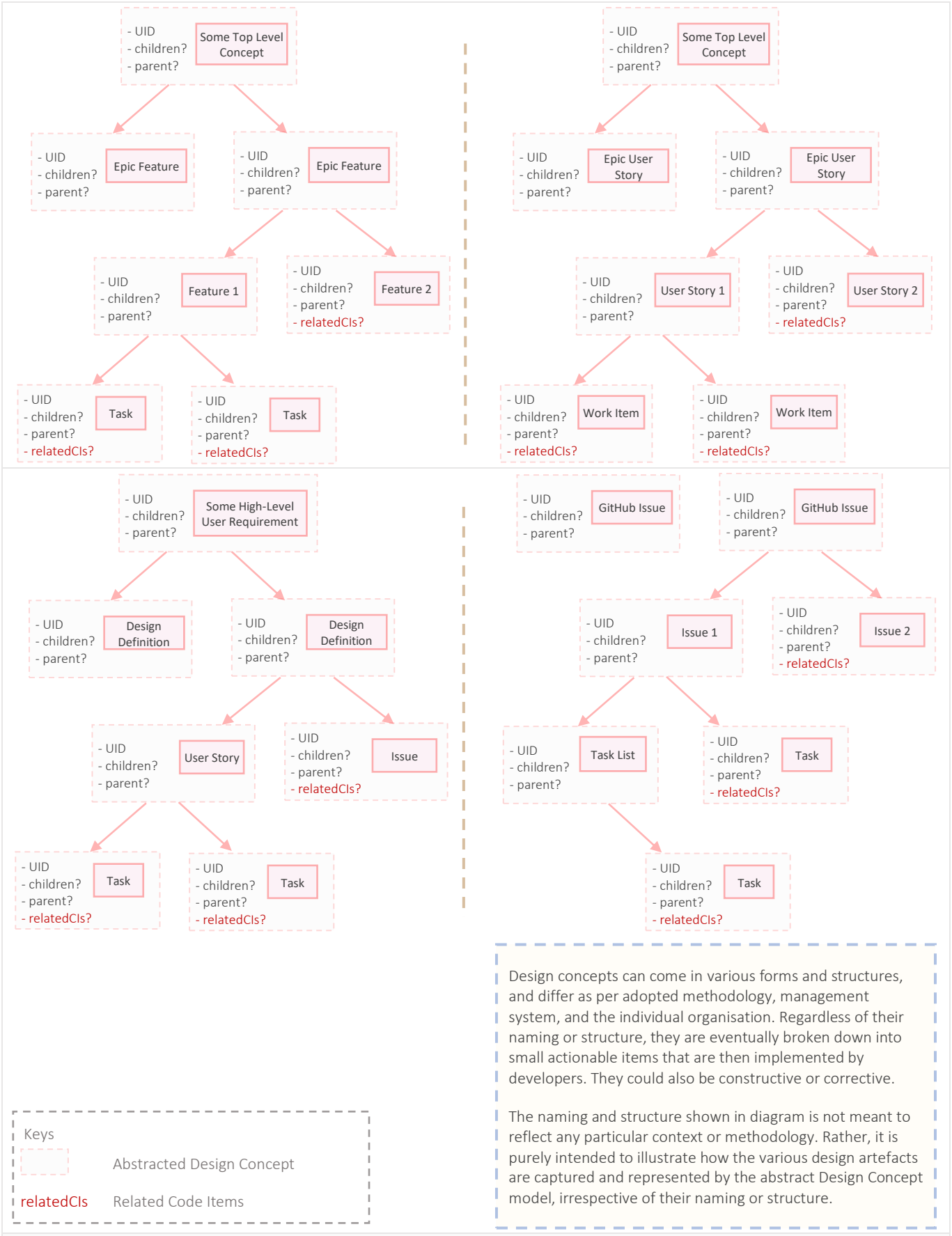


Figure 4.4: Overall diagram illustrating how the design concept abstraction is used to represent any design artefact. Note that tracelinks are only captured for those artefacts that are directly actioned by developers.

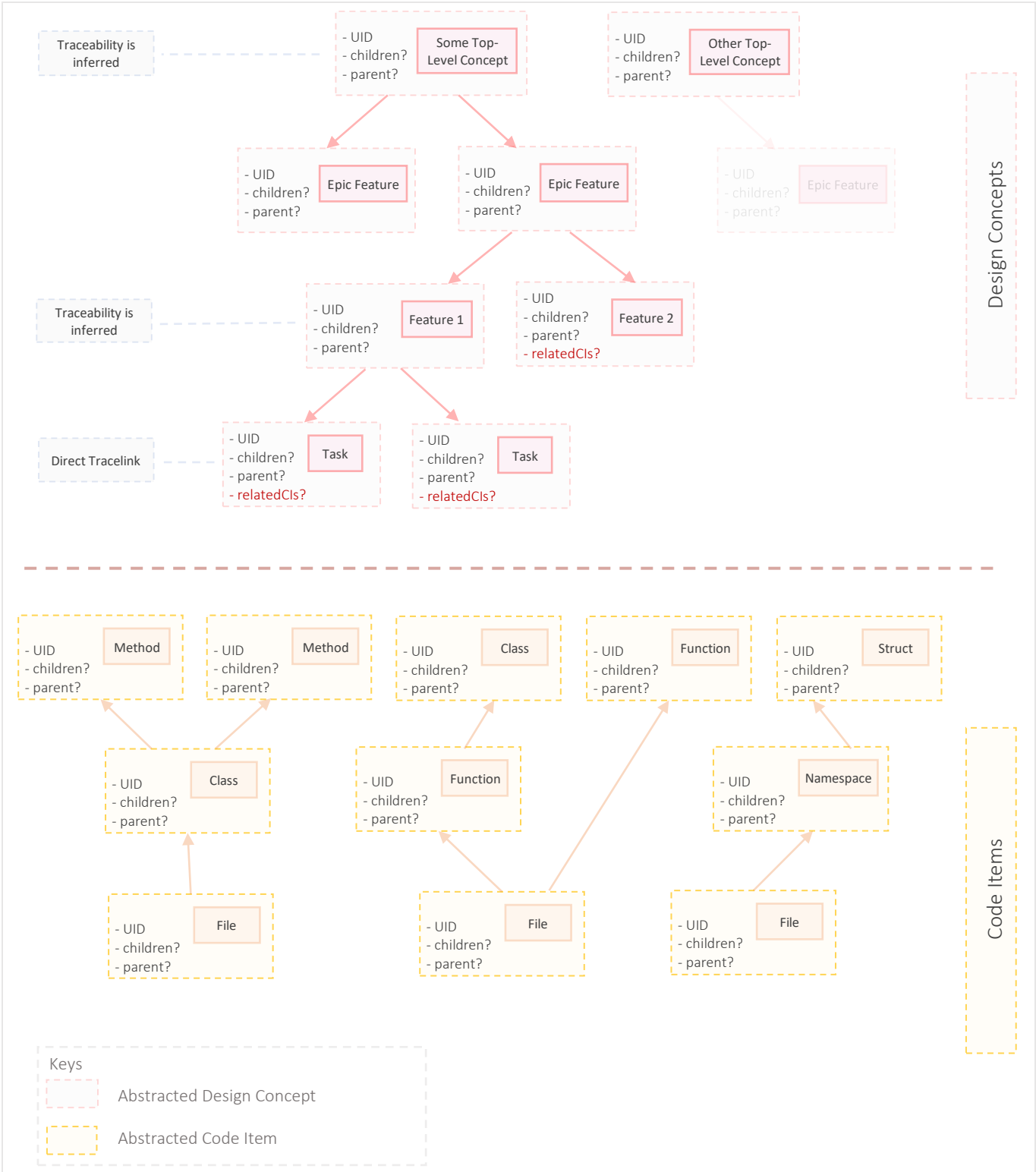


Figure 4.5: Overall diagram illustrating how the abstract Design Concept and Code Item models are used to capture both worlds of the software process—the design and the implementation. Note that the code structure shown is not meant to reflect any specific programming paradigm or language. Rather, it is only intended to illustrate how the Code Item abstract model is used to capture and represent the various code structures, irrespective of programming language. More elaboration is provided later in Chapter 6. The overall diagram, with its upper and lower sections, is meant to illustrate a single abstract view of both worlds, from top level concepts down to implementation.

Closing Remark. This section was not intended to present a full approach for realising traceability, rather it simply serves to illustrate how the introduced abstraction makes achieving such traceability possible—and relatively easy. An actual and full implementation that uses this concept is presented in later chapters. Interestingly, while this work had different origins and motivations, our effort to synchronise the design concepts with their implementation appears to present a promising application in artefact traceability, which crosses paths and shares parallels with the efforts of researchers in the traceability discipline. For example, the work of Agrawal et al., covered in Chapter 2, illustrates efforts at establishing tracelinks between features and source code, but focuses specifically on safety-critical systems, developing their approach around Safety Assurance Cases and safety requirements (Agrawal et al. 2019). The work of Delater and Paech (and later of Seiler and Paech) attempts also to establish tracelinks between features and source code, but builds their approach around a body of high-level system 'Features', 'Feature Descriptions', 'Requirements', and 'Work Items' (Delater and Paech 2013d; Seiler and Paech 2017; Seiler, Hubner, and Paech 2019). The concept introduced here is deliberately generic and is intended to enable the establishing of tracelinks across any application context, regardless of the 'type' of design artefacts, the development practice adopted, or the language of the resulting source code. Also, both works above are primarily aimed at evaluating the viability of their approaches (which in both cases is described as providing 'lightweight' traceability) rather than demonstrating a practical implementation. While both works do cite the use of tool support to facilitate the approach, the tools and the extent of their implementation is not covered, as the papers primarily focus on presenting their approach and its evaluation. As for the term 'lightweight', in the field of traceability research it is commonly used to refer to approaches that are especially not based on sophisticated techniques used to 'reconstruct' tracelinks in retrospect—which often apply machine learning techniques to rediscover and build the tracelinks. Instead, it is often used to refer to simpler approaches, which typically involve a form of active input from users. In other words, lightweight traceability is often characterised as being 'proactive' to prevent the problem from occurring in the first place, while other complex approaches are 'remedial' in nature, focusing on rediscovering and reconstructing the missing links after the fact. In this spirit, it is asserted that the concept presented here offers a lightweight or 'affordable' mechanism—as other researchers refer to it (Marcén et al. 2020)—to realising traceability between design artefacts and implementation. This is due primarily to the abstraction and the identification scheme that make it possible to expose those objects as generic building blocks, that could then be pointed to and referred to. In fact, as is illustrated later in this work, this generalisation enabled by the identification scheme makes it possible to build tools where the design concepts and implementations are brought side by side, allowing the user to navigate and explore a product's features alongside their implementations at one place and in context.

4.5 Proactive Tagging—a Promising Approach Towards Realising Traceability at the Code Item Level

In a perfectly structured world it would be ideal if each design concept could be tightly linked to the specific code items that implement it—this includes high level design concepts down to the actionable ones such as user stories or bug fixes—via a strong bond that principally unifies the two worlds. Such unification was a motive driving the early stages of this work. Reflections in the prior section suggest that similar tight bonds appear to have been achievable in other disciplines.

In the early stages of this work, realising such a level of bondedness seemed highly unlikely in software engineering. However, as the work progressed, evidence from the literature as well as that collected from collaborations with experts from industry seemed to suggest that there appears to be potential promise of this. Moreover, in light of the abstraction presented earlier, creating such bonds could essentially be achieved if each developer was willing to declare what code items they have contributed in their efforts towards implementing a particular design concept.

Interest in lightweight and 'proactive' traceability mechanisms appears to be increasingly gaining momentum among traceability researchers—probably driven by an advantageous and particular growing practice among development communities that make it possible. As results from our collaborative work with expert users later reveal, the practice of tagging commit messages with the IDs of issues, tasks, or user stories, appears to be more common than initially perceived. More importantly, developers appear to be more open and willing to practice the tagging of their work than initially thought. In fact, equipped with the right tooling that facilitates this with minimal effort, it appears that there is good potential to establish traceability right at the time of development, avoiding the need for expensive rediscovery approaches applied after developers have moved on.

More importantly, researchers in the traceability field are indeed showing increased interest in such approaches. For example, the work of (Agrawal et al. 2019) employs tags in commit messages to refer to Jira Issues. In 2017, (Seiler and Paech 2017) conducted a study specifically to evaluate the viability of tagging as an alternative approach to maintaining traceability. Their work also attempts to create tagging in the source code itself, at method and class levels. Also, in 2015, Cleland-Huang reported on a project where developers were asked to link 190,000 design artefacts to their corresponding test cases and code artefacts (Cleland-Huang 2015). In 2021, Hammoudi et al. collaborated with the original developers to collect tracelinks at method levels for three open-source projects (Hammoudi et al. 2021).

In summary, we argue that achieving traceability between individual code items and their design concepts might not be as challenging as previously thought and reported (Narayan et al. 2012). With

increased acceptance in the community, and with the proper tools that support the practice, we believe that there is good promise to realise this level of traceability, and that the community appears to be ready to embrace it. Indeed, developers appear to be willing to engage in proactive tagging of individual code items as long as their effort is kept to a minimum. In the coming chapters, and equipped with the abstractions introduced here, we report the implementation of a tool that works in part towards this goal, and the evaluations of that tool conducted with industry experts.

4.6 Software Structure Visualisation—Reducing Cognitive Load, Navigating Concepts and Implementations in a Unified Form

Earlier in this chapter parallels were drawn between the process of software engineering and that of construction and manufacturing, in order to foster support around traceability and item identification. In this section, we turn our attention to particular aspects of software engineering that are different, and unique to the discipline. In contrast to those other disciplines, software engineering is characterised by the very frequent changes that are commonly required to sustain a product throughout its lifecycle. Another particular characteristic is the personal nature of the craft of producing source code: a particular feature could be handed to two developers who would then each produce a separate successful implementation, with exactly the same user behaviour, and yet the inner implementation (source code) could differ markedly. Yet, even more interestingly, is that it may be that other developers would not be able to directly and readily tell how and where that particular implementation was accomplished within the codebase. The problem only exacerbates over time, and the same developer who originally produced the feature could themselves find it challenging to identify where exactly their implementation is located. This problem of separation between original concept and implementation thus becomes more impactful and can have more significant consequences in the software engineering context than in other disciplines—and addressing it hence becomes ever more important.

This underlines the motivation to work towards unifying the two worlds of the software development process, so that users come to be able to navigate and explore its building blocks conveniently, much like switching between two views. Such unification would not be possible, however, without the existence of that strong bond between each individual feature and its exact implementation. Research in the traceability domain makes capturing and establishing this bond possible. From this perspective, working towards enabling traceability was thus also seen as a path towards achieving the original objective of unifying the two worlds.

Once each feature is tightly linked to its implementation, the outcome is that each original design concept across the higher hierarchies also becomes linked to its implementation. However, a medium is then needed to present this unification to users so they may benefit from it in their daily practical work. Given the motivation and support discussed in earlier chapters, software structure visualisation is considered to present a promising medium to offer and make this unified world available to users.

The benefits of software visualisation in succinctly presenting information regarding the potentially complex environment of software source code is not just well-established in literature, its potential around reducing cognitive and mental effort appear to also be recognised and acknowledged by the

development community. Support from the literature has been presented and discussed in earlier chapters. However, our early work with experts from industry has also shown that there is good interest in and appreciation of the ability of visualisations to convey knowledge succinctly and readily about a system's codebase. In fact, developers in the community increasingly appear to have also been complaining about the difficulty they face when working with large and dense codebases. For example, during our technology research phase, we came across a community-initiated tool named 'Sourcetrail'[20] that, driven by the mental overload problem, resorted to creating a visualisation tool to attempt to aid peer developers with what they called 'building a mental model'. We quote here their literal motivating words as we believe they are valuable and affirm issues raised in literature: *"… regardless of the language, if a project reaches a certain size, it's hard to keep a consistent mental model of the source code's structure"*. They then continue: *"the problem here is…. the high information density (sic) of code. Every line in the source code has a purpose and as software developers, we spend most of your (sic) time searching for those small pieces that are currently relevant. Why can't we take a step back and see how the components connect with each other, without constantly looking at every detail of code?"* ("Sourcetrail" 2020). At this point, it is worth drawing a connection to our earlier discussion where we argued that developers often have to build their own mental model of the code they are working on, and that the visualisation is not just a mechanism to aid and alleviate this task, but has the potential to offer a *shared* mental model among a team.

In this regard, we reiterate our motivation that bringing the original design and implementation together, and making them readily and visually accessible to users, is not merely useful for traceability applications. Rather, we argue that it has potential for a range of applications, the most evident of which is the problem of comprehending large codebases, but it could also support other activities such as team communication, onboarding new members, conducting software quality analysis, and conducting change impact analysis. Such potential is revealed during our evaluation activities with the expert users, and is elaborated further in the following chapters.

[20] See: https://sourcetrail.com/. The project has been decommissioned as of 2021, however a GitHub page remains available and original website may also be accessed through the web archive tool: http://web.archive.org.

Chapter 5
TECHNOLOGY RESEARCH & ASSESSMENT

5.1 Prelude

This chapter presents and discusses the technologies that came to shape this research. From the selection of the 3D Graphics Engine to the environment around which the tool is built, it discusses how the various building blocks and infrastructure elements were selected to develop the concept and realise it into a functioning product where the overall research concept can be tested and evaluated.

Discussing and engaging in the design decisions pertaining to the technology underpinning this research is important for two reasons. On the one hand, it serves to demonstrate the robustness of the work in terms of its academic rigor. As has been argued in Chapter 3, part of what distinguishes academic design science research from routine systems development is the former being strongly and soundly informed by the knowledge base of the field (A. Hevner and Chatterjee 2010; Iivari 2007). While theories, frameworks, and established knowledge are readily evident elements to form the basis of any successful research, a less obvious part—but which is equally important and is especially relevant to design science—is the extant base of technology. This could be technology produced as output of earlier research works. Less attended to though, is the technology extant in—or made available by—the field's industry and its practicing community. This is an important aspect to consider to ensure the relevance of the research (and its prospective output) to the field's domain and its practicing community. Ignoring this part could risk making the research out of touch with the environment it is intended to serve. It is postulated in this work that this point is probably a major factor as to why much software visualisation research has failed to make it to light to the industry or the practicing community.

On the other hand, covering the technology and its assessment process is also beneficial for the sake of knowledge transfer. It is hoped that it could form a basis for future researchers who might wish to build on and advance the research without having to re-invent the wheel. Instead, one could have the benefit of starting where other researchers have stopped.

The following sections first present a big picture of the type of technology parts being sought—the building blocks—and the purpose of each. This includes the criteria that must be met by each part in order to realise the overall goal of the research tool. The key technologies are then covered individually with some details where relevant.

5.2 The Building Blocks

Before discussing the details of the backbone technological parts, it is sensible to first present a high-level view of the objective behind the research tool, what capabilities are being sought, and how the various parts need to fit together to achieve these capabilities.

3D Graphics Engine. One major concept of the research is the visual representation of software intangible entities (e.g., a code item such as a class or module) into physical-like metaphorical objects. This requires a graphics library to design and model the objects in the 2D or 3D space, and an engine to render the scene and display it on the screen. Since our work capitalises on scaffolding user's cognitive ability by taking advantage of the brain's spatial quality in amplifying knowledge (see Chapter 4), a 3D library is imperative in order to take advantage of the added richness provided by the third dimension. The details of the 3D graphics library and the factors influencing its selection are covered in section 5.3.

Development Dashboard API. Another key aspect of the research involves working closely with the software development process and its artefacts. For example, part of the problem space being navigated is making development artefacts such as user stories—or any other form of user requirements—readily accessible to the developer within the development environment[21]. An additional aim of the research is to synchronise those requirements with the specific code items they relate to. With the present domination of the Agile software development practices, comes the rising popularity of cloud-based[22] development and project management dashboards such as Jira and Asana. This is very instrumental as it makes the design and development requirements (or design artefacts in general) to be now better accessible to third party tools—as compared to traditional and locally-run requirement management tools (e.g., IBM's DOORS)—or even worse, word processors or spreadsheets[23]. Section 5.4 discusses the selection of the agile development dashboard, its API, and the drives behind it.

Source Code Modeller/Parser. In order to visualise source code items and present them as physical-like metaphorical objects, a codebase must first be syntactically parsed to build its constituent elements and structure into a corresponding accessible model. The model needs to accurately reflect the structure of the entire codebase, from its top level components such as packages and modules, down to individual functions or methods inside the source files. Section 5.5 elaborates on the features being

[21] See Design Item definition in Chapter 4.

[22] Most of today's cloud-based dashboards also offer the option to run their services on private servers for paying clients.

[23] Research shows that large number of organisations still use MS Word or Excel to manage their user requirements (Lian et al. 2016; Riegel and Doerr 2015).

sought in a parser solution, which are crucial to building the tool. Some options that were examined in the process are also briefly covered.

Hosting IDE. A primary goal of the research tool is to bring the benefits of software visualisation academic research out to the practicing community and to the everyday developer. An Integrated Development Environment (IDE) is where a developer spends most of his or her day around. A sensible decision becomes thus to bring those benefits to the IDE itself, minimising any impact or disruption to the existing workflow of the developer. More importantly, it maximises the chance of a developer coming across the tool in the first place, and then hopefully adopting it. However, choosing the appropriate IDE is a crucial factor into whether this goal is to be achieved successfully or not. **Section 5.6** elaborates more on the role of the IDE and the factors guiding the selection process and decision.

The diagram in Figure 5.1 illustrates the technological components discussed above and how they come together to form the backbone of the research tool.

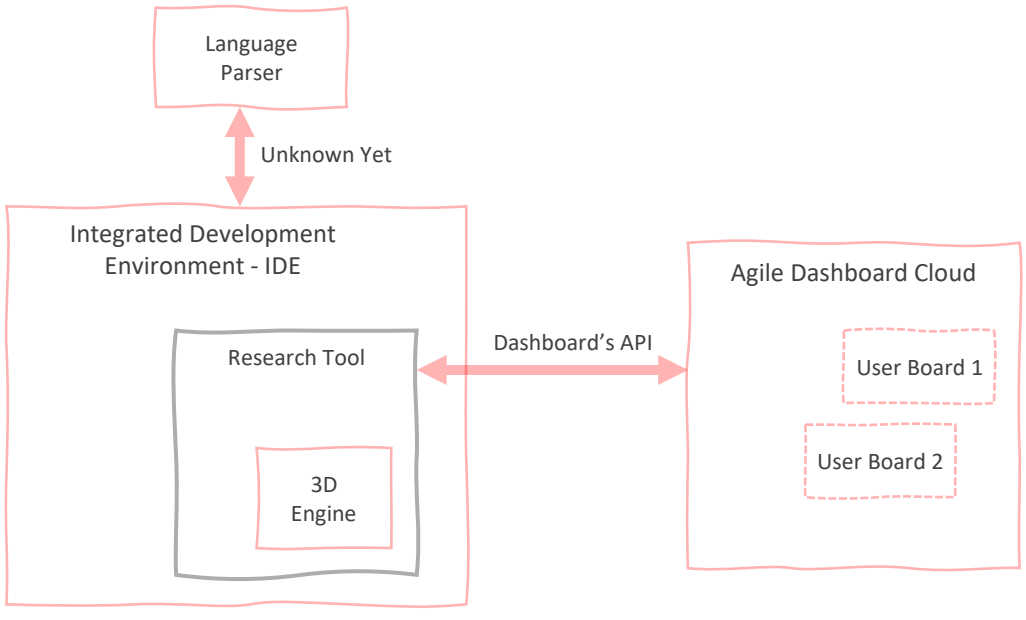


Figure 5.1: Basic diagram illustrating key technological components that form the backbone of the research tool.

5.3 Graphics Engines and APIs

3D graphics are used heavily by almost all sectors of today's industry, and there is a very large number of libraries to choose from. The capabilities of each, and hence its uses and applications, vary immensely as well. Some are very sophisticated and feature-loaded autonomous platforms—better known as game engines. The capabilities can range from simple object creation and modelling—e.g., creating a 3D object to be loaded later into another environment or displayed on the web—to those full-fledged powerful engines that can render immensely complex scenes on a user's screen and provide advanced interaction capabilities. Unity and Unreal are two examples on the extreme end, which are well-known and popular among the 3D animation and game designing industry. They are self-sustained development environments that are overly complex, heavy, and can be outright deemed unsuitable for our purpose. In contrast, there is D3, X3DOM, and XML3D[24] as examples from the lower-end and simple 3D content creation libraries.

To create the tool for this research, a 3D library had to satisfy a number of elements. Some were essential to realise the concept, while others were deemed desirable for aesthetic or performance reasons. Table 5.1 presents the factors that guided and influenced the search for a suitable 3D library, and explains the reasoning behind each.

Out of the group of 3D libraries that have been examined, a selected number are briefly discussed below. The discussion serves to review some aspects that are believed to be of particular relevancy to the researcher in the software visualisation domain. It is hoped that it will also help give the reader a sense of the nature of these libraries and what sets each apart when it comes to the kind of application they could be used in. In the process, some light will be shed on the role played by these libraries in some relevant visualisation research—which future researchers might find helpful.

D3/d3-3d. This is one of the simplest and lightest graphics library that is specifically designed for the web environment. It is particularly well-suited and popular for *information* visualisation applications. The library capitalises on modern web standards (HTML, SVG, and CSS) to bring their full potential to the user with comparatively very lightweight result[25]. Scenes created in D3 will render fast and natively in any modern web browser[26] thanks to the fact that it consists purely of W3C standard elements. Thus, to a browser a D3 content is rendered exactly like any other standard HTML content with no additional rendering engines or helpers required. However, it is often combined with extension libraries such d3-

[24] XML3D is largely similar to X3DOM in terms of features and capabilities, except that it is designed to strongly adhere to W3C technology stack. The library has not enjoyed much traction though and is thus not of interest here.
[25] As compared to other lightweight libraries such as X3DOM and XML3D.
[26] A modern web browser is that which is compliant with modern standards of the W3C.

3d[27] if full 3D projections are desired. Moreover, with the recent advent of cross platform desktop applications, the benefit of D3 (and any other web technology thereof) is no longer confined to the web application domain (see section 5.6). Figure 5.2 shows a visualisation example created using the D3 library. In D3, extensive data can be tightly mapped to corresponding DOM elements, which can then be visualised in numerous manners and techniques. The user can then explore the connected data interactively or choose to dig deeper to unveil further dimensions. This allows information to be presented at gradual pace and in contextual form without overloading the user if it otherwise were presented at once—i.e., avoid visual clutter problem. The reader is referred to the API's official gallery to examine what can be achieved using D3[28]. Some notable relevant works that have utilised D3 in their visualisations include '*Visualizing Software Hierarchy and Metrics over Releases*' by (Humayoun et al. 2018), and '*Visualizing Time and Geography of Open Source Software with Storygraph*' by (Shrestha, Zhu, and Miller 2013). As a side note, the works referred to are seen to employ general 'Information Visualisation' techniques rather than specialised 'Software Visualisation' technique, as discussed earlier in Chapter 2. For more on this issue, please see sections 2.2 and 5.3.

X3DOM is the modern call to the old ISO standard library, X3D. It builds on top of X3D, and in a similar fashion to D3, utilises modern web technologies to bring lightweight 3D graphics to the web without requiring any plugins to display its scenes. Unlike D3 though, X3DOM brings most of the main features of a typical 3D graphics library such as meshes, materials, lighting, shadows, and other rendering and navigation features. It is a good library where richer 3D features are desired at a lower and simpler level than a full-scale 3D engine. It is worth to note that the driving engine of X3DOM (i.e., X3D) was an instrumental library to the software visualisation discipline in its early days. It has been covered to a good extent in our early work (Alshakhouri 2013). X3DOM continues to play a strong role in some very notable and recent software visualisation works. It is featured in the recent work of Baum's et al. '*GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation*' (Baum et al. 2017). However, the lighting and material quality of the resulting rendered scenes in the authors' work appear notably to be on the lower spectrum in terms of visual aesthetics (see Figure 5.3). Interestingly, in their 2016 paper, Limberger et al. discusses a comparative study that involved the same library, X3DOM, but where the rendered scenes appeared to be of substantial higher quality (see Figure 5.4) (Limberger et al. 2016). In their subsequent paper titled '*Progressive High-Quality Rendering for Interactive Information Cartography using WebGL*' the Limberger et al. introduced an improved rendering technique using a specialised multi-frame sampling method that can be applied to different 3D Libraries to achieve a substantial improvement and high-quality visual effect

[27] https://github.com/Niekes/d3-3d
[28] D3's official gallery: https://observablehq.com/@d3/gallery

with regard to material and shading (Limberger, Pursche, et al. 2017). In this later paper, the authors apply their technique to the 3D library Three.js. However, it appears that the high quality visual effects achieved with X3DOM in their earlier 2016 paper was also a result of utilising this same specialised rendering technique. Unfortunately, it was not possible to establish if this was indeed the case.

Table 5.1: Factors guiding the search for a suitable 3D Graphics Library/API to Build the Research Tool

	Feature	Explanation	Nature
1	Integrate-able	The library has to be of a nature that makes it possible to integrate within the to-be-selected development environment (IDE). Specifically, the integration should be seamless, such that the library's APIs can be invoked directly from the IDE's extension or plugin APIs without additional middle-tier layers. Resulting scenes must also be render-able within the IDE, and without any additional layers or plugins.	Essential
2	Message Exchange	The API must support live message exchange to and from the hosting IDE's native APIs. This is important so that rendered objects inside the 3D scene can be interacted with smoothly in response to events firing in the IDE and vice versa. For example, a user can click a button in the IDE that will in response trigger a change in the 3D scene. It is equally important that interaction can be initiated the other way around too.	Essential
3	Interactivity	The API must enable a good level of user interactivity such as object picking, keyboard input, and good navigation and camera options. Support of advanced user interaction devices (e.g., motion controller) and multi-touch gestures is highly desirable. The latter is desirable because work is aiming to run user evaluations on multi-touch screens—whether as part of this research course or in future work.	Essential
4	API's Language	Ideally, the APIs' language and running environment need to be matching with that of the IDE's native APIs' language and running environment. This is to avoid the need for any middle-tier layers, which can significantly complicate data exchange, affects real-time responsivity and performance.	Desirable
5	Lightweight	The library should provide fast rendering capability and be able to do it from within the hosting IDE's environment. This should include animated objects without significantly degrading navigation responsiveness.	Desirable
6	Material, Lighting, and Rendering Quality	The library needs to feature high quality material and lighting capabilities. This includes shadows, and advanced rendering techniques such as anti-aliasing, to improve the look and appeal of objects. While not a detriment to functionality, a visualisation with better aesthetics and lighting is instrumental to user satisfaction and adoption.	Desirable

7	APIs Maturity & Community Support	In the world of 3D graphics, technology changes rapidly and without an active community of developers, a library can quickly fall behind.	Desirable
8	Ease of Learning	The library should not have a steep learning curve and should be easy to use. This is true for most high level graphics APIs as opposed to low level ones such WebGL or OpenGL. It should also have rich and active learning resources and examples supporting it.	Desirable

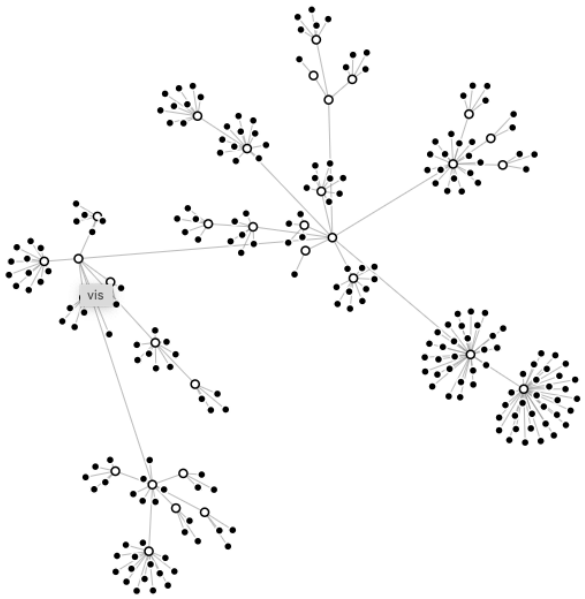


Figure 5.2: A Force-Directed Tree created using the D3 Library API. Source: https://observablehq.com/@d3/force-directed-tree

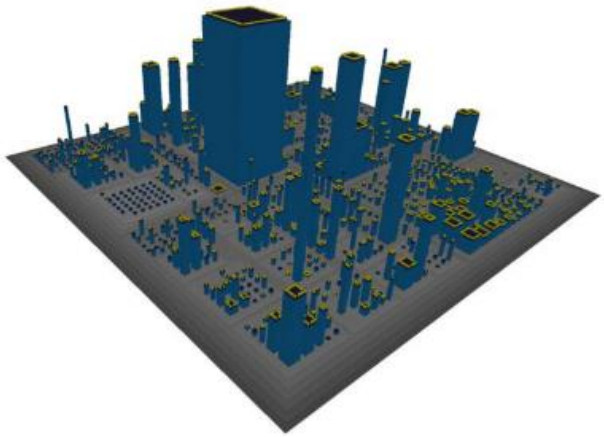


Figure 5.3: A rendered scene using X3DOM as featured in (Baum et al. 2017). Visual properties such as lighting and shadowing are noticeably low in quality.

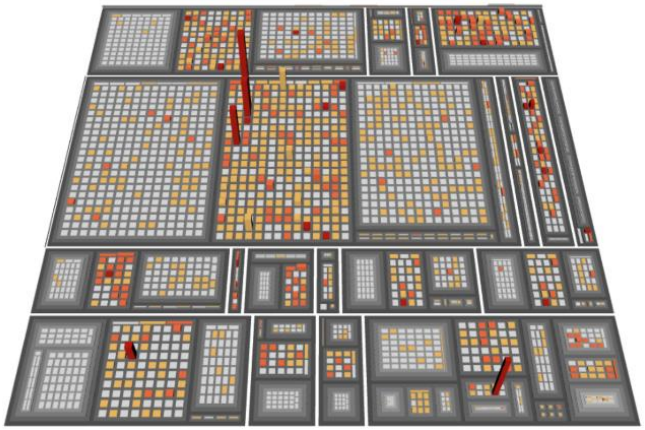


Figure 5.4: A rendered scene using X3DOM showing higher quality visual effects (Limberger et al. 2016).

```
<body>
  <h1>Hello, X3DOM!</h1>
  <p>
    This is my first html page with some 3d objects.
  </p>
  <x3d width='500px' height='400px'>
    <scene>
      <shape>
        <appearance>
          <material diffuseColor='1 0 0'></material>
        </appearance>
        <box></box>
      </shape>
      <transform translation='-3 0 0'>
        <shape>
          <appearance>
            <material diffuseColor='0 1 0'></material>
          </appearance>
          <cone></cone>
        </shape>
      </transform>
      <transform translation='3 0 0'>
        <shape>
          <appearance>
            <material diffuseColor='0 0 1'></material>
          </appearance>
          <sphere></sphere>
        </shape>
      </transform>
    </scene>
  </x3d>
</body>
```

Figure 5.5: Hello World example illustrating X3DOM's descriptive syntax. Source: https://doc.x3dom.org/tutorials/basics/hello/index.html

Findings. X3DOM is declarative 3D library, has a descriptive syntax similar to that of HTML or XML, and is primarily designed for the web. Figure 5.5 shows a hello world example to demonstrate the descriptive syntax. Despite its attractive lightweight and fast rendering quality, it lacks the flexibility and convenience offered by other APIs that are based on a full-fledged imperative programming language such as C# or Java. Furthermore, when the backend processing or the main application is running in a non-web-based environment, then a declarative library will not be an ideal solution. There will need to be a bridging or a middle-tier layer between the main data and logic processing happening in runtime (e.g., a .Net or a Java environment) and that of the rendering layer (the HTML and web). As a result, real-time interaction would be unnecessarily complex. In addition, the backend runtime will have to painstakingly generate the descriptive syntax for every rendering operation—and that is after internally building the structural model of the codebase that is to be visualised. In summary, this and other similar libraries fail the 'integrate-able' and 'Message Exchange' features being sought, without which it would not be possible to embed a 3D scene and have it display natively within an IDE. It would also not be possible to have natural and smooth interaction between objects in the rendered scene and the native UI of the IDE, as event communication would need to be bridged via a third layer between the two sides. This will become clearer when discussing the selected library in next section. On top of those two key elements, X3DOM also fails to meet the 'API's Language' criteria and to some extent, the 'Material, Lighting, and Rendering Quality' criteria.

As for D3, it is also a declarative library, but it has the advantage of being a JavaScript library and uses an imperative chain syntax—making it easier and less verbose to use. Because it is a JavaScript library, much of the capabilities of the language are available to the user, and hence this library delivers many of the features being sought in Table 5.1. However, it specifically falls short in terms of its 3D capabilities, which are only possible when combined with extensions like d3-3d, but even then, the 3D features remain still quite limited.

5.3.1 Selected Graphics Library

The research focused next on selecting an API that is based on an imperative and full-scale programming language. After reviewing popular libraries in this category—at both, academic research and development community levels—**Three.js** came as a prominent one. Three.js (also written, ThreeJs) has a highly active community of developers, has a strong and mature base of user support and examples, is based on a simple yet flexible imperative language, and satisfies all of the features being sought. It particularly offered the required level of visual effects and rendering quality—specifically, material, lighting and shading quality. While it has barely been featured in software visualisation research—only two such works were found to have used it (Limberger et al. 2013; Sinhabahu, Wimalaratne, and Wijesiriwardana 2020)—it nonetheless enjoys a very good (and rising) base of applications in information visualisation and other academic research. Figure 5.6 shows the syntax of the library with a hello world example.

Three.js is a pure JavaScript engine. Its API embraces modern features of JavaScript such as Classes, Promises with Async/Await (the advanced successor to AJAX), Arrow functions, De-structuring, Callbacks, and Modules. This makes it attractive from a development perspective. The library is also kept continuously up to date by its active open-source community of developers. Despite the traditional impressions of JavaScript, today's modern JavaScript is highly flexible and versatile, competing with the like of Java and Python. Furthermore, and as discussed in section 5.6, the advent of Electron and Node.js means that JavaScript applications are no longer confined to the frontend of the web.

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera( 75, window.innerWidth/
window.innerHeight, 0.1, 1000 );

var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );

var geometry = new THREE.BoxGeometry( 1, 1, 1 );
var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 }
);
var cube = new THREE.Mesh( geometry, material );
scene.add( cube );

camera.position.z = 5;

var animate = function () {
  requestAnimationFrame( animate );

  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;

  renderer.render( scene, camera );
};

animate();
```

Figure 5.6: Hello world example illustrating the syntax of Three.js

With Three.js looking like a promising candidate, tests and experimentations were next carried out to examine three aspects of it in-action and in the specific development environment (machine) that will be used. These are; integrate-ability in an IDE, rendering performance, and visual effects quality. The following sections discusses each accordingly.

5.3.2 Integrate-ability in IDE

To assess how well Three.js can be integrated in popular IDEs, a web canvas was set up in order to use it to display a test 3D scene. For this to work well, the canvas has ideally to appear naturally in place and feel native compared to the rest of the IDE's UI. Another crucial point is the communication between the two. As described earlier, a well-integrated state should permit events from the IDE's native UI components to be communicated seamlessly to the web canvas. The other way around must be true as well. A third important point is that the web canvas should be able to render the scene natively without any other additional components or plugins being required. Lastly, user interactivity within the web canvas should feel natural and enjoy the same level of responsiveness as that of the IDE's other UI components. This is especially important when the web canvas is actively displaying the 3D scene. Two IDEs were selected for the assessment, Eclipse and Visual Studio Code (Vscode). More on the IDE's selection is presented in section 5.6. The specifications of the test environment are presented in Table 5.2.

Eclipse. An Eclipse RCP version was used for the test—specifically, version 2021-12 (6.22.0). Eclipse APIs support a built-in simple browser making it straightforward to carry out the test. No development is required except for the Three.js test scene itself. However, Eclipse's internal browser is just that—a web browser, nothing more. To actually run the Three.js test scene, an external server is required to host and serve the Three.js script—i.e., for backend processing. A Python 3.9.6 HTTP server was used for this purpose. Figure 5.7 shows the result of this test.

Vscode. Vscode has a more integrated and advanced web canvas that is called Webview. Setting up a Webview in vscode is more involved requiring some development. However, once done, it offers incomparably greater flexibility. Thanks to the nature of Vscode, its Webview will host and serve any web content without requiring external components for backend processing. More technical explanation on this matter is provided in section 5.6. Figure 5.8 shows a Three.js test scene running natively in Vscode environment inside a custom-built Webview canvas.

Findings. Both Eclipse and Vscode offered a native-looking web canvas. Looking at the results pictures, Vscode's canvas does undoubtedly feel more in place. The web address and navigation bars in Eclipse's

internal web browser can be set to hidden, which could make the canvas to look more in place with the rest of the UI. However, deep under the hood, Vscode's webview is completely native and homogenous with the other Vscode components—other native UIs and widgets are in fact built using internal webviews.

As for events communication, Vscode's webview was found to be incomparably better equipped. While custom user-contributed webviews are sandboxed for security purposes, they still offer native communication to and from the rest of vscode's UI through message passing technique. The developer gets full access to the native vscode APIs, thus allowing wide range of opportunities for the developer. In contrast, there is no direct event communication in Eclipse between the internal web browser content and its plugin development APIs—the RCP library package. Unlike Vscode, the two environments here are of completely different nature—one is Java and the other is web. Eclipse does nonetheless offer a basic and indirect one-way communication. The developer can write a JavaScript method that will indirectly invoke a Java function defined in their custom plugin[29]. However, support of the opposite way of communication doesn't seem to be possible though—investigations carried out failed to find any possible means to achieve it, at least not at direct level. This is a far comparative result from the native two-way communication pipeline offered by Vscode. Being a modern and recently introduced IDE, Vscode had obviously the advantage of being architecturally designed with modern technology and needs in mind.

Lastly, while rendering of the Three.js scene required no external component in both environments, the need for an external server to host the web content in the case of Eclipse adds clearly undesired complexity.

The next section discusses the subject of scene rendering, assessing its performance between the two environments.

Table 5.2: Testing Environment Specification

Monitor	BANG & OLUFSEN, HP Envy32 inch.
Machine	MacBook Pro (Retina, 15-inch, Mid 2015). Processor: 2.5 GHz, Quad-Core Intel Core i7. Memory: 16 GB, 1600 MHz DDR3. Graphics: AMD Radeon R9 M370X 2 GB. Running macOS Monterey.
Eclipse	Eclipse IDE for RCP and RAP Developers. Version: 2021-12 (4.22.0)
Vscode	Version: 1.62.3, Date: 2021-11-17, Electron: 13.5.2, Chrome: 91.0.4472.164, Node.js: 14.16.0, V8: 9.1.269.39-electron.0, OS: Darwin x64 21.2.0
Background Apps	Chrome with three tabs, one playing video music. MS OneNote. MS Word with two windows open, one Finder window.

[29] For more info, see Class `BrowserFunction` in package `org.eclipse.swt.browser`.

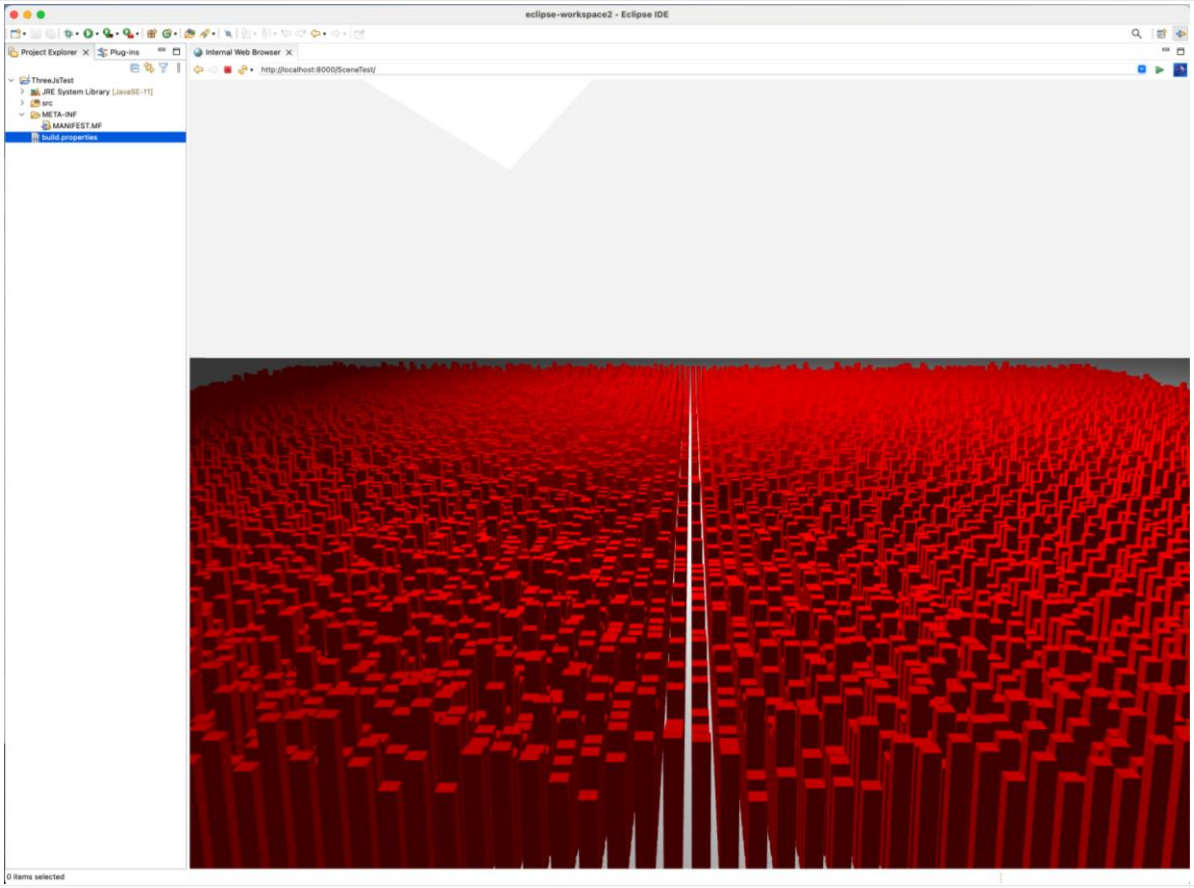


Figure 5.7: Three.js test scene integrated within Eclipse environment (using the built-in browser). The scene shown is part of a performance assessment activity.

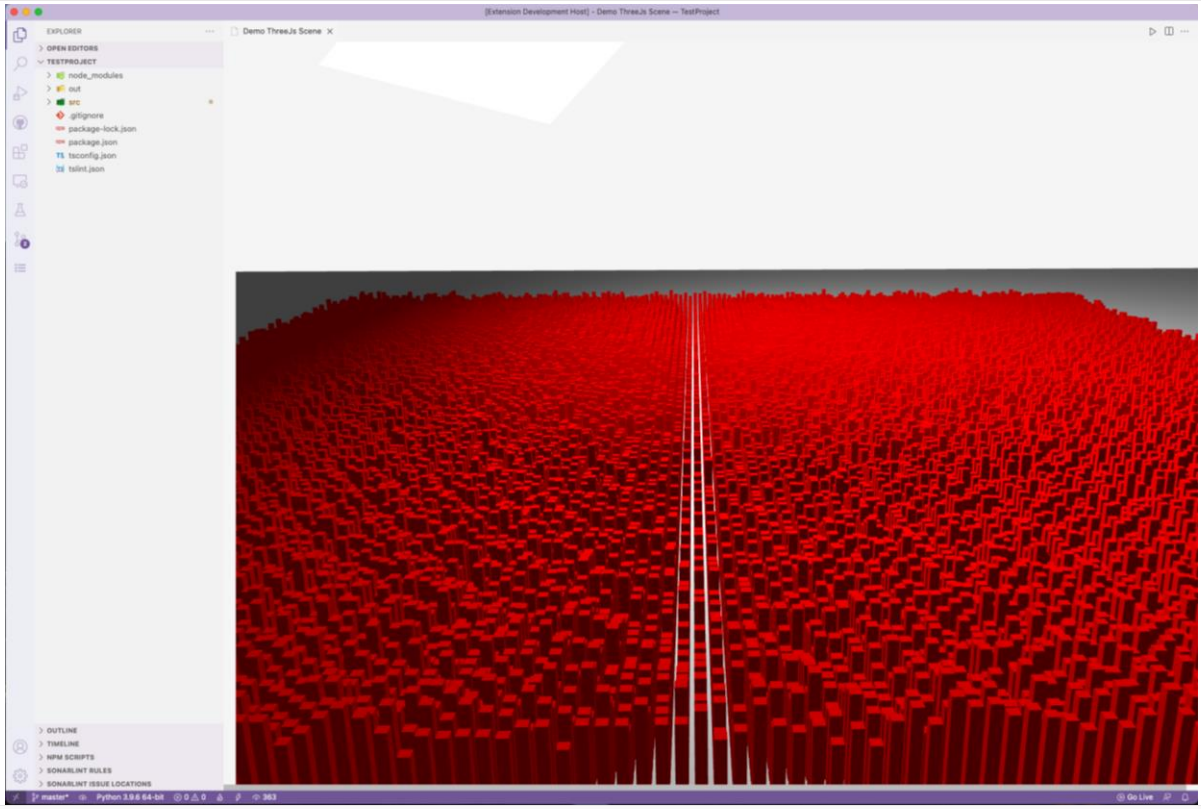


Figure 5.8: Three.js test scene running in a custom-developed Vscode Webview. The scene shown is part of a performance assessment activity.

5.3.3 Rendering Performance

Responsiveness and user interactivity came out as a major dividing factor between the two environments. To assess the rendering performance in the embedded canvases of the two IDEs, a test scene was designed comprising of a total of 22,500 3D object nodes. Each object was further animated to mimic use cases that will be built in the actual tool. To understand the relevance of this assessment, a little of explanation is required before proceeding with discussing the test outcome.

Factors Influencing Rendering Performance. In the world of 3D graphics, performance and rendering capacity is highly influenced by the number of nodes being displayed. In particular, it is tightly dependent on the number of geometries and materials used to create those nodes, and on how meshes are created. Figure 5.9 is helpful in understanding how these elements fit together. Meshes can be thought of as placeholders for where an object need to be positioned in the scene. It could also hold other properties such as orientation and scale. Geometry is what gives the mesh its physical shape. Material comes next to give the mesh its final appearance properties, including color. Depending on the nature of the scene, it is possible to share geometries and material among multiple meshes. This will drastically improve performance when the scene has large number of nodes. 'Shared geometry' is the term used in graphics to refer to this performance improvement technique.

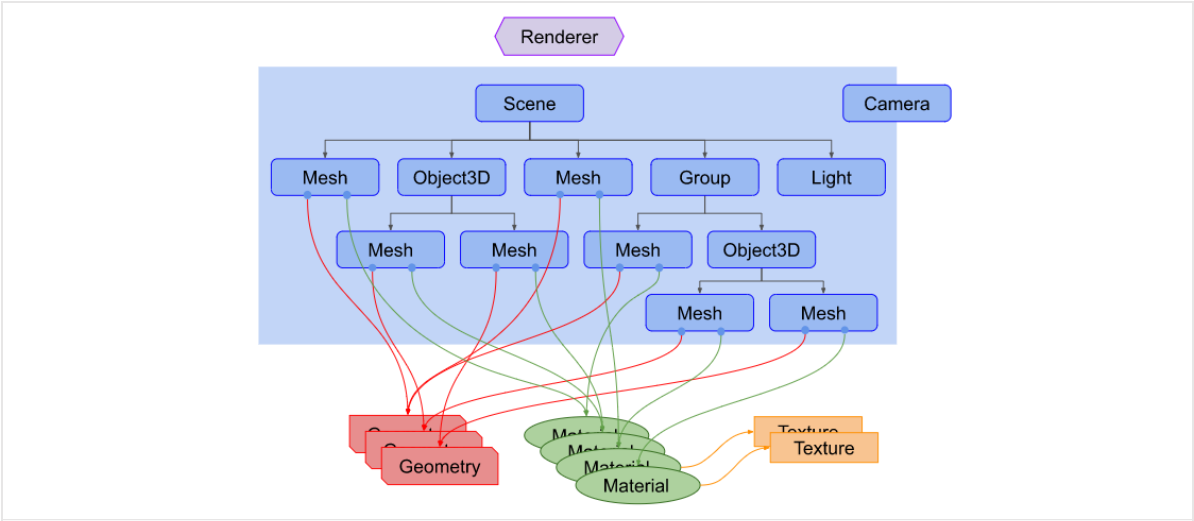


Figure 5.9: An illustration of typical composition elements in a 3D world scene. Source: Three.js documentation, https://threejs.org/

These details are important because the number of nodes in a scene is strongly related to the nature of application in this research. The visualisation technique used requires creating a node for each structural component in the codebase (e.g., a folder, a class, or a method) that is to be visualised. While shared geometry technique is possible in the application of this research, such a solution would be convoluted, adding significant programming complexity. It would also result in object picking feature—

which is important to the application here—being effectively useless. Thus for the purpose of performance assessment, separate geometries were used to represent the 22,500 objects—the material was shared though. While some performance improvements could be implemented to that set up, the test scene is purposely intended to push the performance assessment to its extreme case in order to obtain a benchmark result.

> Remark. Three.js offers two inbuilt features to improve performance—*BufferGeometry* and *InstancedMesh*. The former stores node attributes in buffers to reduce the cost of passing the data to the GPU while the latter is a new attempt introduced to reuse Mesh objects through instancing. BufferGeometry is adopted in this research but not InstancedMesh as it does not fit with the nature of the scene that needs to be created. During performance assessment however, it was noticed that improvements brought by BufferGeometry was insignificant. This is likely to be particular case related to the nature of the test scene, which was set up to closely mimic the actual visualisation scene that will be produced, rather than a generic outcome.

Findings. Both Eclipse and Vscode were able to load the test scene in a reasonably acceptable time-frame. Eclipse came slightly faster with an average of 1.65 seconds, while vscode loaded the scene in an average of 1.86 seconds. However, after loading the scene Eclipse became immediately unresponsive and kept crashing upon any attempt to navigate the scene[30]. It also struggled to render the animation of the objects (reduced frames per second). On the other hand, Vscode was able to handle the very large number of nodes smoothly without any noticeable deterioration in the responsiveness of the IDE. Navigating the 3D scene was reasonably acceptable as well, with only slight delay in responsiveness—quite expected at this level of a very large number of nodes. Animation of the objects was rendered successfully by Vscode without any visible impact on the frames per second rate. Figure 5.10 and Figure 5.11 show the memory and CPU/GPU loads of the test machine during both tests.

To simulate a machine environment that is close to that of a typical developer, the test was repeated with few other applications running in the background. As expected, the load time increased slightly in this case for both IDEs by few milliseconds. Figure 5.12, shows the results of the performance assessment carried out.

[30] This could potentially be due to the default allocated memory for the eclipse virtual machine being exceeded. Modifying this configuration was not attempted though as it is of no significance to the assessment purpose.

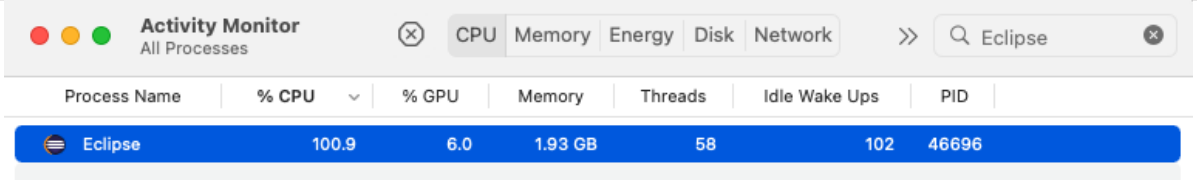


Figure 5.10: Resource usage by Eclipse during the performance assessment.

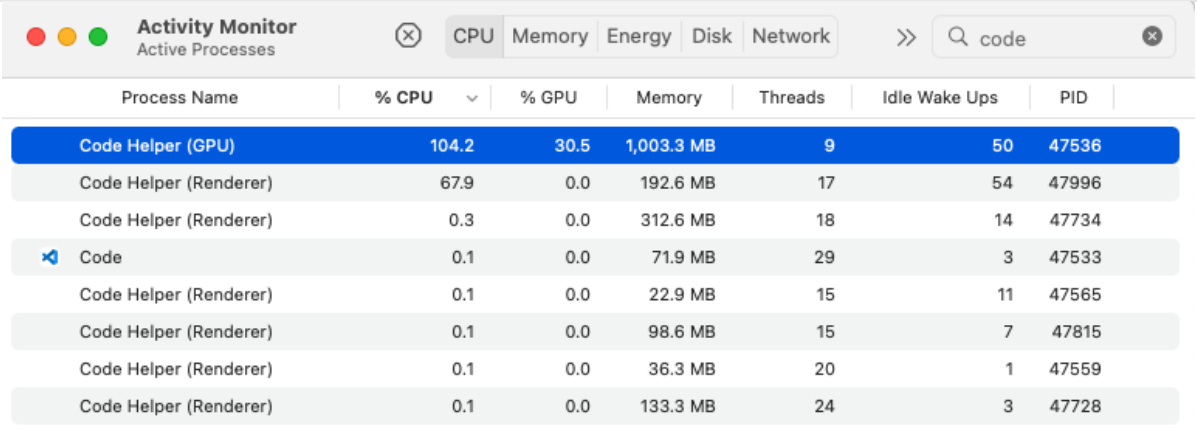


Figure 5.11: Resource usage by Vscode during the performance assessment. Unlike Eclipse, vscode is seen to be utilising parallel processing power as well as better use of the GPU to handle the rendering.

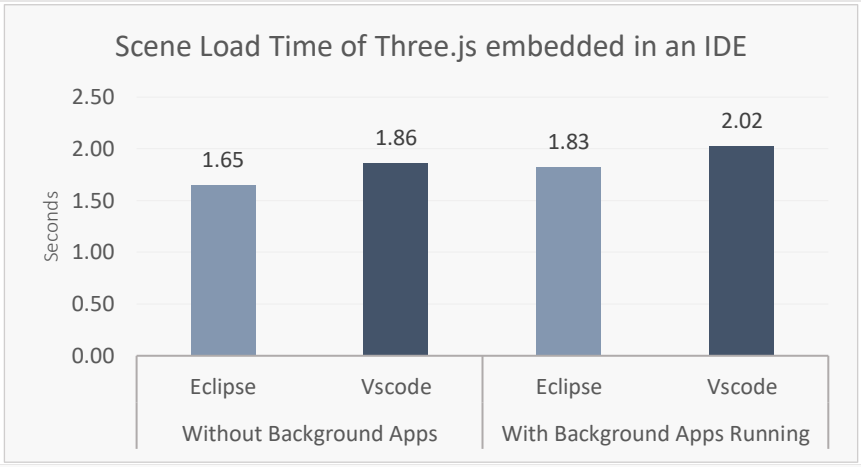


Figure 5.12: Load time results obtained during performance assessment of Three.js when integrated in Eclipse and Vscode, respectively. A test scene of 22,500 full-3D animated objects was used, with separate[31] non-buffered geometries, and shared material. Load time is averaged over 10 runs that were carried out in 3 separate settings. See Table 5.2 for testing environment details.

To put things in perspective, it is of particular interest to draw comparison of these results with those obtained by other researchers. The most recent work where similar performance measurements were carried out is that of Limberger et al. 2016. It is also the only work that we could find where this issue is treated in the context of software visualisation and with experimental results. The authors run rendering performance assessment on three graphic libraries and compare the loading time under

[31] Using a separate geometry for each node adds significantly extra runtime load, hence the results are pushed to the extreme end of a performance load. In contrast, sharing the geometry between nodes brings significant performance improvements.

different scene authoring techniques that are intended to enhance performance. The one of most interest to this work is their results when using the *shared geometry* technique with the *X3DOM* library. Their test involved 2990 nodes and scored an average loading time of 3.60 seconds[32] (see Figure 5.13). The reader is reminded that the authors are employing here the shared geometry technique (which brings significant performance improvements). In contrast, the loading time achieved in this work scored no more than 1.86 seconds on average, for a total of 22,500 nodes—and is rendered using separate geometries. This is a significantly improved load time for a significantly higher number of nodes, without yet taking advantage of the shared geometry technique. The significance of the improvement obtained in this work become yet more apparent when knowing that X3DOM is perceived by the development community as a much lighter, significantly simpler, and a less resource-consuming library than Three.js[33]. In addition, the authors' work was a web-based implementation (server/client), which is expected to draw lower resources than the IDE-based implementation used in this work. It must be noted however that the 2990 nodes in the authors' work were structured in a parent/leaf hierarchy (tree-map), which undoubtedly adds extra runtime footprint. However, this extra overload is not expected to be drastic. The parent/leaf hierarchical structure is intrinsic to the nature of visualisation technique used in this research and confirmation of this surmise came indeed as true after the research tool was built[34]. For a comparison based on real-world open-source systems, the reader is referred to section 7.4.1 where more comprehensive benchmarks are provided.

Basis	Approach	DOMContent Loaded	First Frame Displayed	Continuous Rendering	Remapping all Colors	Remapping all Heights	Highlighting of one Leaf Node
X3DOM	**Boxes**	0.48s	4.05s	15fps	87ms	152ms	91ms
X3DOM	**Shared Geometry**	1.37s	3.60s	15fps	87ms	124ms	86ms
X3DOM	**Pre-baked Buffer**	0.63s	1.19s	60fps	112ms	552ms	696ms
XML3D	**Mesh**	1.59s	2.05s	30fps	173ms	98ms	37ms
XML3D	**Assets**	2.21s	2.70s	29fps	97ms	104ms	40ms
XML3D	**Pre-baked Buffer**	1.46s	2.78s	28fps	291ms	293ms	37ms
glTF	**Static**	0.24s	0.29s	60fps	–	–	–

Table 1: *All measurements were captured and averaged over* 1.000 *iterations for the same data set of 2990 nodes (358 parent and 2632 leaf nodes) with the same attribute mapping applied. All data was deployed on a server (http://hpicgs.github.io/web3d-treemaps/) and loaded/processed locally in Chrome (50.0.2661.87 64-bit) running on a Notebook (Intel Core-i7 6700HQ, 16GB RAM, Windows, Intel HD 530). For polyfill and DOM publicly available resources were used based on X3DOM 1.7.1, XML3D 5.1.4, and glTF 1.0 (with three.js r76).*

Figure 5.13: Load Time Results of multiple declarative 3D libraries as reported by Limberger et al. in their 2016 work.

The goal of drawing the comparison above is aimed to support the design choices made in this work. The factors discussed are seen as sufficient to accomplish that. However, the comparison could actually

[32] Note that this is considered the best result for their X3DOM tests in terms of practical applicability. This is because the pre-baked buffer method is not suitable for an interactive visualisation (it does not allow object picking).

[33] See community thread: https://www.quora.com/Is-three-js-the-best-javascript-toolkit-for-webgl-and-why.

[34] For example, the load time of 4111 parent/leaf nodes came up to an average of 666 milliseconds on the completed Vscode extension—a significant improvement over the 3.6 seconds in a web-based tool.

be carried further. The scene featured in Limberger et al. work was actually simpler and less resource-hungry than the scene used in this work's performance test. This is because the scene objects used in the authors' work were not full 3D but only 2.5D. This means that not all sides of the geometries were being rendered—a design choice by the authors intended to reduce the load time of the web page. Furthermore, all the nodes were fully animated in the test scene of this work, while no animations were used in the authors work. Lastly, it appears that minimal heights were used for most of the geometries in the authors' test scenes[35], whereas the test scene in this work used a minimum height of 5 units for all nodes—drawing thus even more rendering load. It is hence interesting that better performance was obtained in this test despite featuring more complex 3D scenery with a far higher number of nodes, a heavier graphics library, and a more resource-demanding environment. This could be seen as a testament that effort put in to select the right technological components based on careful research and analysis is likely to give fruitful outcomes.

[35] This impression is based on the featured images, and some of the sample codes provided in the accompanying presentation slides. It remains an assumption only.

5.3.4 Visual Effects Quality

While visual aesthetic qualities are not a functional requirement, it is nonetheless an important requirement for two reasons. The main drive behind software visualisation is to capitalise on the power of our visual senses to scaffold the cognition of a given problem. The more a scene is reminiscent of a real-world environment, the better the chance of the brain quickly picking up on familiar features and drawing similarities with the new problem at hand—hence triggering the process of cognition of the new problem. This is a key principle behind the concept of using a metaphor to elicit and communicate knowledge—regardless of it being a linguistic or a visual metaphor. While not expressed in the same form, support for this line of justification can be found in the work of (Ben-Ari and Yeshno 2006; Siau and Tan 2005; Bedu, Tinh, and Petrillo 2019). Thus, the better visual capabilities made available, the better the chance of creating a good real-world resemblance of the metaphor, and hence stimulating cognition scaffolding more effectively.

Another importance of the visual effects is the role it plays in user satisfaction. Just as creating a good UI is crucial for any application if it is to receive successful adoption by its intended users, so is the aesthetic factor of any successful visualisation. It should be straightforward then to conclude that a visually appealing visualisation is more likely to enjoy better reception by its intended user community than a poorly presented visualisation—and hence higher chance of adoption. More importantly, some certain aesthetic elements can even render the visualisation to be less effective—in practical sense—when not addressed properly—e.g., a bad lighting could make object distinction harder, so is bad shadowing. This can be clearly visible for instance in some of the 3D scenes featured in the work of (Baum et al. 2017) compared to those in the work of (Limberger et al. 2016), which were both discussed earlier. The aesthetic improvements in the latter work had significant impact on the final discernibility and expressiveness of the scene (see Figure 5.3 and Figure 5.4). Further elaboration of the role of aesthetics is beyond the scope of this work, but the subject is well-treated and attended to in the wider field of Information Visualisation—and to some good extent in the specific domain of software visualisation as well. A notable paper of the former is that of (Lau and Moere 2007), while the work of (Beck, Burch, and Diehl 2013) and (Baum 2015) offer a good treatment of aesthetics in the context of software visualisation.

In light of the above, visual capabilities of the candidate graphics library were given due attention. A number of experimentations were carried out to assess the visual features offered by Three.js library to ensure good aesthetic quality can be obtained when producing the visualisation. This primarily focused on lighting, shadowing, material properties, and anti-aliasing quality. Each of these features were tested out in different settings that simulate desired aspects in the to-be-produced visualisation.

Figure 5.14 shows examples of these assessment activities in preliminary stages. Advanced assessments and testing that were carried out in later stages can be further seen in Figure 5.15.

In summary, the visual effects capabilities offered by Three.js were found to be well satisfactory to the needs of this work—even exceeding the expectations in some. More importantly, those features were tested to ensure that they did not add significant rendering overhead. The only feature that was not quite on par with the desired level of visual appeal was the anti-aliasing filter. The prepacked anti-aliasing filter that comes with Three.js was good enough for most objects. However, it did not offer that level of smoothness in geometries that featured some acute edges—which is an attribute of the visualisation technique employed in this work. Fortunately, a custom modified filter was found to be available[36] that significantly improved the anti-aliasing mechanism in Three.js, and was hence adopted in this work.

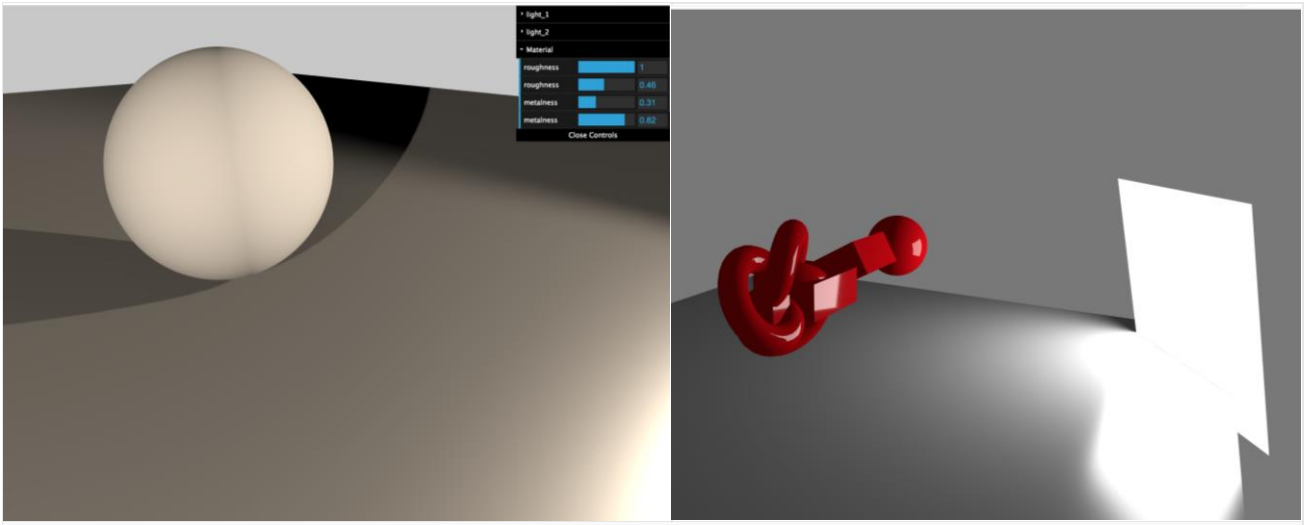


Figure 5.14: Two screenshots showcasing part from the preliminary testing carried out on Three.js graphics library to assess its aesthetic capabilities such as lighting, material, and shadows.

[36] Source: http://alteredqualia.com.

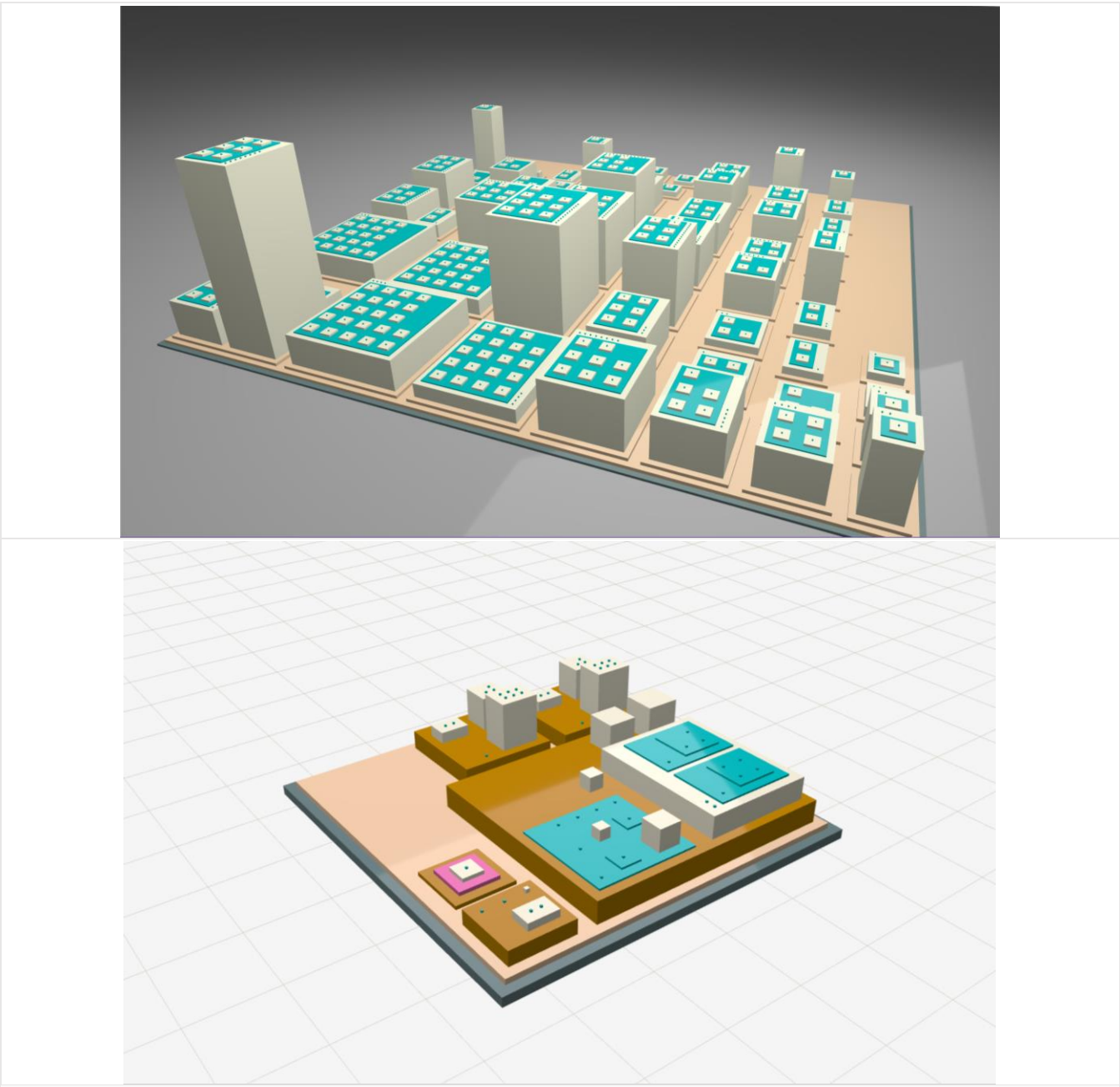


Figure 5.15: Example of other advanced testing carried out in later stages to assess the visual effect capabilities offered by Three.js. These tests are now utilising the same visualisation technique that will be implemented by the tool, and are also running in a development environment (IDE) comparable to a target user environment.

Closing Remark. The details in the preceding sections might seem unnecessarily involved for the scope of this research. However, this level of elaboration is intended and is hoped to help advance the field. The following discussion attempts to explain why.

As pointed out in the motivation part of Chapter 1, a key problem behind the lack of software visualisation adoption by the practicing community and industry is the issue of tool separation and availability. There is a good number of software visualisation tools produced by researchers. However, very few of them were made available to the developers' community—and barely any[37] are known to have made it to industrial adoption. In particular, within the context of software structure visualisation that is based on 3D metaphors, those that were made available have almost exclusively been offered as stand-alone tools. There are very few research tools offering software structure visualisation that are integrated into Eclipse, but these almost exclusively offer 2D visualisations. Most recent example is the SourceMiner Evolution (R. L. Novais et al. 2013; R. Novais, Santos, and Mendonça 2017). Other very old research tools like Creole, Softwarenaut, and X-Ray were also introduced, but these are of no interest to this research context. The reader is referred to our earlier work for broader coverage of these and other early tools (Alshakhouri 2013). Thus, there is strong lack of publicly available software visualisation tools—to begin with—and then these that are available are disconnected from the developer's work environment.

In fact, in the intervening period since we conducted our earlier work, and as far as the extent of this research has revealed, the only IDE-integrated tool[38] that provided 3D visualisation of software structure based on the popular city metaphor is that called Manhattan, which is featured in the 2013 paper of (Lanza et al. 2013), and originally appearing in Francesco Rigotti's 2011 master's thesis. Even when considering 3D software visualisation in its broadest form—i.e., not necessarily based on the city metaphor—only two other tools are found to have integrated their 3D visualisations in the body of a popular IDE (Montaño, Aponte, and Marcus 2009; Sharif et al. 2013).

The point that is being highlighted here is that while the software visualisation technology could offer great advantage to the developer, and its benefits have long been well-established among researchers (see Chapter 2), the technology has not been picked up by its intended users—despite two decades of supporting research. This work argues that a major impediment and cause of this problem is the difficulty and complexity of presenting the technology in a practical manner. As argued earlier, to be of practical use, the technology needs to be made available around the ***developer***, right where they spend

[37] This is a cautious statement—otherwise not a single tool or metaphor is known to have been adopted by the industry.
[38] This does not included proprietary or custom integrated development environments, such as MOOSE and Glamorous Toolkit (see section 5.5).

most of their everyday activities—that is the IDE. This is because the majority of software visualisation tools are specifically addressing the everyday developer as their main user (e.g., to reduce information overload, assist comprehension, and support other software engineering activities). It is hardly practical though—let alone attractive—to expect the everyday developer to adopt a new extra stand-alone tool to aide their development activities[39]. Context switching is undesirable for sure, but even more critical is the problem of codebase synchronisation between different tools.

Unfortunately, there is a problem impeding the integration efforts. The 3D city metaphor (also called Software Maps by some recent researchers) has been demonstrated as very effective and extremely versatile in supporting numerous software activities—earning a rising popularity and credibility among the SV research community (see M. Alshakhouri, Buchan, and MacDonell 2018) for more details. However, integrating this 3D visualisation technique within today's popular IDEs is very challenging. Presenting it yet in a usable and practical form is even more intricate.

Hence, the technical elaborations presented in this section—and the sections to follow—are meant to reflect some of the complexities that need to be navigated. By exposing these details, it is hoped that future researchers would benefit from such information to better advance the field.

Other researchers have already highlighted this issue of separation and calls towards IDE integration can be seen in earlier works (Kuhn, Erni, and Nierstrasz 2010; García et al. 2015). Additionally, in 2013, (Limberger et al. 2013) built a web-based visualisation tool citing specifically the "disconnected" situation as their primary motivation. Rather than focusing on the IDE though, the authors chose to develop their tool as 'web-based' to make it easier to integrate with what they referred to as "(web-based) software engineering tool set". Although they advocate and mention 'connecting' their Software Maps tool to existing collaboration platforms, it was not clear if this was indeed accomplished with a real-world platform. Little discussion was also afforded to how such integration is (or to be) accomplished.

[39] In all of the three phases of evaluation conducted in this research, expert users have consistently recounted their revulsion of adding an extra tool to their daily set of use.

5.4 Agile Dashboards APIs

Prelude. A key aspect behind the motivations of this work is utilising visualisation to assist developers in their daily activities—and more specifically—to support the agile development process. The justifications behind this has been presented in Chapter 1 and elaborate discussion appeared in our earlier work as well. What is important to the discussion at this point is that the research tool aims to synchronise and link the agile user requirements—or, in broad terms, design artefacts—with the visualised codebase. Given the rising popularity of cloud-based dashboards to manage the agile development process, it only makes sense then to access those artefacts directly from their provider. Fortunately, most of today's popular agile software development dashboard products[40] provide—and take pride in—open-access APIs that allow third-party tools to access the data stored on their users' boards. This data is primarily none more than user requirements expressed and structured in different forms as per the specific agile methodology that each of these dashboards were originally built to support. For example, some dashboards are structured around User Stories, others around Issues, and yet others around Cards. This data format structure is then found to be reflected very strongly in the API of that dashboard (almost all dashboard products were designed with a particular agile practice in mind—so even when a dashboard allows the user to structure their boards to suit a different agile practice, that structure is only superficial; under the hood the data is found to be still stored in the same original data structure[41]—e.g., based on stories or issues format).

This recent development of having user requirements stored electronically in structured formats and being available to access by third party tools was essential to be able to realise the concept behind this work. Until recently, this was hardly possible and user requirements—the backbone and main ingredient of the development process—was confined to textual based files (most likely with versioning nightmares). In best circumstances, it would be locked up in proprietary requirement management tools with little hope of third party tools making use of it. This issue has undoubtedly influenced research in the field of requirement engineering in negative ways, hampering efforts in requirement traceability, in particular. Many researchers that needed to work with requirement access in generic fashion had to come up with their own format to structure, store, and access these requirements. This was the case in our earlier work, as well as for many other researchers—e.g., (Rahimi, Goss, and Cleland-Huang 2017; Cleland-Huang and Vierhauser 2018; Lian et al. 2016; Delater and Paech 2013b).

[40] For the sake of brevity, these will be referred to simply as '*agile dashboards*' in the rest of this document.
[41] This was noticed in Jira, Asana, and Trello.

It is unsurprising then to see a recent proliferation of third-party applications racing to tap into these repositories and make creative use of users' data stored on them.

The remainder of this section will discuss some aspects of today's agile dashboards that are relevant to this work. It will also discuss the qualities required to be able to integrate such product into the research tool, and the factors guiding the selection of the specific dashboard product to adopt.

Primary Factors. Availability and open-access to the user requirements is identified as a key factor in a candidate agile dashboard for making the research tool building possible. It is also found to be readily satisfied by most of today's major dashboard products—as the discussion above attempted to make clear. However, of particular importance to this work is the structure of these user requirements. While each dashboard product brings its own format to structure, store, and present these requirements, they all have the common characteristic of using small units (or chunks) of requirements as their fundamental building blocks. This is a natural by-product quality stemming from the intrinsic nature of the Agile development practices, as they all advocate small units of requirements as a basis to their way of working. Looking back at the motivations driving this research, it becomes possible now to see how this characteristic of user requirements' structure aligns particularly well with the goals of this work—which aims to link those pieces of requirements with the specific code items they are associated with. Had requirements been structured instead in larger chunks, their code realisation would most likely result in larger number of code items. However, with smaller requirement units, their realisation is more likely to be confined to limited number of code items—probably a couple of methods or classes. This means the accuracy of linking a small piece of requirement to its couple or limited number of code items is now higher (and hence more effective) than if the piece of requirement was larger. In that case, it would have been more likely to result in (or impact) a larger number of code items, possibly even several source files, instead of smaller and more confined code items. In summary, and as a rule of thumb, the smaller a user requirement unit is, the more confined and limited its implementation is, and consequently, the higher the chance of the created tracelinks having better accuracy and practical efficacy.

In a big picture, the research tool in this work attempts to integrate these agile requirements—also referred to as agile artefacts and design items elsewhere in this work—into the IDE itself to make them readily accessible to the developer. It then leverages on this accessibility to create a synchronised linking between these agile requirements, the specific code items they are eventually translated to, and the visualised representations of those code items.

A fit-all solution. The question that remains now is which agile dashboard to adopt in this work. Fortunately, despite the different formats used by today's popular agile dashboards, they all have a general common theme of a hierarchical breakdown structure. After looking at the APIs of three popular agile dashboards—namely, Jira, Trello, and Asana—it became evident that it is possible to build the tool in a manner that allows it to integrate to all. In broad terms, instead of having the tool's core functionalities interfacing directly to a particular dashboard's API, it could be designed to interface instead to a common intermediary module—named later on as Dashboard Provider (covered in detail in section 6.4). The dashboard provider module can then be provisioned by the specific implementation of a particular dashboard's API without disrupting the core modules of the tool. As long as the implementor module provides well-demarcated small units of requirement artefacts, the other core modules should function correctly because the linking mechanism is created at this lower granular level. The presentation UI for the display and interaction with the agile artefacts can then be implemented separately by each dashboard provider. In fact, thanks to the high modularity of the selected IDE (see section 5.6), there is no need to introduce any new components to work as intermediary module. The described design architecture simply capitalises on the API architecture of the selected IDE, creating an alternative implementation to an existing native API interface for any agile dashboard that is to be integrated. As will be covered elaborately in Chapter 6, UI components can be fulfilled dynamically at runtime by multiple providers. To give an example, a view pane that retrieves and displays the agile artefacts from their cloud-based board, can be programmed to be provisioned dynamically at runtime by the implementor provider of the specific dashboard that the user is currently authenticated to. This gives abundant flexibility to present the artefacts of each dashboard in the best structure that matches the data format adopted by that dashboard—without any impact on the tool's inner working, its overall UI skeleton, and (most importantly) without requiring the building of a separate extension. A diagram of the architecture enabling this design decision, along with elaborated discussion, are presented in sections 6.2.4 and 6.4.

Secondary Factors. Aside from the provision of small units of user requirements artefacts—which is a key factor to enable the building of the tool—there are a few secondary factors that are identified. These are listed below and discussed briefly afterwards.

Other identified factors guiding the selection of an agile dashboard:

- Access Authorisation
- API Nature
- API Maturity and Level of Support

Fortunately, given the relative modernity of most agile dashboards and the fact that they were managed by commercial organisations, all of these three factors were found to be satisfactorily—and competitively—met by all of the three dashboard products introduced earlier. As hinted earlier, the availability of an open-access API to allow third party tools to hook into these dashboards and build services on top of them is seen as a distinguishing prime feature for these competing products. Luckily, this has resulted in their APIs to be well-taken care of and to receive considerable attention. With a quick visit to the Developer's sections of Jira, Trello, or Asana, the quality and maturity level of their APIs stand out immediately. They all offer RESTful-style[42] APIs, supported in multiple prominent languages, and provide high quality user support and documentation[43]. To illustrate this, example code from each product's API is shown in Figure 5.16 to Figure 5.18. Each is found to systematically provide a working example in multiple languages for almost every single functionality that is offered. Jira has additionally a wrapper library in Node.js, thus offering even simpler accessibility to JavaScript developers working on Node.js platform (see Figure 5.19). As for Asana, they offer wrapper libraries for their REST APIs in five major languages. For the purpose of the research tool building, this meant greater flexibility and one less technical complexity to worry about when integrating any of these products when selected.

As for the access authorisation point, all of the three products support the OAuth[44] authorisation framework as their preferred means of granting third-party tools access to their users resources. Other basic forms of authorisation are also supported. For the application of this work, OAuth framework will work perfectly as long as the hosting IDE is able to support URL redirects mechanism. To quickly explain this point, it is helpful to note that there are three parties involved in this kind of authorisation—the agile dashboard provider, the public user owning the board or resource, and the third-party tool trying to access the user's resource. In simple terms, OAuth enables a third-party application to obtain a unique API token from the service provider. When the third-party tool needs a user to grant it permission to access their resource, it directs them to the service's authorisation page attaching along its unique API token. If the user is not authenticated, their service provider will ask them first to authenticate. It will then display a prompt to explicitly ask them if they would like that particular application to access their resource. If access is granted, the service provider will redirect the user to a special URL allocated by the third-party tool, attaching along a secret token identifying that user. That last step, if all worked well, should take the user back to the original third-party tool—in this case the

[42] RESTful-style APIs provide web services over HTTP and are supported by all major programming languages.

[43] As Trello has recently been acquired by the parent company of Jira, Atlassian, they both share the same platform for third-party developer support.

[44] OAuth is a standardised authorisation framework that enables applications to offer access to third party tools to their services over HTTP on behalf of a resource owner. See: https://datatracker.ietf.org/doc/html/rfc6749

IDE hosting the tool. The third-party tool must then attach this secret token in every future request made to access that user's resources on the service provider. Users' secret tokens must thus be safely stored and managed by the third-party tool. While the details of this authorisation process should not normally be of relevance to be elaborated here, it was important to make clear how the ability of the to-be-selected IDE to support URL redirects is crucial for this kind of authorization to work. Complexity around this point was in fact the only major challenge—with regard to the agile dashboard integration aspect—that was encountered during the development of the tool. Other relevant details are covered in due course in section 6.4.1.

The Selected Dashboard. As all of three dashboard products in consideration are now found to be reasonably suitable for the research tool purpose, the choice of which specific dashboard product to adopt will have no fundamental consequences in shaping the architecture and development of the tool. Hence, the selection of which specific dashboard product to integrate was not given much serious analysis—beyond what has been discussed above. Instead, the selection relied next on an ad-hoc assessment of which was perceived to be most commonly used at the time by industry and the agile development community. Jira and Trello came particularly on top of our list, each for its own particular reason. By a quick review of online development communities and platforms, Jira came readily at the top—more or less—among the agile practicing community. As for Trello, while not as highly popular, it still enjoyed a very large user base among agile practitioners. Moreover, Trello has been adopted for few years back then by few software engineering courses offered at the school's department where this research is being conducted. This meant that a good base of student project boards were readily available, which could prove extremely useful to carry out required testing during the development. This last point has eventually lured us into starting with the implementation of Trello. However, given the high degree of similarity between the APIs of the two products, a decision was made to try to build the implementation for both if time permitted.

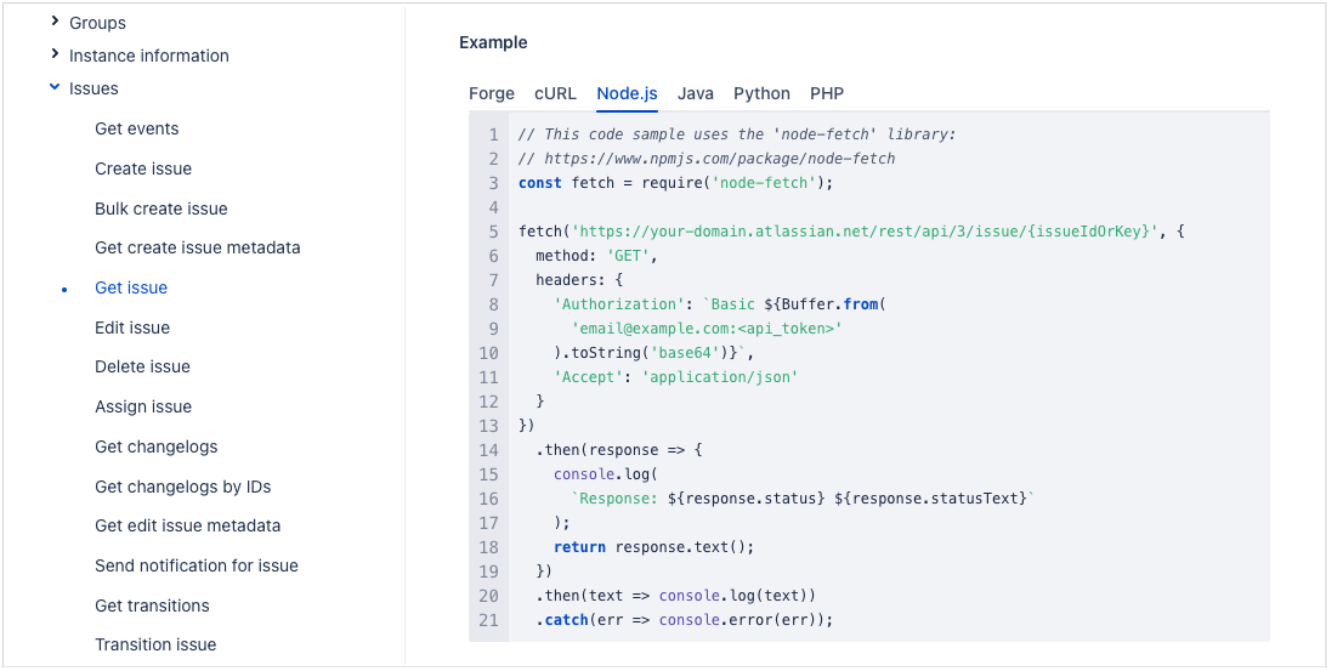


Figure 5.16: Example code in Node.js of Jira's API for retrieving an 'Issue' item. Note the convenience to get code examples in other languages. Source: https://developer.atlassian.com/

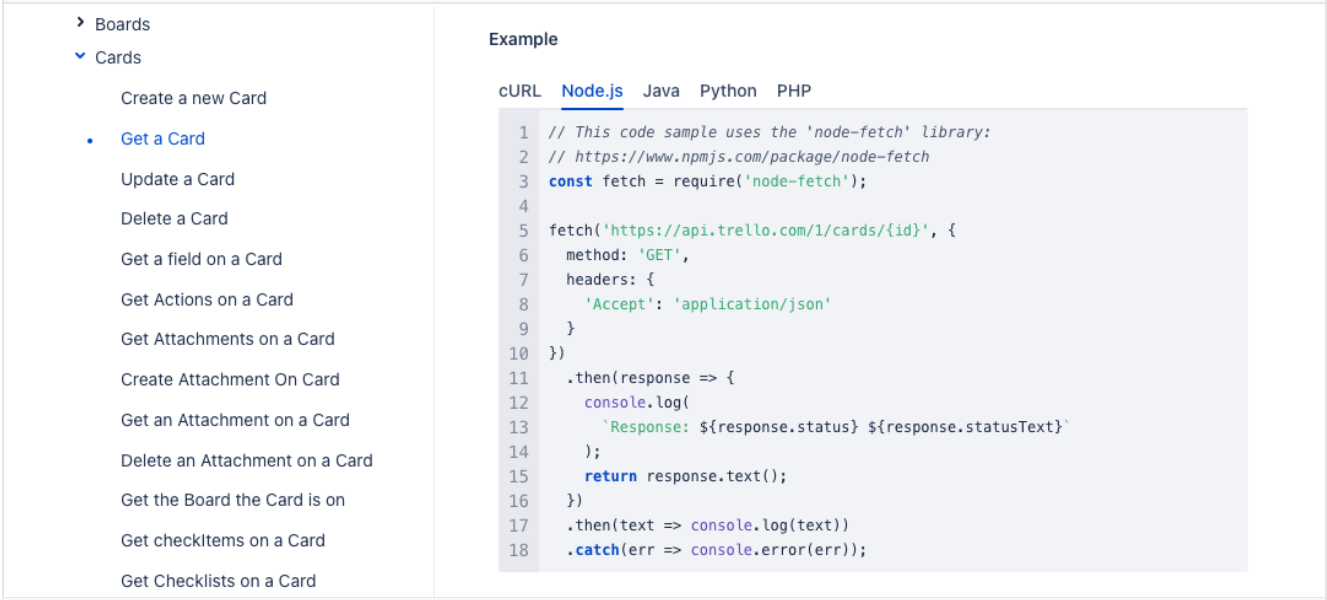


Figure 5.17: Example code in Node.js of Trello's API for retrieving a 'Card' item. Source: https://developer.atlassian.com/

Figure 5.18: Example code in Node.js (using a wrapper) of Asana's API to retrieve a 'Story' item. Source: https://developers.asana.com

```
export async function JiraTest() {
    // Initialize
    let jira = new JiraApi({
        protocol: 'https',
        host: 'agile-insight.atlassian.net',
        username: 'agileinsight.serl@gmail.com',
        password: '[redacted]', /// special generated API token
        apiVersion: '2',
        strictSSL: true
    });

    jira.getAllBoards().then((response) => {
        console.log(`My Jira boards:`, response);
    });

    jira.findIssue('AGT-2')
        .then((issue: any) => {
            console.log(`Issue status name: ${issue.fields.status.name}`);
        })
        .catch((err: any) => {
            console.error(err);
        });

    try {
        const issue = await jira.findIssue('AGT-2');
        console.log(`Issue found: ${issue}`);
    } catch (err) {
        console.error(err);
    }
}
```

Figure 5.19: Example code demonstrating the convenience offered by Jira's wrapper library. Note the direct method calls here (e.g., `findIssue()`) instead of using request URLs as appearing in the example seen in Figure 5.16. This example is from tests carried out as part of experimentation and assessment—hence the basic authorisation method instead of OAuth at this stage.

5.5 Source Code Parsers and Modellers

Prelude. This section presents the journey of the search for a suitable source code parser and modeller to help build the research tool. It describes the challenges encountered, discusses some of the engines and solutions investigated, and—most importantly—serves to communicate the lessons learned.

This research is about visualising source code and working with the building blocks composing that source code—also referred to simply as 'code items' elsewhere in this work. To break down a body of source code into its constituent building blocks, a parser engine is required. When the parser is able to present a model of how these building blocks are structured—an abstract model that exposes how they fit together—then the parser is also called a modeller. In summary, to be able work with the constituting code items of a source code file, a parser and modeller is required to first read and breakdown the syntax of the code, and then build a model of the components—code items—making up the code. Depending on the parser engine, that model could come hierarchical in structure, or flat. The output could also come in various formats; a live object model in some programming language, or an external file based in some defined scheme.

An Ambitious Goal. Creating the tool in a way that makes it able to work with as much programming languages as possible seemed very tempting. Considering one goal of the research is supporting the agile developer, it seemed very narrow to limit the work to a certain language. There is rarely any project today that would not involve a spectrum of programming languages and API stacks. Moreover, a tool confined to a single programming language has certainly a weaker chance of reaching out to the wider development community and successfully advocating any objective—in our case, promoting the benefits of software visualisation. It would also reduce the circle of developers that could eventually participate in evaluating the research tool.

Thus—and to remain realistic given the scope and resources—the research started with a goal to attempt to find a parser and modeller package that supported at least a few of the major programming languages. If that proved infeasible, then the plan was to look for a few independent parsers for key languages that could be integrated into the tool and work in unison. None of these attempts has worked out. However, the efforts still bear some good and interesting findings that is thought to be of value to share. It could be particularly useful for other researchers whose work requires some mechanism of source code parsing. That being said, the search did not end fruitless either. An exciting outcome—with far better potential than initially targeted—has been waiting at the end of the journey. The following discussion presents our findings.

5.5.1. Parser Engines Of Interest

In a similar fashion to the research activities performed earlier to find the best suitable technology to satisfy the other core components of this work, the investigations carried out here followed a parallel approach. The primary criteria desired was first identified and then relevant resources in academia, as well as in industry, were explored in search for a potential match.

As for the primary criteria sought in a potential parser engine, these are: producing the resulting model as a live object, supporting multiple languages if possible, being open-source, and can be tightly integrated as part of the tool. Each is discussed next to explain why it is identified as a criteria.

Live Object Model. This refers to obtaining the result of the parsing process as an accessible object in some programming language as opposed to a textual model streamed out in an external file. As the tool is expected to parse very large number of source files, dealing with streamed or serialised external files will significantly impact the capacity of the tool to handle large codebases. Live JSON objects might be an acceptable middle solution if necessary, but that would call for an assessment to establish its efficiency. A live object model is particularly crucial if the tool is to handle live source code parsing while the developer is actively working on their file—an essential feature for the concept of the research tool.

Supporting multiple languages. A parser is normally able to parse one specific language. However, many products combine multiple parsers to enable them to handle multiple languages seamlessly. This work aimed to find such product where its multi-parsing capability can be reused or consumed as a service. While it is highly desirable and greatly simplifying to the tool building if fulfilled, suitable individual parsers would still be workable if necessary.

Open-Source. As a research tool, it is undesirable to use a commercial component as part of its core architecture. The work is intended to be made available as open-source at some stage, and thus incorporating a commercial component in its design would hamper this goal. Lastly, being open-source means the parser engine can be modified—if needed—to better suit the research tool.

Can be tightly integrated. This refers to the engine being of nature that allows it to be structurally incorporated within the tool. There are many toolsets or products—including some software analysis ones—that require the user to install an array of other 'pre-requisite' components before that product could achieve its functionality. This inter-dependency is found highly undesirable and could thwart or hamper the reachability of the research tool, hence the target here is for an autonomous self-sustained solution.

A. Moose and FAMIX

First there was Moose, then came Glamorous Toolkit. Moose (Nierstrasz, Ducasse, and Girba 2005; Fabry et al. 2014) is an innovative and open-source software and data analysis platform that was the result of years-long research and collaboration led by the Software Composition Group from the University of Bern. It offers an extensive range of features that includes parsing, modelling, visualising, and live exploration of not just source code, but data in general. Its powerful features are best unlocked however using commands and programming rather than a chain of UI actions. Of particular interest to this work, is that Moose incorporates a meta-modelling framework named FAMIX (Ducasse et al. 2011) that can represent source code—or any structured data—in abstract models regardless of their language. The platform is based on a less-known programming language called Pharo (Bergel et al. 2012). Pharo is essentially Smalltalk but greatly augmented with a range of library stacks designed to make it more effective and flexible for interactive modelling, visualizing, metric querying, and other functionalities that are particularly suitable for software analysis activities (See Figure 5.20). Pharo is also the name of its development environment (IDE) that was specifically designed for it. The highly influential CodeCity (Wettel and Lanza 2007a) was in fact built on top of this technology stack. Other relevant discussion about Moose and the FAMIX meta-model appears in our earlier work (Alshakhouri 2013).

> As a side note, Glamorous Toolkit (Girba and Chis 2015) has recently come to the scene rising as the next glamorous phase of Moose and Pharo. It is presented as the next awaited evolution to the traditional IDE, where visualisations play a central "first class citizens"-role in the development process. Further discussion is irrelevant to be deliberated here, however, the toolkit appears to present advanced and feature-rich software development IDE and analysis platform[45]. It also supports a good set of key programming languages.

Having utilised the FAMIX framework in our earlier work, there was a natural inclination to study if the framework could be adapted and reused in this work[46]. Two attractive qualities that were drawing attention to this framework is the extensive set of software metrics offered, and the promised language independent modelling. While our earlier work was limited to Java, in this situation the technology was being investigated if support of other languages is possible. Work was specifically done to examine if the parsing component could be reused—or in worst case, consumed as a service. While the platform is in Smalltalk, two parsers—a Java importer library (jdt2famix) and a C# importer (roslyn2famix)—were made publicly available so users can parse their code and generate FAMIX models[47]. This certainly falls

[45] For more information see: https://gtoolkit.com/

[46] In the previous work, FAMIX was reused as part of another Eclipse plugin. The author of that plugin has used a Java-based representation of the model.

[47] The recent platform, Glamorous Toolkit, does support parsing for three more languages but at the time of writing, only the PHP parser is additionally made publicly available.

short from the range of languages desired to be supported in this work, however, Java and C# seemed a promising good start. The intention was that if these two libraries would make the FAMIX model of the parsed source code available and exposed, then it could be tapped into and reused in a live fashion. This could enable the tool to be built to consume a live object representation of the FAMIX meta model, and consequently achieve language independence—parsers for other languages are expected to come by in the future.

Unfortunately, after careful examination of the jdt2famix library—and some helpful tips received from its author—it was realised that the Java and C# libraries only serve to export a serialised format of the FAMIX model (see Figure 5.21) that is intended to be loaded into the Moose platform. The libraries on their own expose only limited properties of the model. Figure 5.22 shows the FAMIX model as a Java object as seen in the jdt2famix library, which turned out to expose only limited data about the parsed codebase. The full FAMIX model is only exposed at the Moose Platform itself where the actual modelling and metrics happen to be analysed and presented. Figure 5.23 and Figure 5.24 present examples illustrating the feature-rich capabilities achieved after loading the exported model into the Moose platform.

The other option of consuming the FAMIX model directly from the Moose Platform was not feasible and in either case, it would have violated the objective of achieving tight integration within the tool.

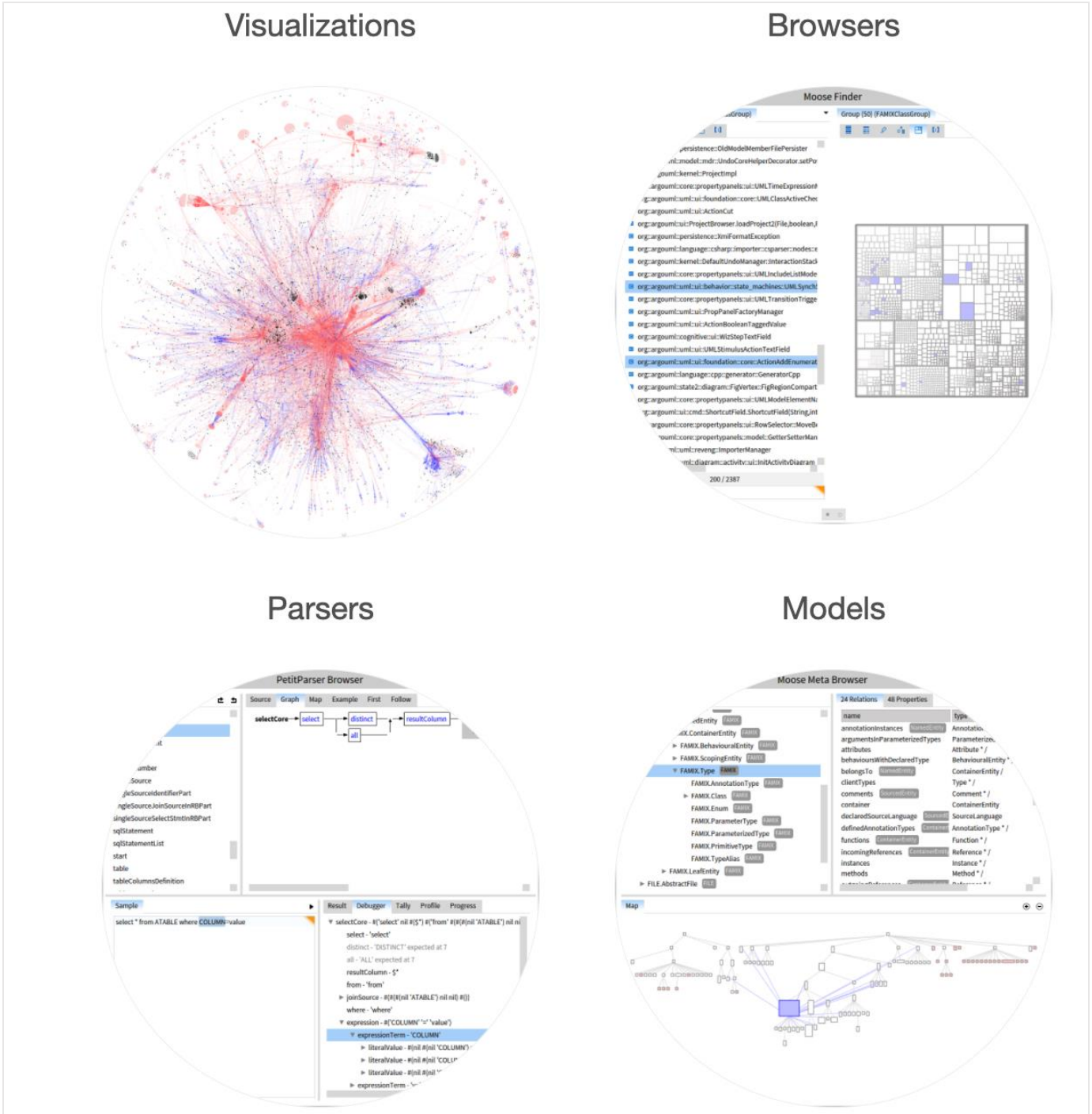


Figure 5.20: Example of Moose Platform applications as featured on their website. Source: https://moosetechnology.org/

ScrumCity v2.0.mse

```
(
    (FAMIX.Invocation (id: 1)
        (astStartPosition 8447)
        (astStopPosition 8462)
        (candidates (ref: 414))
        (receiver (ref: 8529))
        (sender (ref: 2154))
        (signature 'app.getScrController()'))
    (FAMIX.Inheritance (id: 2)
        (subclass (ref: 2151))
        (superclass (ref: 11551)))
    (FAMIX.Method (id: 3)
        (name 'FilterPostProcessor')
        (isStub true)
        (kind 'constructor')
        (modifiers 'public')
        (parentType (ref: 1338))
        (signature 'FilterPostProcessor(com.jme3.asset.AssetManager)'))
    (FAMIX.Invocation (id: 4)
        (astStartPosition 45546)
        (astStopPosition 45552)
        (sender (ref: 3065))
        (signature 'glyph.getEntity().getName()'))
    (FAMIX.Inheritance (id: 5)
        (subclass (ref: 10753))
        (superclass (ref: 11551)))
    (FAMIX.ParameterizableClass (id: 6)
        (name 'ArrayList')
        (container (ref: 8701))
        (isStub true)
        (modifiers 'public'))
    (FAMIX.Invocation (id: 7)
        (astStartPosition 3634)
        (astStopPosition 3642)
        (sender (ref: 6171))
        (signature '((GlyphData)spatial.getUserData("GlyphData")).getEntity()'))
```

Figure 5.21: Serialised Famix model generated by jdt2famix library after parsing a test codebase. ScrumCity—a tool developed in an earlier work—was used as a test codebase.

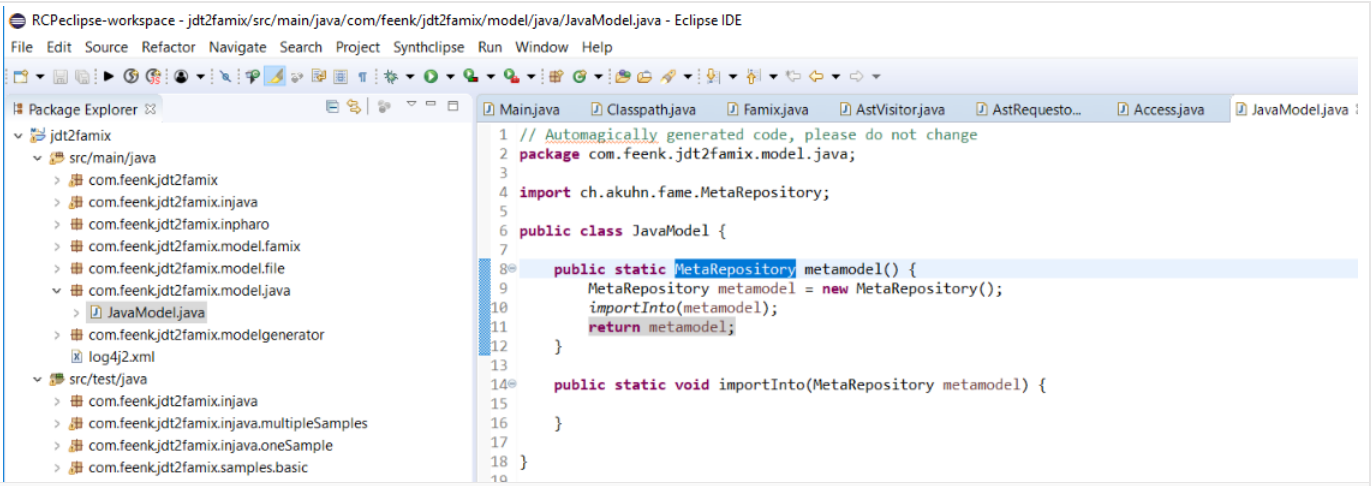


Figure 5.22: Famix meta-model as exposed in the Java library, jdt2famix.

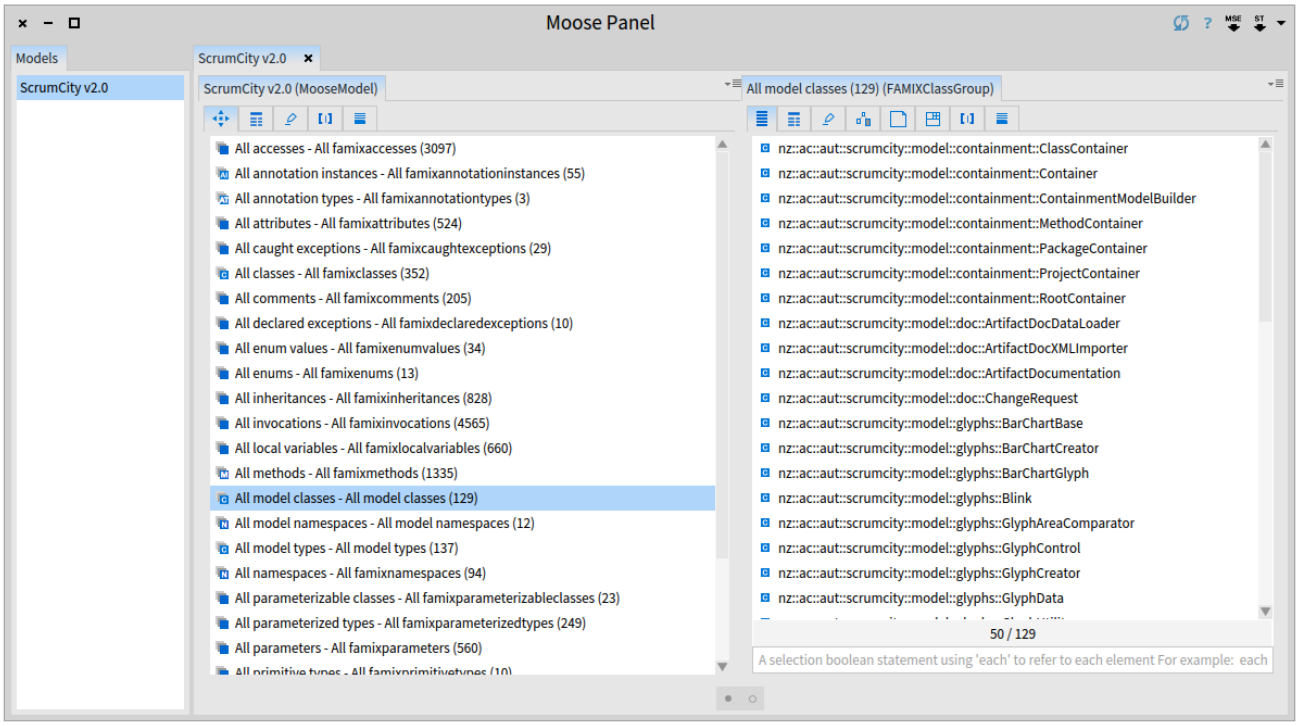


Figure 5.23: Fully exposed FAMIX model objects as seen in the Moose Platform. ScrumCity is used as test codebase.

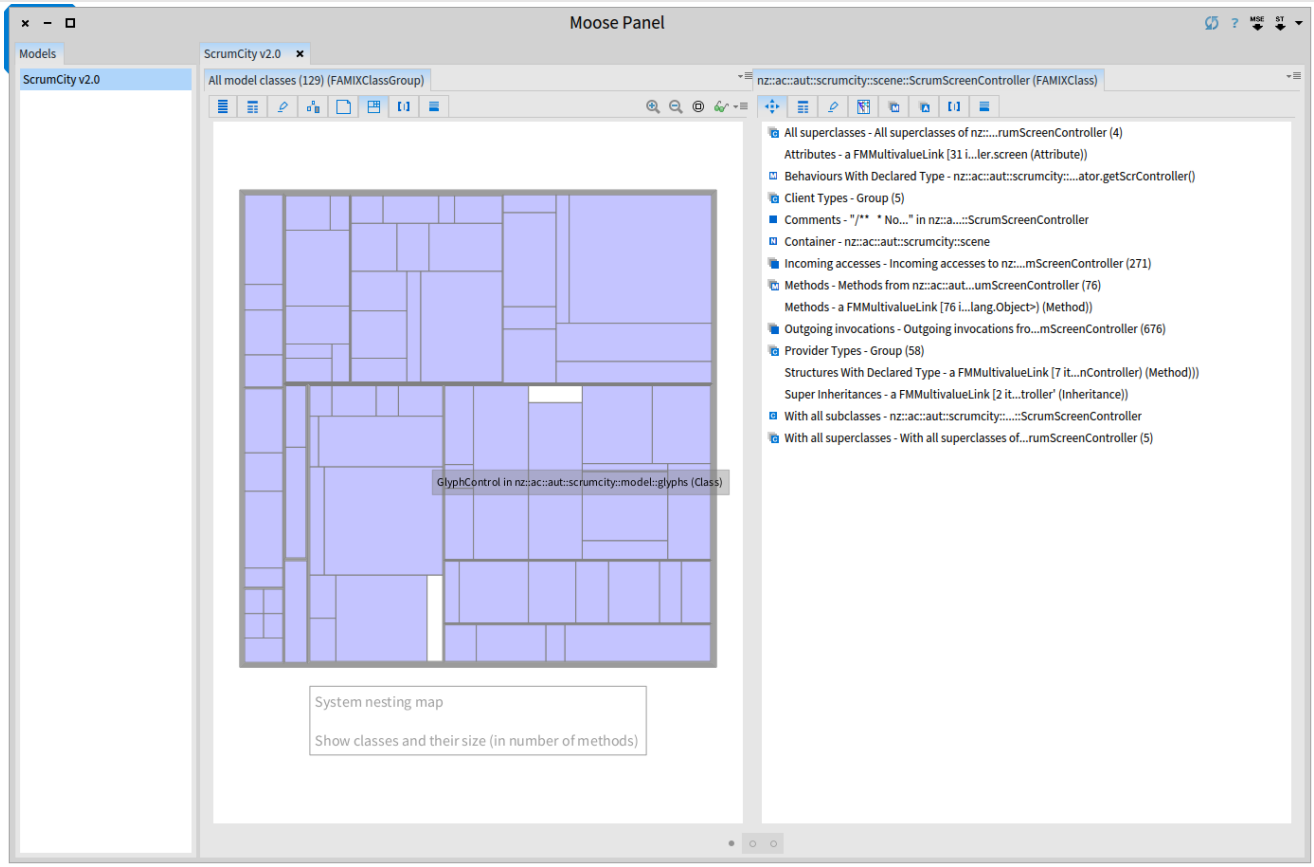


Figure 5.24: Example of some visualisation and analysis offered by Moose Platform as seen applied to ScrumCity codebase

B. SourceTrail

In many disciplines—and particularly so in the software engineering world—advancement is not solely confined to academia. SourceTrail[48] demonstrates such case where valuable resources can be sometimes found outside academic literature, contributed by industry or members of the development community. Driven by the same motives behind software visualisation research, its authors sat to develop a solution to tackle the bothersome mental overload issue associated with navigating and understanding large software systems. Their tool brings a new approach to visually navigate source code using interactive graphical representations (see Figure 5.25). Not to sway from the objective though, its parsing components are the target of interest here. Being open-source and able to handle Java, Python, C and C++, it was seen as a good candidate to be examined to determine if these components could be reused[49].

Findings. Unsurprisingly, the parsing capabilities—in terms of the data and model generated—fall far shorter from that offered by FAMIX-based parsers or the analysers available on Moose. However, the tool could still serve well in exposing the main structural elements of a source code. Rather than a well-integrated product, it unfortunately turned out to rely on an array of dispersed components and plugins from visual studio code to Eclipse and others to achieve its multi-language parsing ability. It uses TCP to share data back and forth between the main application (a separate tool) and the supporting plugins. Specific configuration is also required for each language to be able to parse it. The major drawback however is that it accomplishes the parsing operation in a batch fashion and relies on external SQLite database files to store the data resulting from the parsing operation. This renders it ineffective and completely unsuitable for the purpose of this work as it directly violates the 'Live Object Model' requirement discussed earlier. Trying to readapt the individual parser engines for each language is not a feasible option either.

C. SonarSource

SonarSource is a well-known commercial organisation offering an umbrella of source code quality and analysis products—supporting 27 languages in total. Some of their products[50]—and their underlying parser engines—are open-source too. Being commercially driven, those engines come by unsurprisingly feature-rich, and enjoy high quality of design and development support. Most importantly, they offer

[48] CoatiSoftware. 2020. SourceTrail. https://www.sourcetrail.com/ (recently moved to GitHub: https://github.com/CoatiSoftware/Sourcetrail)

[49] At the time of writing, support for C#, TypeScript, Perl, & Go are also being independently developed by contributing authors. For more see: https://github.com/CoatiSoftware/SourcetrailDB

[50] At the time of writing, the engines for some key languages including Java, C#, JavaScript, TypeScript, and HTML are available as open-source.

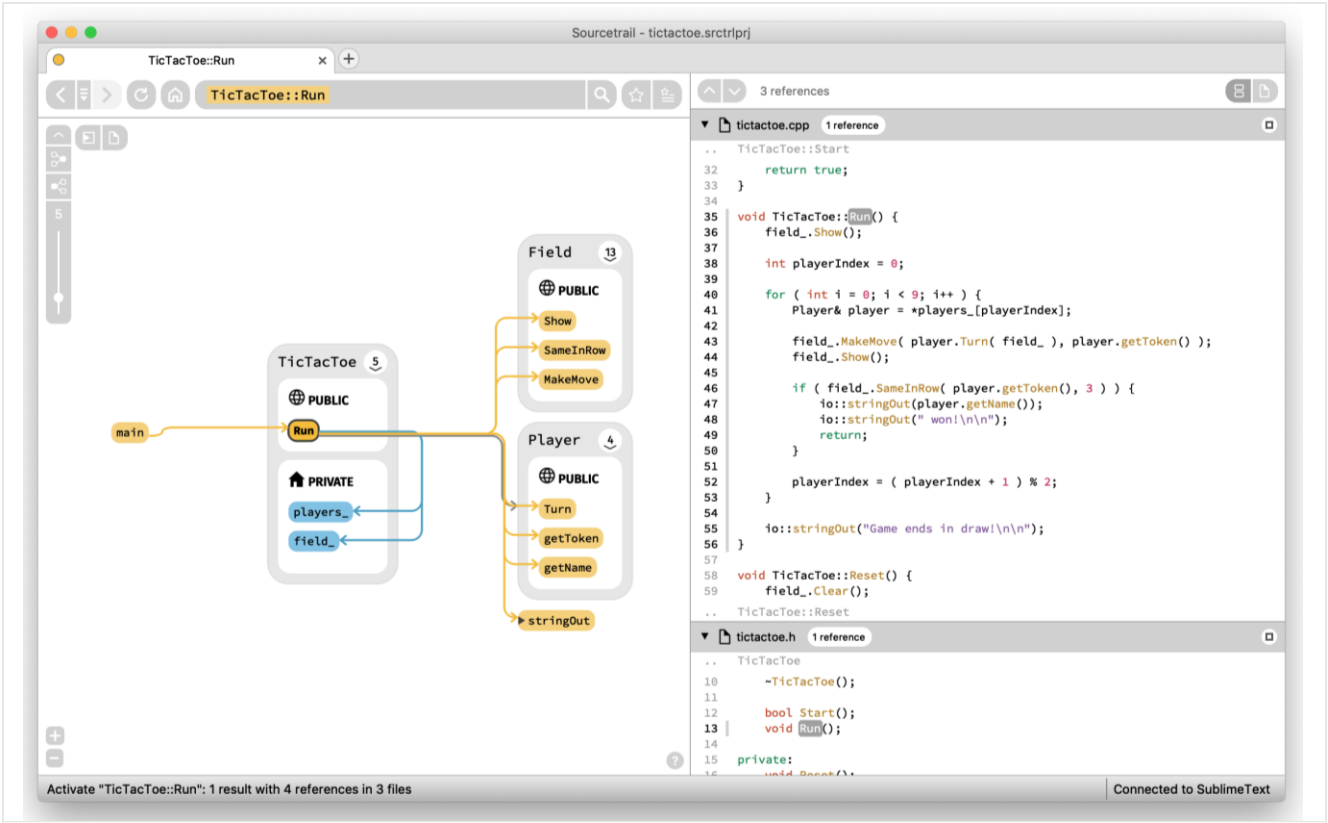


Figure 5.25: Exploring source code using a novel graphical representation as seen in SourceTrail.

on-the-fly live parsing with results being accessible instantly from the API's data objects—that is, they perfectly satisfy our 'live object model' requirement.

The drawback is that all these engines work completely independently with no common model that binds them together. This means that to stay loyal to the original target of building the tool with a language-independent design, each parser engine would need to be re-purposed independently—to interface to a common model that is yet to be created—an effort that is undoubtedly beyond the scope of this work.

D. Spoon

Before closing this section, a particular product that came to light during this exploration activity was **Spoon** (Pawlak et al. 2016). It is an academia-based Java parser and modeller that offers excellent set of features and is found to be particularly suited for the kind of applications considered in this study. It is built on a meta-model—meaning there is a complete abstraction layer—and offers intuitive navigation and querying of the AST (Abstract Syntax Tree) structure. Should research plan falls for a Java-focused solution, then Spoon would probably come first in the candidate list of parsers. While Eclipse's JDT module is widely used in research whenever Java is concerned, it is not the best option

around—especially when source code visualisation is the subject. This is because eclipse's JDT produces a flat structure of the source code, and for treemap-based visualisation applications, a hierarchical structure is typically required, which eventually end up being recreated by the developer. Spoon has also recently been utilised in a number of software visualisation research work (Feist et al. 2016; Seider et al. 2016).

Closing. At this point, the opportunity to find a practical solution for a language independent—or at least a multi-language—capability of parsing and modelling source code seemed to be slipping away. The work appeared to be falling for an undesirable option—proceeding with support of a single language, or two, in the most ambitious case. Java and C# were top candidate languages to consider at this stage. One of the SonarSource open-source engines could be utilised, especially for C#, while Spoon would be an excellent fit for Java.

5.5.2. Parser As A Service—A Language-agnostic Approach

The search for a language-independent approach above has clearly concluded unsuccessfully, leaving what appears as two approaches remaining as the practical ways ahead—either descoping to focus on a single language, or designing a custom intermediary model to achieve independency from the parser. The latter would allow new languages to be supported in future by creating their own interface to the intermediary model. However, it would also significantly undermine the intended reachability of the tool and hamper a key objective of the research—that of promoting software visualisation among the wider community of developers. All other software visualisation research—at least as far as this work could uncover—have, unsurprisingly, fallen to either one of these two approaches. It is by far not known of any work in literature that was able (or claimed) to achieve language-agnostic visualisation of source code. Far from that in fact, it appears that it has always been tightly bound to a single programming language[51] in the vast, if not all, majority of works—with Java being the most targeted out of all.

A solution was potentially evident, but further work was required before it could be realised. With the parser solution still remaining inconclusive, the effort moved next to selecting the IDE that will host the research tool. This step has fortunately led to unravelling a new technology that proved fundamental to realising the research's goal of a completely language-agnostic design for the tool. Instead of trying to pile an array of parser implementations under one roof—and continuously keep developing new parsers for each new roof that might come up in future—Microsoft has recently been working to standardise a Language Server Protocol (LSP) to completely separate IDEs—or any development tool thereof—from a language's implementation and its parser services. Under the concept, the parser becomes akin to a web service (a Language Server to be specific) that is consumed over the LSP protocol[52] by the development tool (or code editor, and the like). This means the parser for any language—represented by its Language Server—will ever need to be developed once, and its services can then be consumed as needed by any development tool to come. Figure 5.26 presents a simplistic diagram to illustrate this separation between the IDEs and language implementations offered by LSP.

It is important to emphasise that the name 'parser' above has been used in a very loose term, solely to simplify the discussion. In reality, a Language Server provides full services for its implemented language ranging from parsing, type checking, and building, to autocompletion, smart actions and inline

[51] This is again by far as the extent of this research has revealed. Works where users have to use a language-specific parser to produce an intermediary FAMIX model that is then loaded into a Moose Platform are excluded from this discussion as they are not live (on-the-fly) approaches and require a separate parser for each language to produce the intermediary FAMIX model.

[52] LSP runs over the communication protocol JSON-RPC.

documentation[53]. Further elaboration about this technology is beyond the scope of this work and the reader is referred instead to the official website of LSP[54].

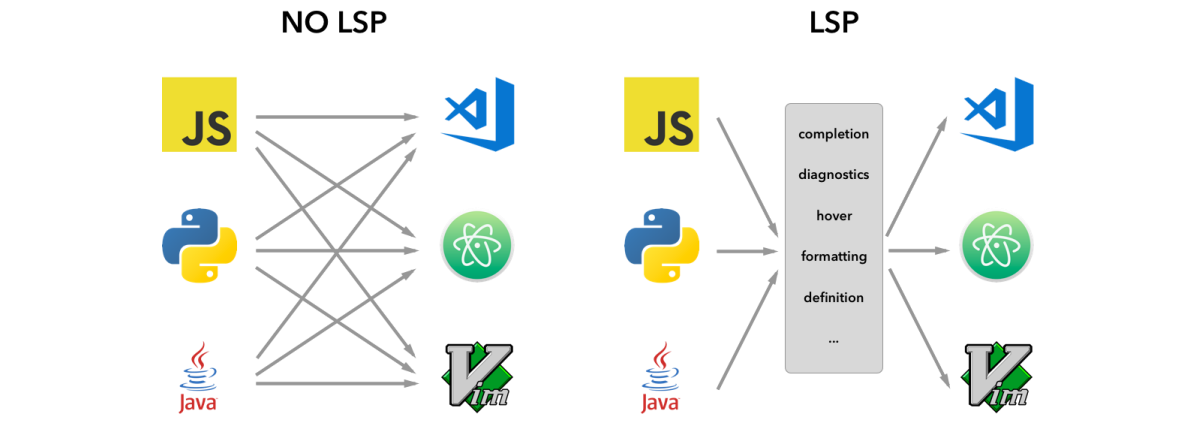


Figure 5.26: A simple diagram illustrating the advantage brough by LSP. Source: https://code.visualstudio.com

Conclusion. The findings above mean that a full language-agnostic approach could finally be realised for the design of the tool. There is no longer a need to be concerned about working with individual language parsers, contriving an artificial intermediary model to conceal the tool behind it, or adopting such pre-existing intermediary model[55]. There is even no need to worry about learning LSP protocol or writing a client side to consume the services of the various Language Servers. The new and open-source Visual Studio Code (Vscode), is designed from the ground up on the LSP framework, and readily comes with implemented clients for tens of programming language servers[56]. In fact, the list of programming languages that have so far had their LSP server implemented—many by leading software corporations such as Apple and IBM—is too long to be presented here.

What the above discussion leads to is that as an extension of Vscode, the tool could—in principle—take advantage of its hosting IDE's API to tap into the extant language-service clients' implementations. If it works, the tool should be able to access the structural model of any source code file that Vscode can currently support—which are in the count of tens—and of all those that it will in future. To establish weather this is possible in practice will involve however a deeper examination of Vscode's Extension APIs—and, as it turned out, parts of its source code implementation too. This particular issue is treated in the next chapter.

[53] While a Language Server could in principle be located anywhere over the internet, for efficiency reasons most current implementations will host it under the same environment where the IDE is located—with some going for a mix-up implementation.
[54] See https://microsoft.github.io/language-server-protocol/
[55] For example, the FAMIX meta model, as was contemplated in the research's early stages as part of potential candidate solutions.
[56] These come as extensions so users can install only the languages they require

5.6 Selection of Hosting IDE—Justifications and Challenges

When it comes to developing extensions (or plugins) for development environments, the Eclipse platform probably comes by large as the dominant IDE of choice. This appears to be particularly so for extensions developed as part of academic research[57]. The platform has a very mature and well-established base of library stacks that are used and supported by a large community of developers, as well as by leading software companies.

Eclipse was in fact the first candidate IDE on the list at the early stages of this work. However, as the various requirements were slowly investigated and assessed, and research came to progress with regard to exploring some latest technology, Eclipse has quickly retreated to the back of the list.

By now, a number of points and technology prerequisites around the fitness of a potential IDE have already been covered as part of the earlier discussions of the other requirements—as these were interrelated with the IDE. The discussions should have also made strong impressions on which IDE aligned better with the goals of this research. As a quick recall, assessments around each of: the seamless integrate-ability of a web canvas, rendering performance of IDE-embedded 3D scenes, support for better and direct event communication between the web canvas and that of the IDE, stability and capacity to handle very large 3D scenes, as well as the opportunity for a language-agnostic design, have all been strongly pointing to Vscode being an excellent candidate for the purpose. As for the last point in particular, it appears to be even crucial for making the concept come to reality.

In addition to the above interrelated factors—which are all pertaining to the functionality aspect—there is also a non-functional quality that is highly significant and influential. Bringing the technology to the everyday developer, right where they perform most of their development activities, was espoused repeatedly as a major goal of this work. Clearly, to achieve best exposure to the targeted audience, the implementation must adopt a highly popular IDE. While the earlier criteria have all unequivocally favored Vscode against Eclipse, a final call came from StackOverflow's Developer's Survey Results. In their 2019 report[58], Vscode has amassed 50.7% of participants' voice, coming as the most popular IDE among developers across almost all sectors. It remained the dominant IDE of choice for the next three years, seizing 71.06% of all IDE usage base in 2021 among professional developers. In comparison, Eclipse's share of popularity scored 14.4% in 2019 and 15.1% in 2021. Figure 5.27 shows the complete results of popular IDEs for the year 2021. Unsurprisingly, during the evaluation phases of this work,

[57] Both statements about Eclipse dominance are mere inferences based on experience and examined literature, respectively.

[58] The survey of that year involved nearly 90,000 respondents, the majority of which identified as professional developers. Source: https://insights.stackoverflow.com/survey/2019

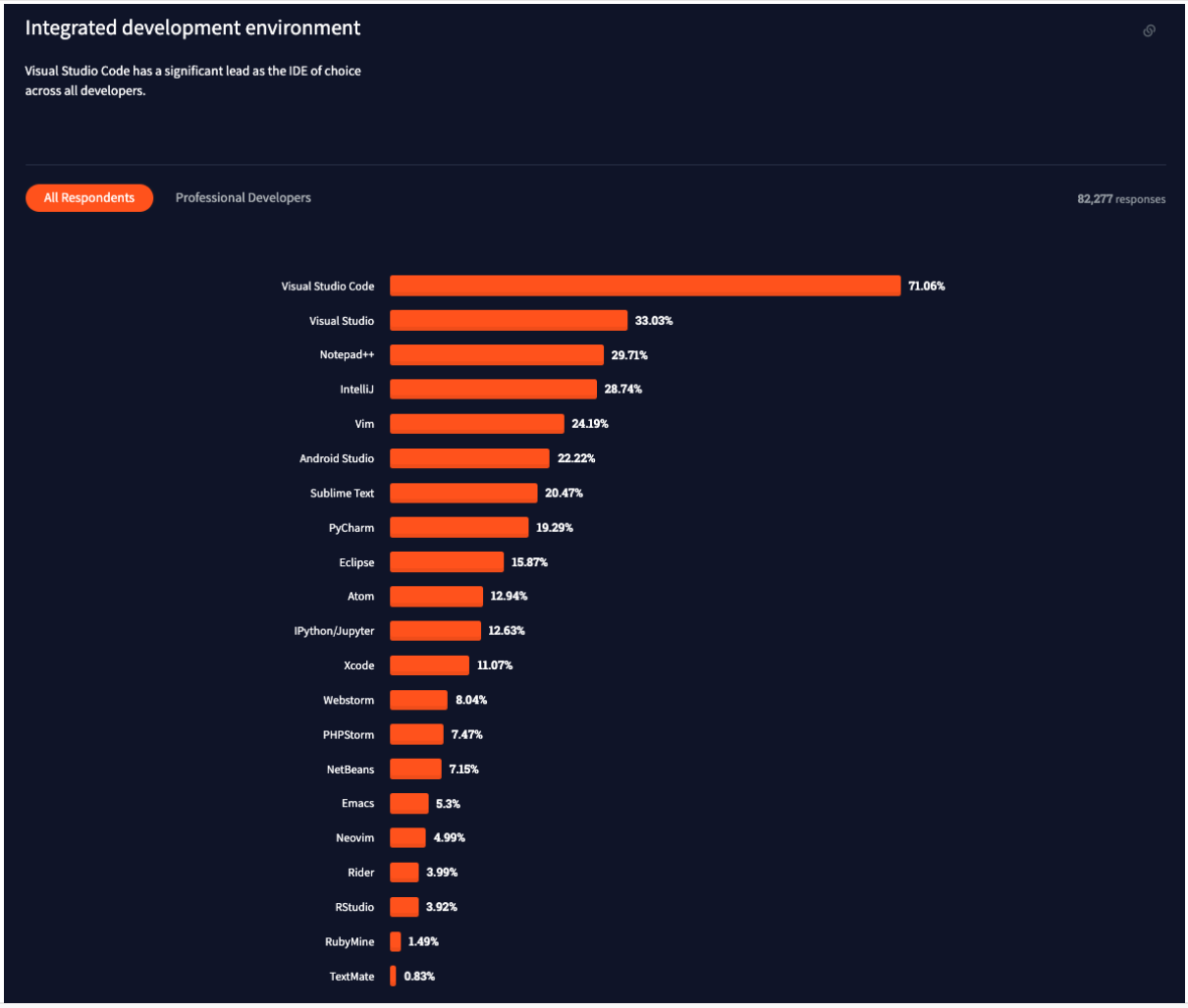


Figure 5.27: Ranking of popular IDEs among software practitioners as revealed by StackOverflow 2021 Developer Survey.

Vscode also came out prominently among the participants. In consideration of all the results obtained and discussed throughout this chapter, Vscode comes strongly and incomparably as the best fit environment to host the research tool and achieve its objectives.

However, while Vscode has proven to be instrumental in realising—and simplifying—many parts of the development, it did not come without its own challenges. Being a very recent environment, Vscode was still undergoing significant development. Many of its features were still on the drawing board or on scheduled releases. This was more true for those features related to the Extension Development API than those addressed to the public user use. The extension development side of the platform did have an excellent base of documentation and support, and a good number of 'Getting Started' examples. However, due to the rate of active development undergoing, the base was clearly struggling to keep up to date with the changes. In some circumstances, some API elements would have outdated naming, wrong documentation, wrong return types, or be completely missing from the documentation. For casual extension development cases, a developer is less likely to face these complexities. However,

once the development starts to require advanced use of the extension APIs, such intricate complications begin to rise—in multiple occasions requiring long hours to resolve, and frequent visits to review the open-source code on its repository—sometimes, to only uncover simple but undocumented end points of the API.

Nonetheless, in the light of enabling features and technology described above, Vscode unmistakeably offered excellent infrastructure to build the tool in a way that best fit the research goals. Moreover, its extension API, despite its recency, is still feature-rich and offers great flexibility and simplicity compared to that of Eclipse. Lastly, given the unrivalled growing base of users, it certainly provides a promising prospective for promoting the benefits of software visualisation.

Table 5.3 presents a summary of the factors influencing the selection of the hosting IDE. These were broadly treated in the earlier sections—section 3, in particular—except for the last three which have been covered in this section.

Table 5.3: Summary of Factors Influencing the Selection of the Hosting IDE.

Factor	Description
Seamless Integrate-ability	This refers to integrating a web canvas where 3D scene is displayed.
Rendering Performance	The rendering performance of the embedded web canvas when displaying large and complex 3D scenes
Stability & Responsivity	Stability of the IDE itself and responsivity of its UI when a large 3D scene is rendered in its embedded web canvas
Event Communication	Two way communication of events between the web canvas and the IDE's UI, in order to achieve natural user interactivity
Language Services Support	This particularly refers to offering programming language services such as parsing, syntax features, and so on.
User Base	The popularity of the IDE among professional developers
API Features & Flexibility	Features and ease of use offered by the API.

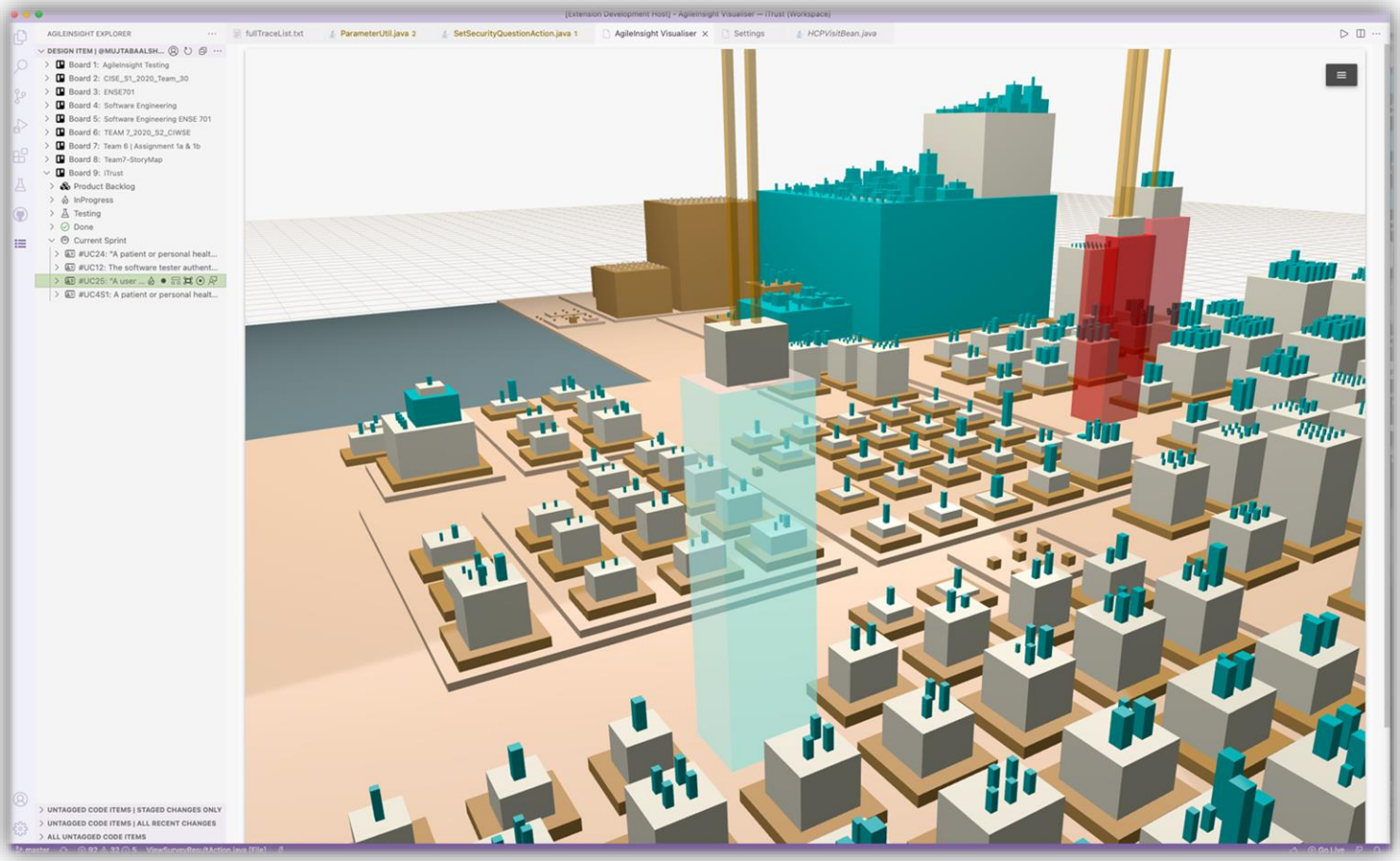
Closing Remark. In addition to the qualities of Vscode described above, there is a foundational quality that sets it apart from traditional IDEs and that was particularly helpful in simplifying the implementation. This relates to the cross platform framework that it is built on. The recent advent of cross platform desktop applications brought the advantage of what used to be web-specific technologies, to be now openly available to standalone desktop applications. In particular, GitHub has

lately used an amalgamation of Node.js and Chromium to create the framework Electron[59] that is now heavily used by industry to create cross platform desktop applications. It uses Node.js as a backend processing engine—hence can run server-side scripts—and uses Chromium to display the frontend GUI—hence can render all HTML and other user-facing web technology stacks. Except there is now of course no backend or frontend (in the traditional sense of the term) as everything is bundled in a local standalone environment.

As a modern and open-source rehaul of Microsoft's comprehensive Visual Studio platform, Vscode has been created from the ground up based on the Electron framework. This opened up a great opportunity to utilise a vast number of development packages and libraries that—until recently—used to be confined solely to the realm of web applications. For the purpose of this research, this opportunity formed a fundamental basis to bring together the required technologies that are essential to realise the research concept. It allowed the runtime environment, the Three.js 3D graphics library, the APIs of the development dashboards, and the language-agnostic design based on LSP, to all come to work alongside each other—natively—without any bridging tiers being required. It enabled the amalgamation of a technology stack based entirely on JavaScript to form the application development and runtime[60]. To put things in practical terms, the visualisation is created using JavaScript, the source code file of any language is structurally accessed and modelled in JavaScript, the user requirements on the agile dashboard—and all related development process information—are retrieved and manipulated in JavaScript; and finally, the tool itself and the environment where everything runs are all written in JavaScript.

[59] https://www.electronjs.org/

[60] Vscode and its extension APIs are written in TypeScript which is then transpiled into JavaScript. TypeScript is a strongly typed dialect of JavaScript with richer Object Oriented capabilities.



Chapter 6
PRACTICAL WORK: TOOL DEVELOPMENT

6.1 Prelude

The previous chapter described the key technological components that are essential to building a functional and usable research tool. It elaborated on the complexities and challenges around each of those components. It especially pointed out how these difficulties have shaped or influenced research in software visualisation. Overall, it served to communicate how decisions were navigated while exploring those technologies, and how best-fit components were eventually selected, in order to meet the research objectives.

In particular, the previous chapter concluded with the argument that implementing software visualisation solutions is complex, that it is more so to integrate into common development environments, and that it is even more so to make the implementation accessible, practical, and available to the everyday developer.

This chapter covers the technicalities of tackling and overcoming those challenges, and describes how those components were brought together to build an autonomous tool—one that is readily accessible to the everyday developer. It presents, as well, the technical details of implementing the key modules of the tool, and how they come together to fulfil the core concepts of this work and deliver a functioning product. As stated earlier, communicating technical details of how key challenges were solved does not only serve to satisfy and adhere to the research's design science methodology, but is seen as a necessity to make knowledge available to future researchers and help advance the field.

Naming is always an important aspect of any product, often carrying a certain message or significance. In this regard, the name AgileInsight was given to the tool developed in this research. It serves to capture its close association with agile practices and to its core objective—providing insightful knowledge to support the development process.

Key Features. Equally important to the overall objectives above is that, through the discussions in this chapter, the key distinguishing characteristics of this work—compared to existing research—come to be exposed and addressed in detail. This includes; seamless integration into a popular IDE, fast and responsive 3D visualisation embedded into the IDE, live parsing of source code files with zero intermediary components, integration with agile dashboards, integration with version control repositories, and lastly, the language-agnostic nature of the solution to help make the tool useful to all developers. The discussion reveals how all of that is achieved in a fully autonomous way without relying on any external components, and with zero configuration being required to run the tool. Once published, any Vscode developer can install the extension from Vscode's Extension Market and start

using it right away. However—and in a conceptual and pragmatic sense—probably a key difference that AgileInsight brings is that it makes the software visualisation benefits readily accessible and available to the everyday developer right where they need it most—in a popular development environment—rather than in a separate tool or custom IDE[61]. More importantly, instead of bringing an idle visualisation functionality, AgileInsight employs the visualisation to solve real problems faced by the everyday developer, assisting them with commonly arising tasks around requirement traceability—a persistent and challenging issue that has been continuously investigated in software engineering research. Further, solutions to these problems are devised based on close collaboration with developers, and guided by their feedback and input. In summary, a key conceptual feature of this work is that it fuses the visualisation into the way of working of developers—particularly those practicing agile methodologies—hence, opening up its benefits to the development process in a practical manner.

The following sections are dedicated to discussing each of the aforementioned aspects in their due detail. An overall architectural picture is presented first in section 6.2. Building of the Webview module and the 3D canvas are then discussed next in section 6.3. Section 6.4 covers then the integration of agile development dashboards through dynamic view panes and OAuth authorisation framework. Section 6.5 presents the technical details of how language-agnostic design was accomplished through novel utilisation of Vscode's LSP feature. Section 6.6 treats the intricate issue of linking source code items with their corresponding design items—that is, user requirements as represented by their agile artefacts. It describes how collected tracelinks are used to achieve bidirectional traceability and how that is then offered to the developer in useful functionalities that fuse into their way of working. Section 6.7 presents how the tracelink knowledge is actively collected from the developer using a novel technique during the development process, with minimum disruption to the developer—an approach that could potentially hold a promising result for an effective and accurate traceability mechanism. The integration of version control systems and the role it plays in achieving permanent traceability is covered in section 6.8. The creation of the visualisation scenery to expose the structure of source code is finally covered in section 6.9. Section 6.10 concludes the chapter with a summarising review and highlights of planned future work. Due to breadth of this chapter, only limited features of the tool are covered here where necessary. A demonstration is available, however, at the accompanying profile page of the research tool: https://blaiski.github.io/agileInsight.page/.

[61] Many of existing software visualising tools are offered as separate tools, or in custom IDEs (e.g., Moose and Glamorous Toolkit are excellent versatile platforms that offer visualisation, but come as part of a custom IDE).

6.2 Design and Architecture

In the previous chapter, Figure 5.1 delivered an abstract view of the key components that shape this work. The diagram in Figure 6.1 presents now a detailed illustration to reveal the actual architecture of the research tool. It shows the designed modules and components, and how they interplay to achieve the overall functionalities of the tool.

This section briefly introduces each component with a high-level explanation. It describes the role of each, discuss its main interactions with the other components, and describe how it is placed in the overall design. Broader details of each part are then provided in the individual sections that follow next.

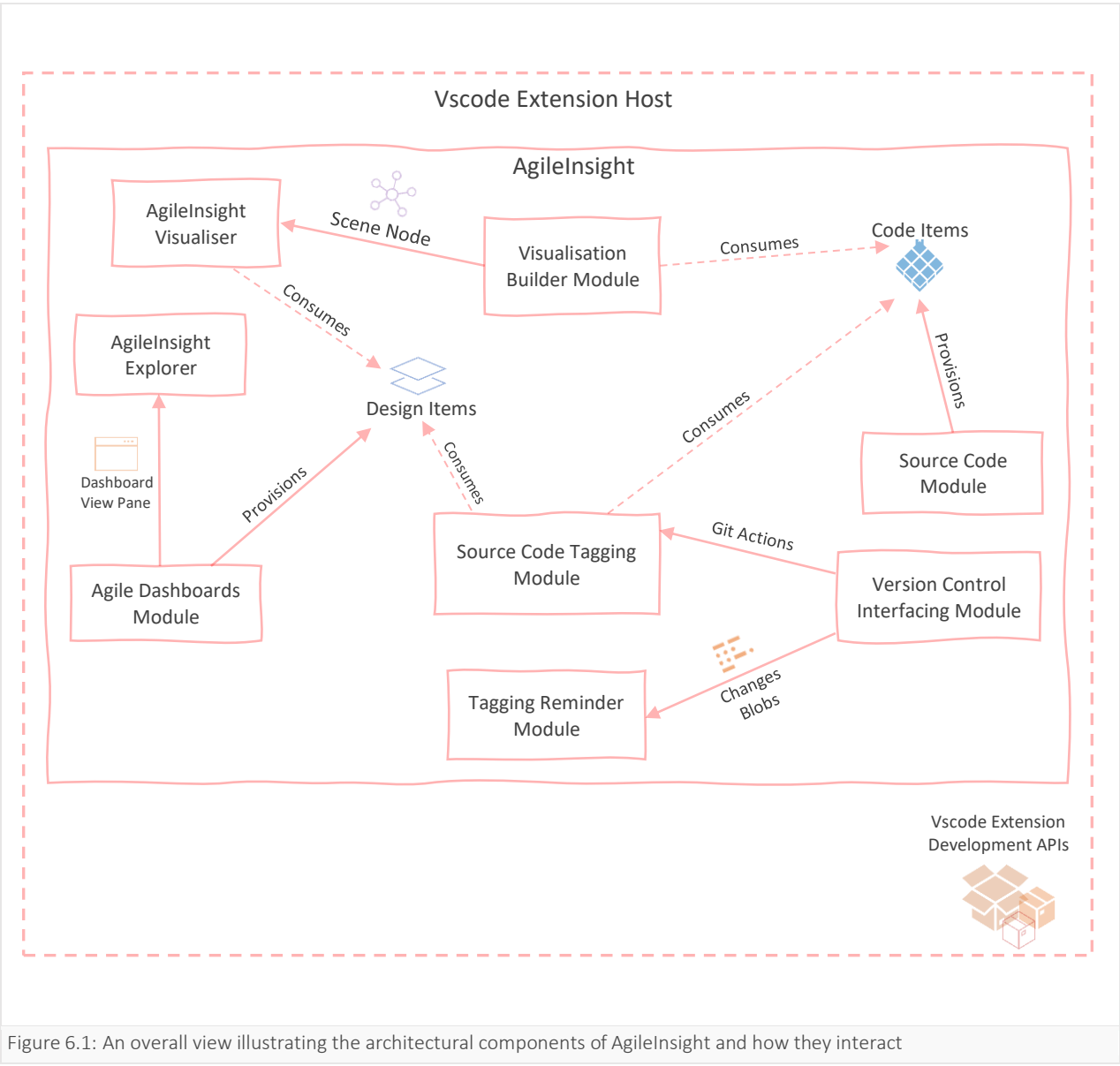


Figure 6.1: An overall view illustrating the architectural components of AgileInsight and how they interact

6.2.1 Vscode Extension Host

This is the part of the Vscode environment where extensions are run. It is a Node.js environment and thus runs in JavaScript. It allows server side web technology stacks to run locally on a user's machine (with the support of the Electron framework as introduced earlier)[62]. This is where AgileInsight is physically situated within the Vscode anatomy.

- Vscode Extension APIs

These constitute the collection of APIs that Vscode makes available for developers to build extensions that run natively within the Vscode environment. Many of them are publicly exposed versions of program interface endpoints that constitute internal modules and packages used to build Vscode itself. In Vscode's vocabulary, these are called 'Contribution Points'. This characteristic is a source of Vscode's high modularity where almost all native user-facing components are built using the same APIs that are made available for extension development. This allows extensions to dynamically 'contribute' to these components to extend Vscode's functionalities with a high native-like outcome.

Using these APIs, the research tool and its primary modules are built and through them they communicate. These modules are presented next.

6.2.2 AgileInsight Visualiser

The visualiser module is one of the key user-facing components of AgileInsight. It is mainly composed of a Webview panel where visualisations of codebase items are eventually rendered and displayed natively within Vscode. Users can navigate the visual scene and interact with its entities just as they would do with other native UI of Vscode. The AgileInsight visualiser is made possible by the following components, which are each discussed briefly next.

- The Webview Canvas

This part represents the visible area of the Webview and is rendered using a custom HTML content provided by the implementing extension. In this particular implementation, it occupies the large space area that is typically occupied by an open-source code editor view. A helpful illustration appears in Figure 6.8 that is introduced shortly in section 6.3. As stated above, this area is used to display the visualisation scene and allow users to interact with it.

[62] Vscode offers other runtime options, including web and remote. In the remote option, extensions can run from a Node.js environment located in a container or a remote location. In the web option, extensions will run from a 'Browser WebWorker' runtime.

Webviews are a special type of extension component of Vscode. Unlike other contribution points and APIs that allow users to introduce new functionalities to Vscode, Webviews offer developers a very high level of freedom allowing them to create custom UI parts and render almost any HTML content[63]. With such powerful capability, security measures must be given utmost care so not to undermine integrity of the platform. Vscode achieves this by accommodating Webviews in a sandboxed context that is separate from the other normal extensions. It further encourages the use of Content Security Policies (CSP) to make a webview as restrictive as possible—allowing just the bare minimum features to run the required functionality.

- Three.js Library

The Three.js JavaScript graphics library (see section 5.3.1) plays a central role here to generate the 3D visualisation scene. Thanks to the Node.js environment and to Webviews flexibility, Three.js is able to render and display the visualisation as part of the WebView's custom HTML content mentioned above. With the right security configurations in place—that is, at extension development level, and not on the user's part—Vscode is able to allow Three.js to run safely within its sandboxed environment, and allow two-way interaction and data exchange between this environment and the rest of the platform's UIs.

- Google's Materialize Library

Materialize is an open-source UI framework for web applications that was originally designed by Google and is based on CSS, JavaScript, HTML. While Three.js takes care of the visualisation, the extension uses the Materialize Library to create necessary UI elements within the canvas itself, making it possible to offer contextual information and contextual user actions when necessary.

- Vscode's Special Webview API

Since the Webview canvas is sandboxed from the rest of Vscode's extension environment, some mechanism is required to allow for two-way interaction between the two. Vscode accomplishes this via a special set of its extension APIs that is designed to facilitate this two-way communication without impacting responsivity. The technique is based on posting messages between the two sides and allow for any data that is JSON-serializable to be exchanged without compromising the security of the platform. For AgileInsight, this practically enables it to achieve interactivity between the web canvas and the rest of Vscode's UI, permitting the communication of any events or data that is required. This API is referred to as Webview's Special API in the remainder of this thesis to distinguish it from the normal API of Vscode Extension Development.

[63] https://code.visualstudio.com/

- Scene Interaction Module

This module implements a number of interactivity features, as well as some animation functionalities related to the 3D scene objects. It also leverages the Webview API mentioned above to implement the required communications between the 3D scene world and the rest of Vscode UIs. The module plays especially a major role in enabling many of the traceability features—particularly those that involve revealing visual representations of the tracelinks. Further details of implementing this interaction mechanism are presented in section 6.3.

Figure 6.2 presents a summarised picture of AgileInsight's Visualiser components and their interaction.

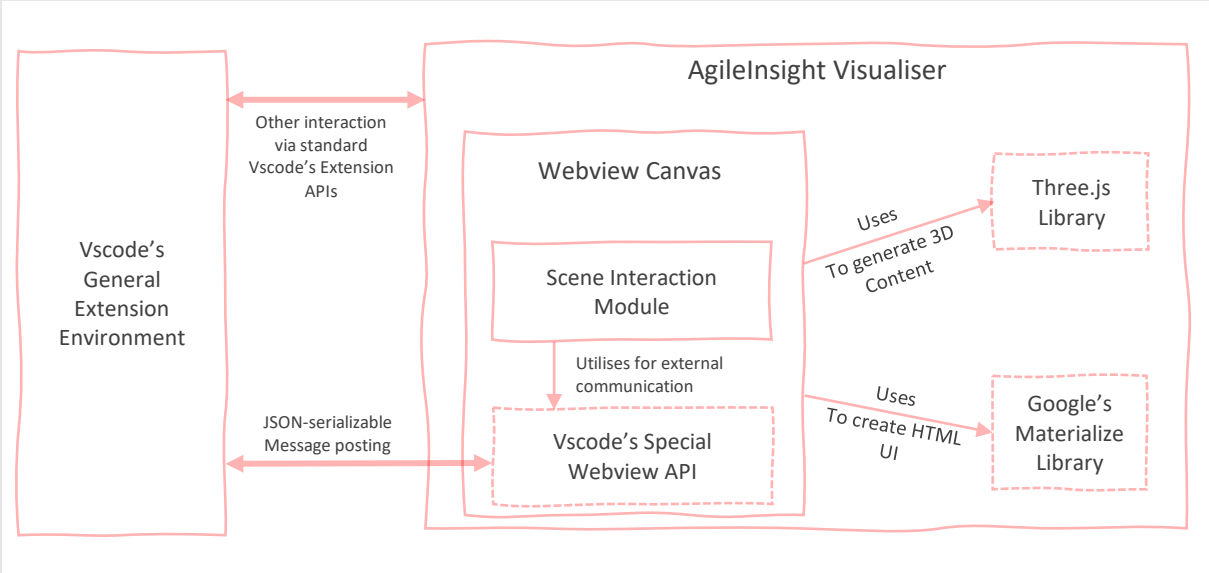


Figure 6.2: An overall view summarising the key components of AgileInsight Visualiser and their interaction

6.2.3 AgileInsight Explorer

Besides the Webview Canvas, this module stands as another key user-facing component of AgileInsight. It contributes a new UI panel view, named AgileInsight Explorer, that is accessible via the side activity bar, where other primary Views of Vscode are similarly situated. In Vscode's vocabulary, this UI module is called 'Views Container'. It serves to host a number of other 'Views' contributed by AgileInsight that are loaded dynamically at runtime based on user actions or current activity. The views are used to display itemised information, which are typically interactive and offer contextual actions to the user. Figure 6.3 shows this UI module with a number of Views actively loaded.

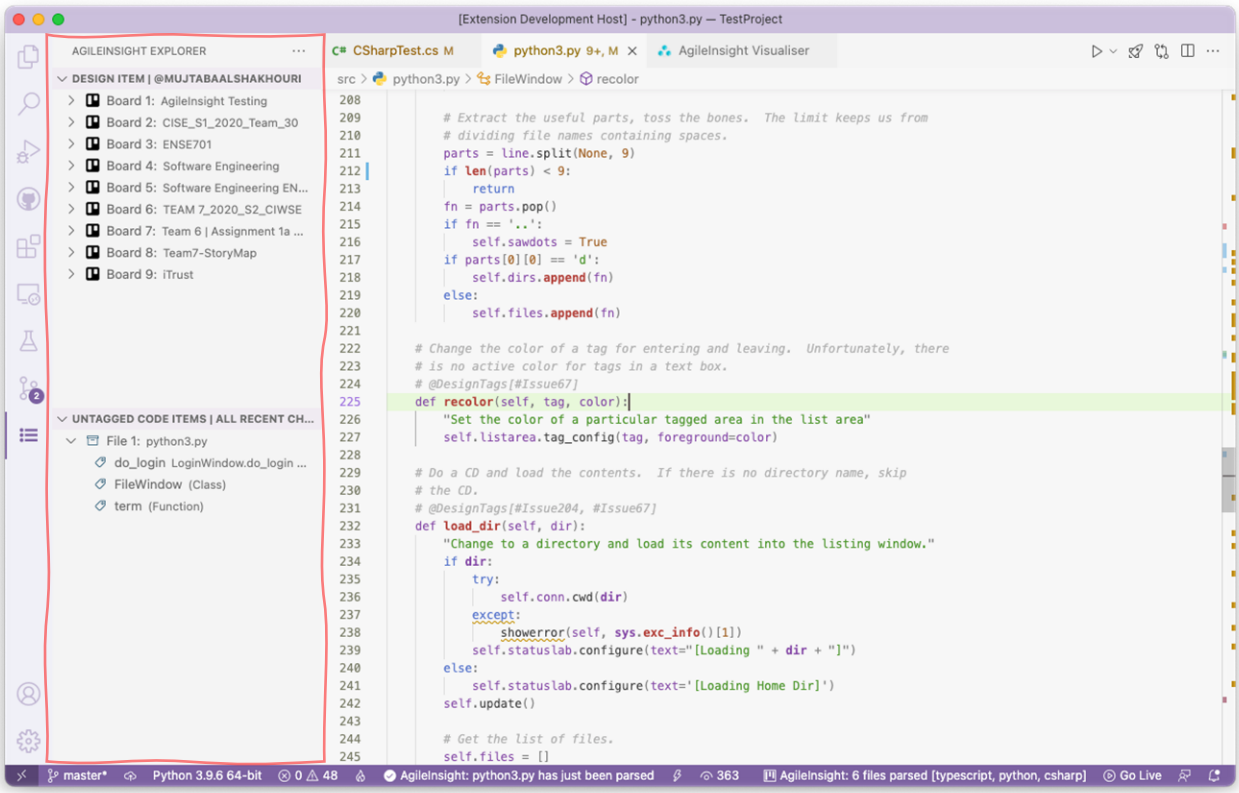


Figure 6.3: An example of a typical view of Vscode running AgileInsight extension. The AgileInsight Explorer panel appears on the left, outlined in red color.

6.2.4 Agile Dashboards Module

This module represents the main entry point to enabling the functionalities related to linking the source code items (CIs) with their original user requirements (referred to here as 'Design Items', or DIs)[64]. From a user-facing perspective, it consists of a primary View—residing in the AgileInsight Explorer—that allows users to access and interact with their agile boards and the relevant design items they contain. The view displays a user's boards dynamically (and in live manner) based on the Agile Dashboard account they are authenticated to (see Authoriser below). The module is driven by few backend components that collaborate together to offer a set of functionalities aimed at supporting the developer. In addition, they offer programmatic endpoints that are used by other modules to access data on a user's boards. These modules include the Scene Interaction Module of the AgileInsight Visualiser, the Source Code Tagging Module and other secondary components such as Hover and Peek features implemented in the editor.

[64] In brief, code item is the equivalent to code artefact, while design item is the equivalent to design artefact. Since the context is user-facing here, these terminologies, which would sound more familiar to a developer, are adopted in tool building. See Chapter 4 for more detail and proper definitions of code artefact and design artefact.

The following presents the chief components of this module along with brief descriptions.

- Dashboard Provider

This component was introduced in section 5.4 where it was described as the intermediary module binding different agile dashboard APIs to the core dashboard functionalities offered by the tool. This is where concrete implementations for a dashboard product—for example, Jira—can be contributed and introduced to the tool without impacting the existing implementation and functionality of AgileInsight. To introduce support for a new dashboard product, an instance Dashboard Provider needs to be written. It needs to fulfil two primary goals; implement an authorisation mechanism to the new dashboard product, and implement a TreeData Provider that handles the retrieval of Design Items from the board, and prepare it to be displayed to the user by the View Pane (see below). These are discussed duly next.

- Authoriser

As the name implies, this module implements the authorisation process to enable AgileInsight to connect to and communicate with a user's Dashboard platform. The authorisation is best implemented using the OAuth framework as described in section 5.4, but basic authorisation would also work well. A user should only need to perform authorisation once[65], and unless the token is manually revoked later, the user can enjoy access to their boards in a transparent manner across any instance of Vscode and across different workspaces. Importantly, this is exactly similar to how developers are used to authorise their IDEs to access their source code version controllers such as GitHub.

Each dashboard provider instance will need to point to an authoriser instance that implements the authorisation as per the APIs of its dashboard service provider.

- Dashboard View Pane

This is the user-facing View Pane mentioned earlier which is located inside the AgileInsight Explorer. It is where design items—e.g., user stories, or issues—are displayed to the user and interacted with. As different dashboard products can adopt different data formats and structures that best suit the agile practice they support (see section 5.4), hence displaying this data in AgileInsight should conform to that structure to guarantee a consistent and undisturbed user experience. To enable this, the Dashboard View Pane was designed so that data to be displayed in it can be provisioned dynamically during runtime by implementors named TreeData Providers (see section 6.4.2). So, while there is one view that keeps the UI design uniform and consistent, its contents can be fulfilled dynamically each time by

[65] If OAuth is implemented, an expiry period can also be set.

a different instance of the TreeData Provider. That specific instance is determined by the implementor Dashboard Provider for the specific dashboard product that the user is authenticated to. This enables the display of content from different dashboard products, while the UI view is fixed. Only one dashboard account can be authorised at any moment in time, hence the View Pane will always have a single TreeData Provider actively connected to it at any point in time.

Figure 6.4 adds insight to this description, and depicts the structure of these parts and how they interplay.

In summary, a user authenticated to Trello will see their card-based design items displayed in this view pane; a user who is instead authenticated to Jira will see instead their issue-based design items organised and displayed in this same view pane.

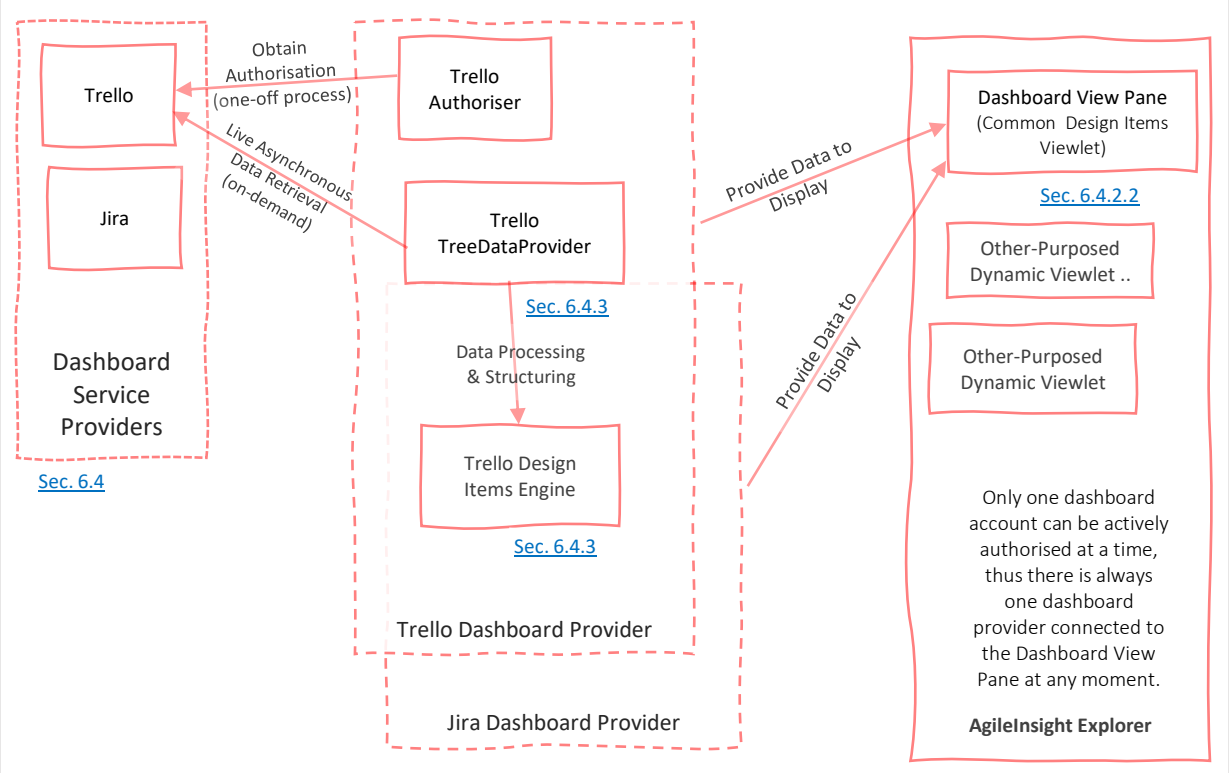


Figure 6.4: A diagram showing the inner components of the Agile Dashboards Module, and how they are positioned and connected to achieve the module's purpose.

- Design Items Engine

An important concept of AgileInsight is making agile user requirements available in the IDE. Since different agile practices adopt variant approaches to structuring these user requirements and can incorporate different related information, the only way to make AgileInsight work smoothly with these different products is to abstract these requirements to their lowest-level element that is common

among them all. Section 5.4 elaborated further on this point. In brief, the term Design Items is a common name that is used to refer to and capture any form (and level) of the user requirements hierarchy used by these practices. It encompasses constructive as well as corrective requirements (see Chapter 4). The smallest unit that is directly actioned by developers should come then at the lowest-level of the hierarchy when displayed to the user. Thus, in the AgileInsight vocabulary, an Actionable Design Item could be a user story, a Jira issue, a Github issue, or a bug. As discussed in Chapter 4, an Actionable Design Item should Ideally refer to the smallest unit of requirement form that is actionable — one that a developer would take as a task and work to implement it into working source code.

With that description in place, a Design Items Engine is thus the module that processes the data received from a user's board, and applies the required abstractions to it so that it conforms to the Design Items concept. This concept was theoretically explained in Chapter 4, and a technical implementation is presented in section 6.4.3. Once the data is encapsulated into Design Item objects, the TreeData Provider can then organise and prepare the items for display on the Dashboard View Pane. This consequently implies that each Dashboard Provider will need to implement its own Design Items Engine for the particular dashboard product it is introducing. While this might appear initially as a complex component, in reality, all of the dashboard products examined and discussed in Chapter 5, provide user-friendly and flexible APIs that make implementing this engine fairly straightforward — albeit to a lesser extent for Trello[66].

6.2.5 Source Code Module

This part is the major component and the backbone of AgileInsight that almost all other modules rely on to achieve their full functionalities. It is where language-agnostic source code parsing takes place. The module achieves its functionality through the cooperation of three parts; two are readily provided by the Vscode platform, and the third is a novel component contributed by AgileInsight.

- Language Servers

Vscode's use and advocacy of Language Servers has been introduced earlier in section 5.5 and applauded for allowing the separation of programming languages service implementation from their consuming parties. Vscode has a very large (and growing) base of language servers that covers the majority of programming languages in use today. Vscode users will normally add a few language support extensions for their desired languages from Vscode's Extension Market before they start using

[66] Unlike Asana or Jira, Trello does not (up until the point of writing) offer a convenient wrapper library to its REST API. It also has a more generic data structure with less categorisation than the others, which meant more effort and text processing was involved to implement the Design Item abstraction. More elaboration is provided in section 6.4.3.

the IDE. Each extension encloses a server-client pair that delivers the services for that language (the server is typically included along—rather than hosted remotely—for performance reasons). What is important for the purpose here is that once a language extension is added to the Vscode environment, then those language services become available to access through a common Vscode interface called Language Services API. To put things in practical terms, AgileInsight will be able, for instance, to invoke an API method to retrieve all code elements in a source code file regardless of that file's programming language.

- Language Clients

A language server provides the core implementation of a programming language. To provide services such as syntax highlight, autocompletion, linting, and other smart features, a client is needed to consume the raw data offered by the server, and to offer implementation in the form of user-facing features—for example, in an editor. In Vscode, the platform is able to support its extensive array of languages through the provision of a client for each language's server, which as mentioned earlier, comes as part of the language support extension when it is added. In fact, a language's server will not be loaded and started by Vscode until a client that needs to use it has been activated. AgileInsight capitalises on those services—both the raw data offered by the server as well as the client's features[67]—consuming them through the common language services APIs, which work the same irrespective of the language underneath.

The next component is made possible thanks to the ability to access the language services in an agnostic manner through those common Vscode APIs.

- Source File Live Parser & Modeller

This component is the part of the Source Code Module that is contributed by AgileInsight. It introduces the ability to parse the entire collection of source file folders (or packages) of any codebase that is opened in Vscode—irrespective of the language of those files. Its operation is focused in two parts. The first is an immediate parsing operation that runs across all folders when a workspace (or a project's folder) is first opened[68]—or when the IDE is first loaded. The operation runs on-the-fly in a separate background process and would typically consume a few seconds depending on the number of files that need to be processed[69]. This generates a complete image of a codebase's source files and its internal structure—from top level folders down to individual functions or methods inside the files. The other

[67] An example of raw data is DocumentSymbols, while syntax highlight and hover are examples of client features

[68] Folders that should be parsed can be configured through a provided user setting.

[69] The actual processing speed is dependent on a language's server-client package that is provisioning the service. A java codebase consisting of 582 files took on average 23.6 seconds for the parsing operation to complete. The operation is non-blocking as it runs in the background. Other language benchmarks are provided in section 7.4.1.

part runs an on-demand process responsible for keeping that source code's image constantly up to date. This process is triggered instantly (but in a controlled manner) whenever a source file has changed or its image has become outdated. The complete image of source files' contents and structure is then made available to the other modules of AgileInsight to enable them to deliver their functionalities. Modules can also invoke a forced parsing process on an individual file if required. Figure 6.5 provides a helpful picture of the components of this module.

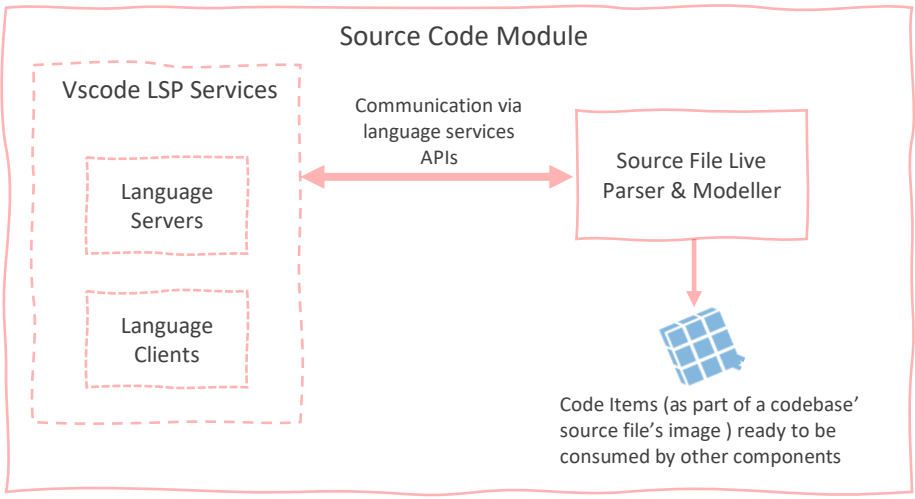


Figure 6.5: An overall view summarising the key components of AgileInsight's Source Code Module and how they interact.

6.2.6 Version Control Interfacing Module

A key part of the linking and synchronisation process between the source code items and the design items involves handling a Commit operation, as well as the use of a few other functionalities provided by version control systems. This module is responsible to cater for these functionalities and accomplishes that by primarily utilising features of a basic Git system. However, it also capitalises on limited features provided by GitHub. The functionalities made available by this module are then consumed by the Source Code Tagging Module and the Tagging Reminder Module—both of which are introduced shortly. Figure 6.6 offers an illustration of these components.

- Git Extension

The Vscode platform comes pre-packed with a Git System Extension. Along with the APIs that Vscode makes available for its extension developers, it also exposes a set of **Commands** that can be executed programmatically. Due to the way extensions are built, most will also contribute their own Commands, that normally become available for use by other extensions—as long as the identifier of the Command is known. In the case of this built-in Git Extension, it exposes both; a set of direct APIs as well as a

number of Commands. AgileInsight makes use of both of these interfaces to accomplish the version control functionalities it requires.

- GitHub Extension

In addition to the primary functionalities consumed from the pre-packed Git Extension, AgileInsight makes use of a key feature provided by the GitHub extension[70]. This feature is called Permalink and, as the name suggests, it offers a permanent link to a selected location in a source file that is bound to the specific version at the moment of creation. This feature is only enabled if a GitHub extension is available with a configured remote repository. Github does not expose this feature as part of its public APIs, but it is available in its list of Commands, and thus AgileInsight capitalises on that to consume the service.

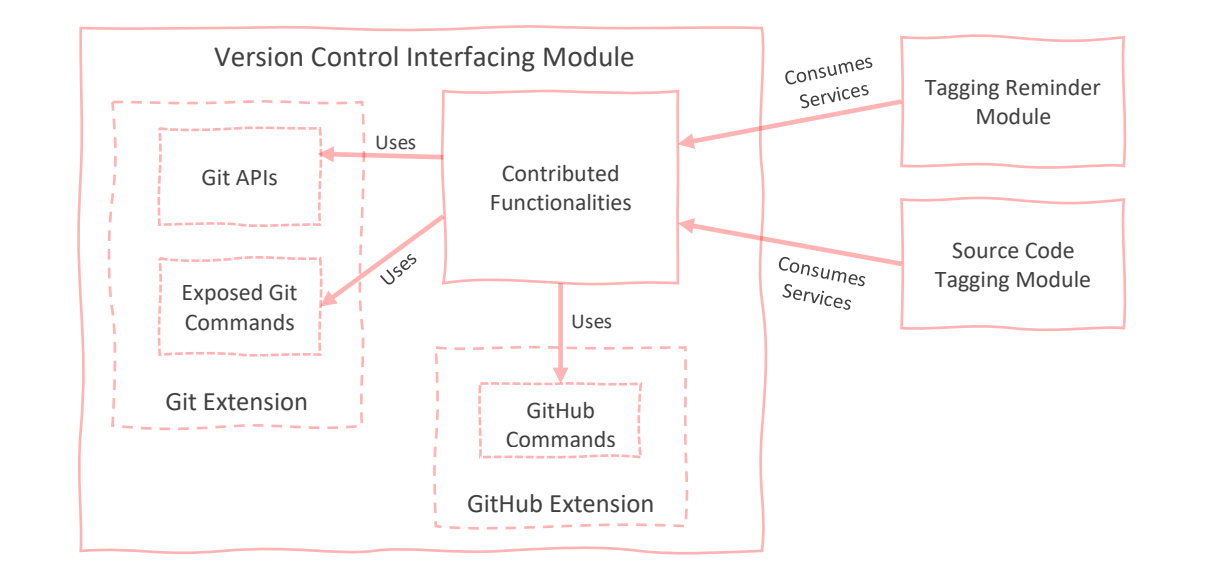


Figure 6.6: An illustration of the components utilised by AgileInsight's Version Control Interfacing Module, and how its services are then consumed by other parts of AgileInsight.

6.2.7 Source Code Tagging Module

After the source files are parsed and their structure is modelled, the Source Code Module will make the data readily available for other modules to use. The Tagging module consumes this data and presents it as a collection of Code Items that are easy to manipulate, access, interact with, and display to the user. From a massive corpus of text and characters that reside passively inside files, the entire source code contents are transformed into live interactive objects that users can instantly pull, look up, refer to, and designate. Whether it is a function, a class, or an entire file, developers can now interact with those code items as individual building blocks that make up their codebase.

[70] As GitHub extension is not a pre-packed extension, AgileInsight will display a prompt to users to suggest installing the extension if it is not available, or to add a remote repository if one is not found.

Thanks to the Agile Dashboard Module, the Design Items come here to complete the picture. In a similar fashion to Code Items, Design Items allowed the transformation of textual user requirements from mere characters in documents into modular, accessible, and structured objects that function in exactly the same manner—they can be instantly looked up, explored, inspected, and referred to as individual building blocks making up the overall design requirements.

In other words, Code Items and Design Items allow the developer to access their code and requirements in small granular units (or building blocks), which they can begin to point to and discuss their effects with consistency and precision. For instance, a developer can point to a function A, and confidently designate it as the implementation of user story B. Equally important, is that these modular and accessible objects are always live and up to date. There is no intermediary module, file, or database, that is storing a copy of the data. A Function's object is a live image of the actual function in the source code. Similarly, a user story's object is a true live image pulled instantly from the user's dashboard. This is made possible thanks to the efficient asynchronous features of modern JavaScript (see section 6.4.2 for more details). Figure 6.7 illustrates an example of Design Items and Code Items being now modular and readily accessible building blocks of a development process.

A significant contribution of this module is making these essential ingredients of the software development process to be conveniently accessible to the developer in a non-intrusive manner right in their IDE, and offer it at the time it is needed in a typical developer's workflow.

In summary, the Tagging module consumes the services from the Agile Dashboard Module and the Source Code Module, to offer a novel synchronisation technique between source code and its original requirements. This module is responsible for both the backend processing as well as the UI presentation of these functionalities. It also consumes services from the Version Control Interfacing Module to complete some of its functions. Further technical details are presented in the upcoming sections.

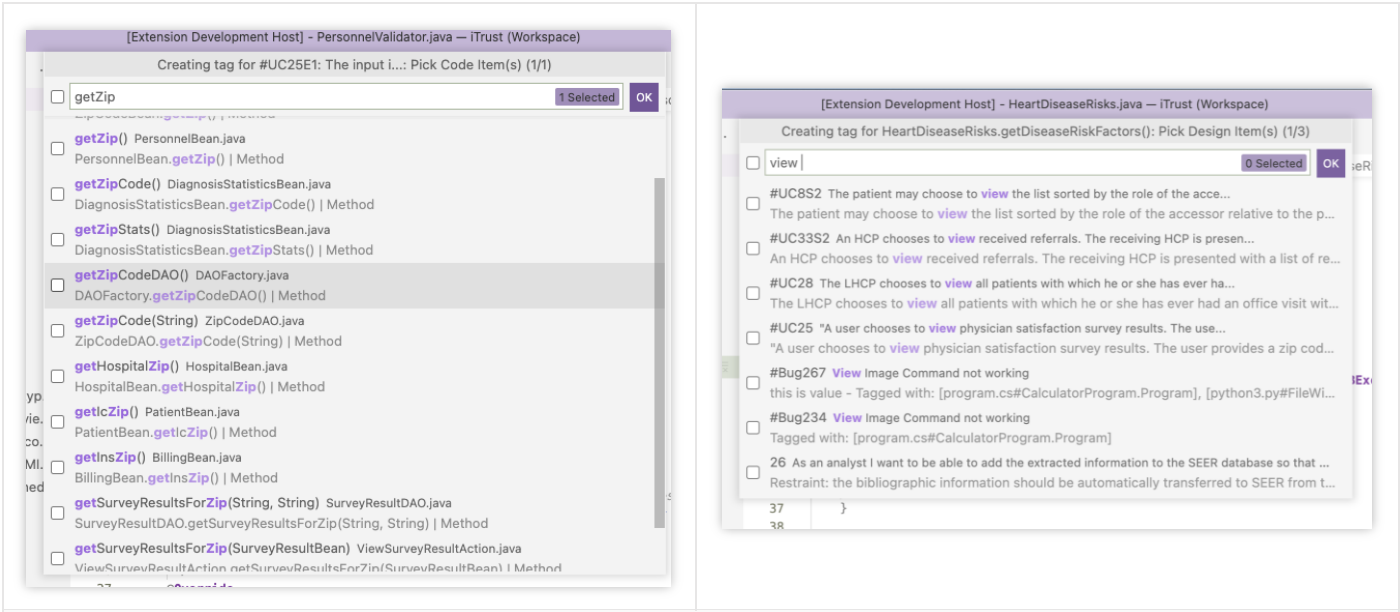


Figure 6.7: Left picture shows the Code Items being offered as accessible components, while the right picture shows the same for Design Items.

6.2.8 Tagging Reminder Module

To support the developer in their tasks without disrupting their normal workflow, care had to be taken to design the features of AgileInsight in a way that integrates seamlessly in developers' way of working. This was possible thanks to frequent collaboration with expert developers (see Chapter 8). Efforts to ensure the introduced features create minimal disturbance should maximise the receptivity of the tool and the likelihood of its prospective adoption. This module attempts to contribute towards this goal by offering smart Tagging prompts when such an action becomes relevant. It employs a few techniques to identify the timing and context of when offering the tagging is likely to be most effective. It also employs an algorithm to identify the exact Code Items for which such tagging has become appropriate or needed. It achieves these capabilities through a combination of user events' detection and a complex textual analysis of Git Changes Blobs[71].

Section 6.8 offers further details and discusses the relevant technicalities of this module

- Git Changes Blob Analyser

This component is responsible for a background process that communicates with Git Extension in an asynchronous and nonblocking manner. It requests source code blobs of recently changed parts of the code, and performs textual analysis to identity Code Items (classes, methods, functions, and the like)

[71] Git Changes Blobs are fragments of plain text source code that is structured in specific order and with special symbols. It consists of the text parts that has undergone some changes, along with their positional context within the file. More detail is offered in section 6.8.2.

that have undergone a recent change. It then offers these to the parent module that consumes them to provide useful functionality to the developer. Because the process is a background asynchronous one, it is able to continuously provide an up to date and live result at any moment in time. More technical details are provided in section 6.8.2.

6.2.9 Visualisation Builder Module

As the name suggests, this module is where the visualisation of source files and their structure is constructed. It consumes the image of source file elements and their structure from the Source Code Module, and generates a city model based on the city metaphor layout introduced earlier. The model serves to reveal the 'physical' structure of the contents of source files, and the entire folder or package hierarchy of a codebase. The motive behind the visualisation is to support developers by alleviating some of the mental effort involved when trying to create a mental image (or a mental map) of a large number of source code pieces to comprehend a problem or a task at hand. Instead, it brings that picture out to the surface in a visual medium that is tangible and shared among all team members.

Once an abstract model of the full codebase structure is generated, it uses the Three.js library to build 3D objects of each element in that model, and then uses a layout algorithm to position each element in its proper location in a three-axis 3D scene node. It further ensures that each of the created 3D objects is tightly bound to the codebase element that it corresponds to (e.g., a file, a method, or a folder). Eventually, the scene node becomes an autonomous 3D object that fully embodies the entire structure of a codebase, in a parent-child hierarchical manner. The module then hands this complete scene node to the Webview Canvas, which finally renders the objects into a visible 3D scene that users can interact with. The technical details of this process are covered in section 6.9.

6.3 3D Canvas Integration—a Webview Solution

In earlier chapters justification was presented for the importance and motivation to integrate 3D visualisation of source code structure into popular IDEs. Chapter 5, added then further elaborations to the argument from a technical perspective. It also discussed the options and challenges of integrating 3D visualisations into today's popular IDE's with regard to technology availability, and highlighted specific technicalities that are a source of common complexities faced, and that must be addressed if the tool is to be practical and win user interest. The previous section offered some high-level answers to the design and solution approach to solving those complexities.

The following discussion seeks to conclude this subject by covering selected technicalities and implementation issues that are thought of particular interest or use to future researchers. This same approach applies to the remaining sections of this chapter (up until section 9) where necessary technical details in key implementation areas are covered to reinforce the higher-level information provided above.

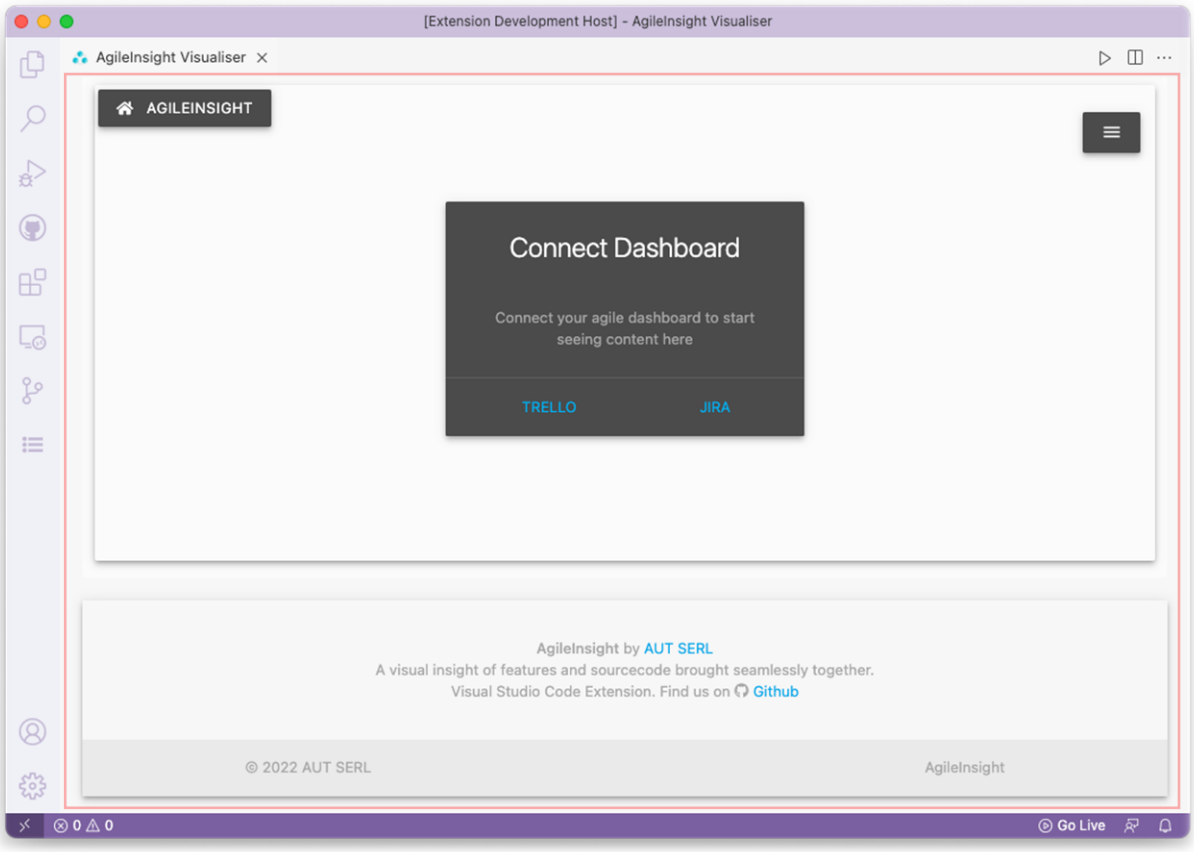


Figure 6.8: A view of AgileInsight's Webview as seen running in Vscode. The perimeter of the Webview component is outlined in light red color.

Figure 6.8 shows the Webview component of AgileInsight occupying the primary working area of Vscode (light red outline). A number of UI elements are also featured. It is worth to note that the entire visible contents of the webview are created using a purely custom web stack (HTML, CSS, JavaScript) that is independent of Vscode's Extension APIs. Yet, it is all running autonomously within Vscode Extension Host environment without any additional components required (for example, no Http server is required). This is an important aspect of WebView's powerful flexibility, and how it enabled the seamless integration of Three.js 3D scenes in Vscode. In this regard, there are a few points to elucidate in terms of how this implementation is achieved.

The Webview Panel. To be able to load a custom HTML view, a Webview panel first needs to be created. This is easily achieved through a direct API method call. The panel on its own produces a tab skeleton—the title bar (see Figure 6.8) and the wide container space it holds. To display content in the container, a custom html data is then provided as a string object through the property: `panel.webview.html`. This object can also be updated dynamically at runtime if needed. However, crafting a full HTML view with numerous interactive UIs and a live 3D scene, and expecting to pass all that in a single string object is surely bound to daunt any developer. This can be alleviated by having a dedicated function or class that dynamically generates the string and injects it with the necessary resources. There are a number of security and configuration measures that are essential for this process to work, and which must be set during the creation of the Webview panel. The following are the most important and relevant configurations for the purpose of this work:

`enableScripts`	This allows the Webview to run client-side scripts. Server-side JavaScript is automatically supported as it will be executed by Node.js before serving the content.
`enableCommandUris`	This is useful for implementing a simple mechanism of UI actions that does not require the use of Webview's special API. More details on this is presented below (see item: 'Using Command URIs').
`localResourceRoots`	Due to the strict security measures around Webviews, this property must be used to declare the directories where trusted resources such as images, scripts, and CSS files can be safely loaded from.
`retainContextWhenHidden`	Vscode's Webview offers two mechanisms to implement state persistence. This flag offers the easiest approach where Vscode will attempt to save all contents automatically when the Webview tab goes into the background, and then restore all when it becomes visible again. However, this method is resource-intense and should be avoided when possible. A manual approach using the methods `setState` and `getState` of Webview's Special API offers better performance, allowing serialisation and de-serialisation of the relevant user and state data only. Vscode's Webview also offers persistence across restarts of the IDE. This type of persistence is called reviving in Vscode, and can be implemented

	by registering a `WebviewPanelSerializer` during the activation stage of the Webview.

Code Block 6.1 demonstrates a code fragment of the above properties in use during Webview Panel creation.

```
panel = vscode.window.createWebviewPanel(
    ViewType.mainViewType,
    title,
    column || vscode.ViewColumn.One,
    {
        //enable automatic state persistence
        retainContextWhenHidden: true,

        // Enable javascript in the webview
        enableScripts: true,
        enableCommandUris: true,

        //Restrict the webview to only load content from the extension's `media` directory.
        //To disallow all local resources, just set localResourceRoots to [].
        localResourceRoots: [
            vscode.Uri.file(path.join(context.extensionPath, 'media'))
        ]
    },
);
panel.iconPath = vscode.Uri.file(context.asAbsolutePath(path.join('media', 'icons', 'light', 'tempLogo.png')));
```

Code Block 6.1: An example code fragment demonstrating the use of some of Webview's security measures that are important to this work.

Loading UI & other Web Stack Libraries. To create any modern HTML-based UI today, use of front-end frameworks and other supporting web stack libraries is almost imperative. For example, in Figure 5.8, the UI is built using Google's Materialise Library. In order to load such necessary resources (scripts, CSS stylesheets, images, and so on) into the sandboxed Webview, three steps are required. The resources need first to be located in a directory within the extension's package, and the path of that directory must be declared under the `localResourceRoots` property as introduced above. In the second step, a special trusted path called WebviewUri must be obtained for each resource using the method: `webview.asWebviewUri`. This trusted path can then be safely injected into the html string object. Otherwise, the Webview will block any local file system paths that are not converted into WebviewUri, or that are located outside the trusted directories. Lastly, a unique nonce[72] value must be created for each resource when loading it into the html content.

Code Block 6.2 illustrates code fragment for loading a trusted resource into the Webview html object.

[72] A nonce value is a randomly generated cryptographic alphanumeric code used to establish the identity of a trusted resource or a communication link.

With these preparations in place, creating the HTML content for the Webview can proceed as in any ordinary web application and almost any custom UI can be created. This includes highly responsive and interactive UIs common in modern web applications. For example, clicking the burger button in Figure 6.8 reveals a popup panel shown in Figure 6.9. However, to ensure a homogeneous look of an extension, Vscode strongly discourages implementing custom UIs in Webviews, unless the feature cannot be implemented using the native Vscode UI stack. As a last remark, additional settings can be further configured by adding a Content Security Policy to the html object—for example, to allow loading resources from an http URL—however elaboration on this is not relevant to the purpose of this work.

```
const threeJs = vscode.Uri.file(
    path.join(context.extensionPath, 'media', 'scripts', 'threejs','threeJSLib.js')
);

const threejsUri = panel.webview.asWebviewUri(threeJs);
```

```
<script nonce="${nonce}" src="${threejsUri}"></script>
```

Code Block 6.2: Code fragments demonstrating the loading of a trusted resource into a Webview's Html object. In the first row, a WebviewUri is obtained for the Three.js library. In the second row, the special Uri is injected along with its nonce value into the html string.

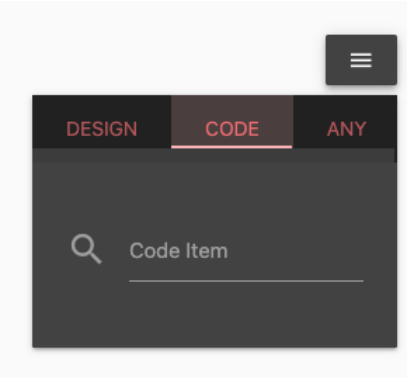


Figure 6.9: Example of an interactive UI implemented in the Webview

Loading the Three.js scene. With HTML content and responsive UI creation having been achieved, the next task is to load the Three.js scene into the Webview. Fortunately, this part is the easiest of the two. The scene itself can be designed and crafted in its own JavaScript file, which is then loaded as part of the html content in the same manner described above. The Three.js renderer object is then simply appended to any desired DOM element of the HTML content—for example, a designated `DIV` element—using the common DOM manipulation technique. As long as the script file containing the scene has been loaded using the correct trusted resource configuration, the Webview should be able to render the scene, as shown in Figure 6.10.

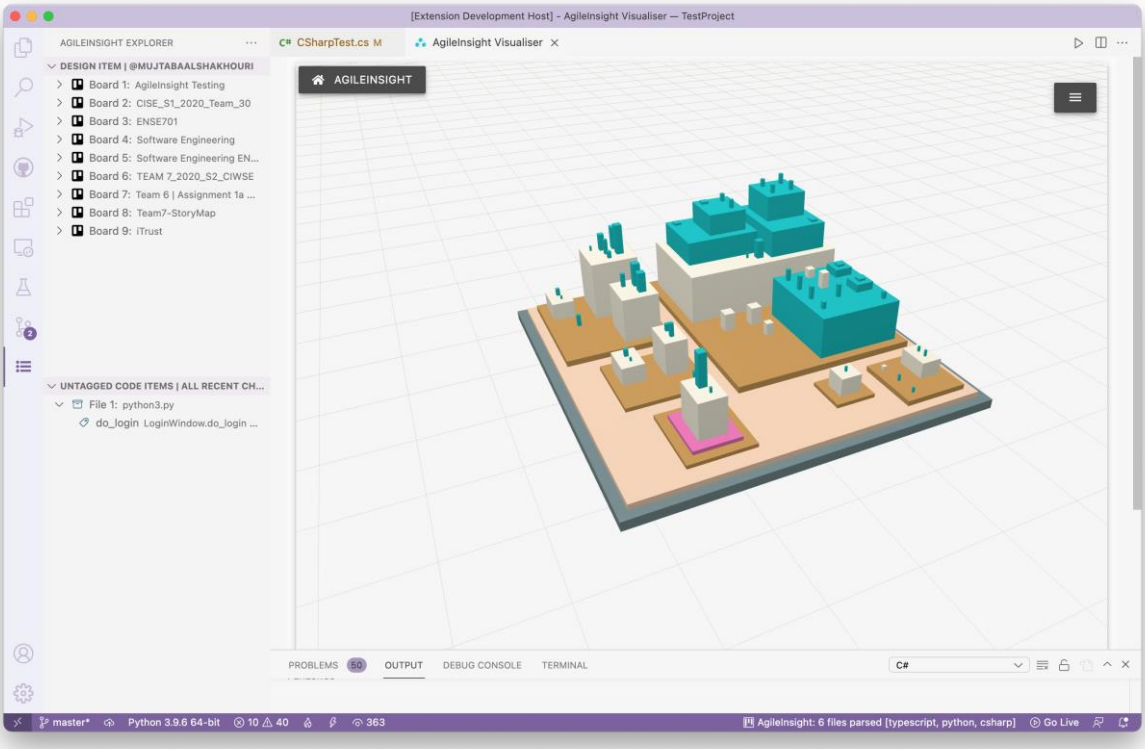


Figure 6.10: A Three.js 3D scene loaded into a custom Vscode Webview

Implementing Actions & Events in the Webview. An HTML view and UI are hardly complete or useful if it is not able to interact with the outside world. The finer support for this type of interaction between the IDE and an embedded web canvas was stated in the previous chapter as an important factor in preferring Vscode over Eclipse. It was argued that it enabled a more-direct and native-like interaction experience for the user, as well as a simpler implementation for the developer. It was also stated that, unlike Eclipse, Vscode offered two-way event communication that facilitated this interactivity. At this point, it is now possible to discuss this surpassing quality in its technical detail, albeit with a degree of brevity.

There are two ways to implement interactivity and communication between the Webview and the rest of the Vscode environment.

1- Using Command URIs. This is the easiest and most direct method. A command identifier is first registered for the extension, which can be done either programmatically during the extension activation stage, or statically in the extension's Manifest file `pacakage.json`. A command identifier is a simple string name, and Code Block 6.3 shows a code fragment for the two types of registration. A Command URI should then be created using the method `vscode.Uri.parse`. That URI can finally be used as a traditional `HREF` value—just as URLs would be typically used—and injected

into the html string under the particular UI where the action needs to be associated. When the user clicks that UI element, the registered action would be triggered, which could simply run a background process, or even reveal a native Vscode UI like a drop menu to select a further specific action or provide an input. Command URIs have an especially useful feature in that they accept any JSON serializable parameters, enabling the developer to pass entire objects. Equipped with the ability to dynamically generate a Command URI, the levels of interactivity that can be achieved become almost limitless. Code Block 6.4 demonstrates in code fragments the generation of Command URIs and its injection into the Webview's html content.

Command URIs, however, offer only a one-way form of communication—from the sandboxed Webview's html to the rest of the Vscode environment and its extension APIs. For two-way communication ability, the second approach must be used.

```
vscode.commands.registerCommand('agileInsight.connDashboard', async () => {
    userInput.showConnToDashboard(context);
}),
```

```
{
    "title": "Disable AgileInsight Codelens",
    "command": "agileInsight.disableCodeLens",
    "category": "AgileInsight"
},
```

Code Block 6.3: Two types of Command Identifier registration methods. Top fragment shows a dynamic way while lower fragment shows a static one used in the manifest file.

```
const connCommandUri = vscode.Uri.parse(`command:agileInsight.connDashboard`);
```

```
<div class="container">
<a href="${connCommandUri}" class="#40c4ff light-blue-text accent-2 hoverable">Jira</a>
</div>
```

Code Block 6.4: Example of implementing a direct UI action in Webview using the Command URI technique. Top fragment shows generating the URI, while lower fragment shows its injection into the Webview's html content.

2- Using the Special Webview API. As described above, the special Webview API is provided to enable seamless communication between the sandboxed Webview and the rest of the Vscode environment. The API is accessed from inside the Webview through the one-time callable handle `acquireVsCodeApi`. The method `postMessage` can then be used to send any data that is JSON serialisable to the outer Vscode Extension context. From there, the listener function `panel.webview.onDidReceiveMessage` is used to intercept the message and receive the transmitted data. Code Block 6.5 demonstrates an example of this communication method with

a code fragment. An exactly similar approach can be used to initiate the communication from the outer Vscode Extension environment down to the sandboxed Webview.

```
vscode.postMessage({
    command: 'showGlyphContextMenu',
    fullIdentifier: GLYPH_POINTED_AT.name,
    selectionRange: GLYPH_POINTED_AT.userData.selectionRange,
    uuid: GLYPH_POINTED_AT.uuid
});
```

```
this.panel.webview.onDidReceiveMessage(
    message => {
        switch (message.command) {
            case 'showGlyphContextMenu':
                const cancellationToken = new vscode.CancellationTokenSource().token;
                let pickItems = [];
                if (message.kind == 'Package' || message.kind == 'Workspace') {
                    pickItems = [{
                        label: 'Focused View', picked: true,
                        description: 'See it in isolated mode for easier inspection'
                    }];
                    vscode.window.showQuickPick(pickItems, {
                        ignoreFocusOut: false,
                        title: `Item selected: [${message.longerName}]`
                    }, cancellationToken);
                } else {

                }
        }
    });
```

Code Block 6.5: Code fragment showing example of message posting initiated from inside the Webview (top), and its reception by the outer Vscode Extension environment context (bottom).

Implementing interactivity for Three.js scene. To complete the topic of interactivity, a particular level of user interaction must be addressed. It was discussed earlier that being able to interact with the visualised 3D objects inside the Three.js scene is a particularly important feature for the purpose of this work. This level of interaction is crucial as a goal of this research is to tightly link the source code items, in their visualised forms, to the design items they are implementing. Most 3D libraries that are at the level of sophistication of Three.js offer a mechanism called Object Picking to enable identification of the object being pointed at by the user. This mechanism is normally implemented using a Ray Casting[73] technique. Covering the details of this implementation is unnecessary here, however, Three.js offers the supporting class `RayCaster` to implement it. Once an object has been identified, the remaining chain of user interactivity can be achieved using the same communication techniques described earlier.

[73] Three.js does offer another technique that is based on pixel's color identification, and is normally referred to as GPU-based method. The `GPUPickHelper` class of Three.js is used to implement this technique.

As for the other way around—that is, to initiate the interactivity chain from the Vscode environment down to the objects in the 3D scene—the `postMessage` can be invoked in this case from the outer Vscode extension API, with an `onDidReceiveMessage` listener function registered inside the Webview. The methods `getObjectByName` or `getObjectByProperty` of the Three.js API can then be used to lookup the exact object in the scene's tree of nodes and apply any desired action on it.

In conclusion of this section, despite its strict isolation and the security measures around it, the sandboxed Webview of Vscode is still able to offer highly seamless integration and interactivity levels with the rest of the Vscode extension environment. Using the configurations described in this section, the event communication methods, and the user interactivity implementation techniques, it was possible to achieve the level of fusion of a 3D environment with a popular IDE that was called for at the beginning of this research. Most importantly, this goal is achieved without undermining the performance and responsivity of the application—as have been tested and demonstrated in section 5.3.

6.4 Agile Dashboard Integration

Section 6.2 presented an overall view of the design and inner components of this module. This section introduces now the relevant technical details that are thought to be of particular interest or use. The Design Items Engine and the TreeData Provider component are the parts where elaboration is mostly deserved. However, before diving into their specifics, implementation of OAuth for the Authorisation Module first warrants some brief attention.

6.4.1 OAuth in Vscode

Figure 6.11 shows two views of the AgileInsight Explorer; before authorisation to any dashboard product on the left, and after authorising to Trello on the right. The full flow of an OAuth authorisation process is well-known, and the reader is referred to Appendix III: for an example of the process in action. Of interest to the discussion here are some particular technical details of how this was implemented in Vscode, a quick glimpse of the challenges faced, and a few tips that could ease the journey of future similar efforts.

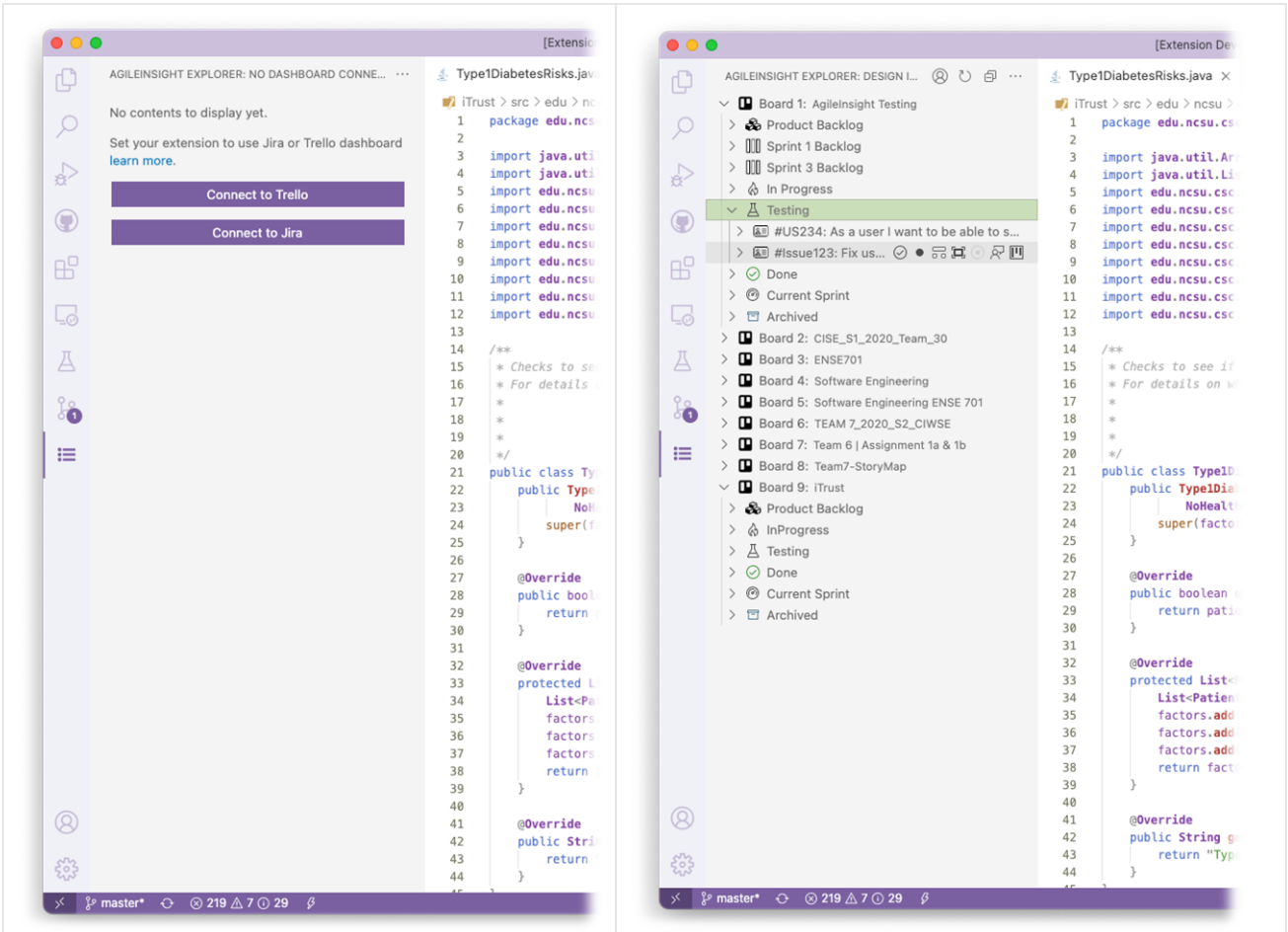


Figure 6.11: Two views of AgileInsight Explorer Panel showing its content before authorisation to any dashboard is granted (left), and that after the user has granted authorisation to their Trello account (right).

A high-level description of how OAuth works was presented in section 5.4. Covering the overall details of OAuth implementation is irrelevant to this work. However, there are few aspects that are of special interest to discuss. For the process to work out flawlessly in Vscode, a few particular steps are required:

A. Vscode needs to transfer the user to the authorisation page of their service provider upon initiation of the authorisation request.

B. When initiating the OAuth process—which can be done via the publicly available JavaScript library of OAuth implementation—it must be configured with a callback address to redirect the user back to the Vscode application.

C. Vscode must have an active process listening to intercept the redirection when it happens, and forward it to AgileInsight to process it.

A particular challenge faced at the time of building AgileInsight is that OAuth support in Vscode was still in its early stages, and their documentation had no guidance yet on how it could be implemented in the platform. However, and as will be shortly revealed below, it was clearly still achievable. Steps A and B were not presenting any challenge as their solution was readily available. However, step C proved to be arduous to overcome. A considerable effort was put to examine developers' discussions on StackOverflow and related threads in Vscode's Github Issues. While a number of discussions proposed the use of an external server to listen to the callback, and some even recommended making use of Microsoft's Azure service to achieve the full OAuth flow, there was a good indication that the process can be accomplished natively within Vscode. This was evident from an initial examination of Vscode's Github extension, which at the time, was providing an OAuth-based authorisation—albeit, it was still experimental and a fail-safe manual approach was being offered in case the redirection fails.[74] Extensive examination of the above resources kept leading to an interface named `ProtocolHandler` with the registration method `registerProtocolHandler`. Unfortunately, the only place these API resources could be found was in one of Vscode's release notes—they did not exist in the actual API nor in the official documentation. A deeper investigation uncovered later their existence in Vscode's Insiders version—an option that is clearly undesirable.

An accidental trial and error process revealed eventually that the API resources were silently renamed to `UriHandler` and `handleUri`, respectively. This enabled the full cycle to be finally achieved natively within Vscode without relying on any external services. At the time of writing, some documentation

[74] The manual approach was simply advising the user to copy their token key that was displayed in the footnote of the authorisation page, and then paste into a field in their Vscode extension.

appears to have been recently added that finally offered an example of implementing OAuth, and stated the correct naming of the API resources—albeit, it is covered under a vaguely named topic[75].

To summarise this discussion, a general outline of the process works as follows:

1- The open-source OAuth API is used to initiate the authorisation process. It is configured with the correct callback address (named callback URI in Vscode's vocabulary) leading to the specific extension, i.e., AgileInsight (see below).

2- The interface `UriHandler` is used to register a listener process during the activation stage of the extension.

3- When a callback request is received by the Vscode platform, it identifies the target extension from a particular segment of the callback URI, called authority segment. Once it identifies the receiving extension, Vscode then executes the UriHandler service that was registered by that extension.

This process is further illustrated in Code Block 6.6.

```
vscode://vscode.git/clone?url=foobar
\____/   \_______/ \___/ \_________/
  |          |       |       |
scheme    authority   path   query
```

```
vscode://AUT-SERL.agile-insight/callback
```

```
class AuthUriHandler implements vscode.UriHandler {
    async handleUri(uri: vscode.Uri) {
        try {
            await AgileAuth.authorisationCallback(uri.toString(true));
        }
        catch (error) {
            vscode.window.showWarningMessage(`Error in handling authorisation callback: `, error);
        }
        console.log('URI query: ', uri.query);
        console.log('URI JSON: ', uri.toJSON());
    }
}
```

```
vscode.window.registerUriHandler(new AuthUriHandler())
```

Code Block 6.6: A Code fragment showing key steps required for implementing the listener service as part of a full and native OAuth authorisation process in Vscode. First row explains the callback URI structure while the second row shows the corresponding callback URI for AgileInsight (note that authority is composed of <Publisher>.<Name> as defined in the extension's Manifest). Third row illustrates a simple UriHandler implementation, with the last row demonstrating its registration that needs to take place during the extension activation.

[75] See: https://code.visualstudio.com/api/advanced-topics/remote-extensions

6.4.2 TreeData Provider

A quick revisit to the diagram in Figure 6.4 could be helpful at this point. In the earlier discussions, it was revealed that a TreeData Provider is responsible to **retrieve the data** (also referred to as user's resources) from a user's dashboard account and to **prepare it for display** on the common Dashboard View Pane. While a few of the component's access operations perform some data modifications, the vast majority are primarily data-fetching in nature.

Live Data. It is important to re-emphasise at this point that all components in AgileInsight use zero intermediary files, caches, or background databases. As argued earlier, this is an important design aspect that is set to ensure the autonomous nature of the tool, eliminate (or at least reduce) adoption barriers, and simplify user entry. This means that all processing and data preparation needs to occur in real-time and on the fly. The performance of each of those components involved hence becomes critical. To achieve this, the implementation in this work is heavily reliant on the modern asynchronous features of JavaScript—Promises and Async/Await. By properly following the development guidelines of these features, it was possible to implement all required live operations with no noticeable performance issues.

6.4.2.1 Data Retrieval & Manipulation

Retrieval or manipulation of the data is accomplished through a product's REST API. These HTTP APIs can be conveniently invoked using the Node.js library, `node-fetch`. The process is predominantly shaped by http requests for the retrieval and manipulation of user's resources. For example—and in the context of the dashboard product, Trello—its TreeData Provider will typically need to implement several methods, each to retrieve a card, a list, a board, and a few others to modify content such as the description of a card. As described earlier, most dashboard products have a highly structured and well-defined data format, and this structured data is normally exposed to consuming parties in JSON-based informational objects. This highly simplifies the retrieval and manipulation of the data. Moreover, unlike Trello, products such as Jira and Asana offer convenient wrapper libraries for their REST APIs, so implementation for these products is further simplified. With that said, implementing this component comes to a typical development routine that is not thought to warrant further elaboration. The only aspect that needs to be accounted for in the implementation is that the data must be retrieved asynchronously—which is conveniently supported by the node-fetch library. The code examples provided in Figure 5.16 to Figure 5.19 offer the reader a sufficient and helpful reference here. Rather than packing this section with other unnecessary examples of the implementation, Figure 6.12 presents

instead a partial list of a few methods' names implemented, which should help to give an overall picture of the kind of task this component needs to fulfil.

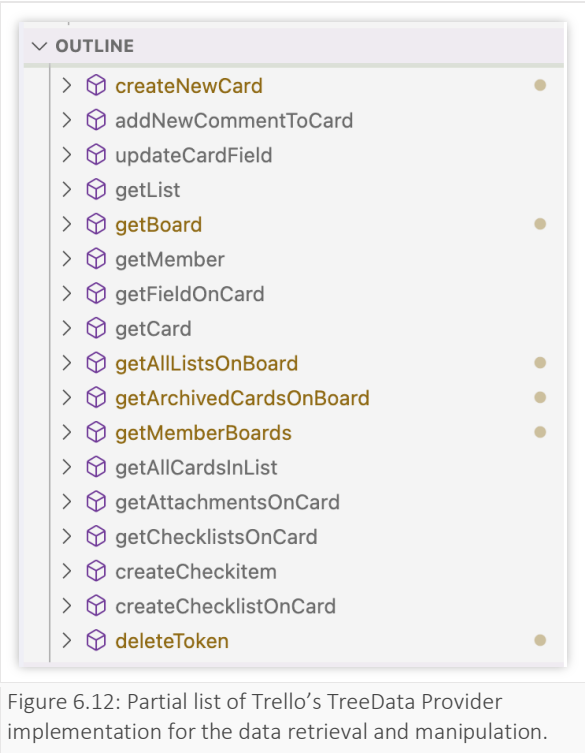


Figure 6.12: Partial list of Trello's TreeData Provider implementation for the data retrieval and manipulation.

6.4.2.2 Data Preparation—Building the Display Structure

After fetching the data, some elaboration is needed to prepare it for display. The preparation is accomplished in collaboration with the Design Items Engine. It can involve a varying level of text processing and data conceptualising, based on the nature of data structure adopted by the dashboard product, and on the accessibility provided by its API.

The primary goal of the preparation process is organising the data in a structure that closely matches that which the user is familiar with when accessing their data on the service provider's dashboard, but that can at the same time be displayed concisely on the View Pane. The former condition is important to ensure a minimal-to-zero disruption to the user's already-established way of locating and working with the data. It is helpful to recall that this data is primarily user requirements, and that it is structured in a certain style that is normally influenced by a certain agile practice. The display of this data on AgileInsight should thus be loyal to that specific style or agile practice. This keeps the mental model (or framework) of the data untouched in the user's mind, and consequently should increase the tool's appeal to the user. However, this loyalty should be pursued equally with three additional elements that must also be met:

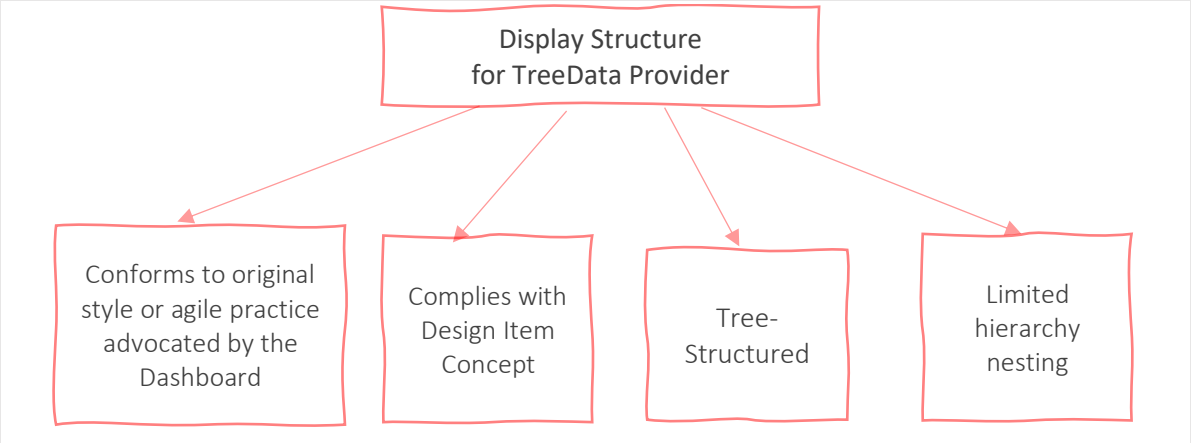


Figure 6.13: The design conditions that should drive the building of the Display Structure for the TreeDataProvider.

1- **Tree-Structured.** As the name of TreeData Provider suggests, the data must be prepared in a form that is suitable for display in a tree structure. This is governed by Vscode's own implementation of the interface `TreeDataProvider`, which is used in this work to display items in the View Pane.

2- **Design Item Modelled.** The data must be modelled to comply with the Design Item concept that was presented in Chapter 4 Additionally, the primary data point sitting at the lowest level of the tree structure must be the smallest unit of actionable user requirement as discussed earlier. The Design Item concept is fundamental to this work, and along with the Code Item concept, they are instrumental to realising the overall objective. More details on this part are covered below.

3- **Limited Nesting.** The hierarchy of the tree data should ideally not extend beyond four levels. This is in line with the design recommendations of Vscode, which is intended to offer the best user experience. Needless to say, this is a quality aim rather than an operational one.

The diagram in Figure 6.13 summarises the conditions that should drive the design of the display structure. With those conditions taken into account, the design of the display structure for each dashboard product can proceed with ample flexibility and in a way that best fulfils the objectives as a whole. The TreeData Provider adopted in this work follows the same implementation and design rules set by Vscode's `TreeDataProvider` interface, which is well-documented with sufficient examples. Covering further technical details is thus irrelevant here and the reader is referred instead to Vscode's

extension development platform for more details[76]. However, a discussion of the Display Structure that was built for Trello should provide a helpful and guiding example. This is presented next.

Trello's Display Structure. The right side of Figure 6.11 provides a depiction of the display structure developed for Trello's retrieved data—which at this point we can comfortably call, Trello's Design Items. Looking at the tree structure appearing in that figure, the board items come visibly atop the hierarchy. Under each board comes next a varying category of lists, such as Sprint backlogs, Product backlogs, In Progress, Testing, and so on. In the third level comes the smallest Actionable Design Items such as user stories, issues, or even corrective design items such as bugs. There are a number of points that demand elaboration here.

1- Identifying Relevant Data. Only data that fits the Design Item concept should be processed and displayed. For example, a user could have a number of Trello boards that contain project data that is unrelated to software development. This situation poses a greater challenge for Trello boards than for Jira, Asana, and other boards that are more oriented towards software development. As indicated earlier, Trello is generic in nature and was designed for a broader spectrum of users and applications. The key approach to work around this problem is to look for data that fall into the Design Item definition, and only display board objects that contain such design items. Applying this rule in a successive manner allows for the identification of applicable Card objects, which in turn determines applicable List objects, which finally determines applicable Board objects[77]. A second heuristic approach based on simple text processing is also applied to aid with the identifications. For example, some List objects might not contain a design item object at the time, but they should nonetheless be represented—for example, an empty Done or Testing List. The names of these lists can hence be processed textually via simple heuristic rules to determine if they are relevant. Any list that is identified under this category is referred to as an Actionable List.

   These two probing approaches (testing for Design Item fit, and the text processing) are where the Design Items Engine comes to be of help. The engine takes care of filtering out the non-relevant data and then moulding those that are applicable into Design Item objects. Admittedly, in the case of the Trello implementation, this part required more work and sophistication than it would likely do for other products for the reasons mentioned earlier. For example, Jira and

[76] https://code.visualstudio.com/api/extension-guides/tree-view

[77] Card, List and Board objects are part of Trello's data structure scheme. For a full list of object types the reader is referred to Trello's official API Reference page: https://developer.atlassian.com/cloud/trello/rest/

Asana[78] come readily with software-oriented data objects such as Tasks, Issues, and Backlogs, making the identification and retrieval process straightforward. For the more domain-agnostic Trello, however, these data objects need to be dynamically created based on the nature of the data present.

2- Dynamic Structure. Continuing with the arguments above, it becomes evident that for each board, its inner display structure should dynamically conform to the data present in that board. A Product Backlog should thus be represented in the structure only if the board has such an object. Similarly, only lists containing Actionable Design Items should be represented—e.g., a Sprint Backlog or a Done List. To adhere to the condition of keeping the original style, it is important to keep and use the same object naming found on the user's board. Figure 6.14 showcases a few examples of this dynamic display structure in action.

3- Dynamic Context Actions. To offer a seamless integrated experience of Dashboards within Vscode, it is imperative to make available common functionalities that developers would typically use during development. Ideally, the developer should be able to perform all relevant access and modification operations on their dashboard design items without leaving their development environment. This would greatly minimise their context switching and should increase the benefits of the tool. It also aligns strongly with the evaluation findings and feedback as covered in Chapter 8. Context actions on design items can be offered via quick action buttons that are temporarily displayed when the user hovers on an item, or via right-click dropdown menus. Other Vscode users tend to be command-inclined rather than clickers, and for that, Vscode offers contextual commands via its popular Command Palette (see Figure 6.15). The presented actions are dynamic in that they can change based on the specific item selected and its state. The actions should also be adapted to suit the specific dashboard product that the user is connected to at the time. The dynamic nature and adaptability are supported by the presented design and by the modularity of Vscode Extension APIs. Figure 6.14 to Figure 6.16 demonstrate the dynamic display structure as implemented for Trello, along with examples of the features discussed above.

[78] Asana is Kanban oriented, and 'Task' is its basic unit of work, which can also have subtasks. For Jira, 'Issue' stands as its primarily unit of work, which can be of different types (e.g., constructive or corrective), and can have children issues known as subtasks. See: https://developer.atlassian.com/cloud/jira/platform/rest/v3/api-group-issues/#api-rest-api-3-issue-post.

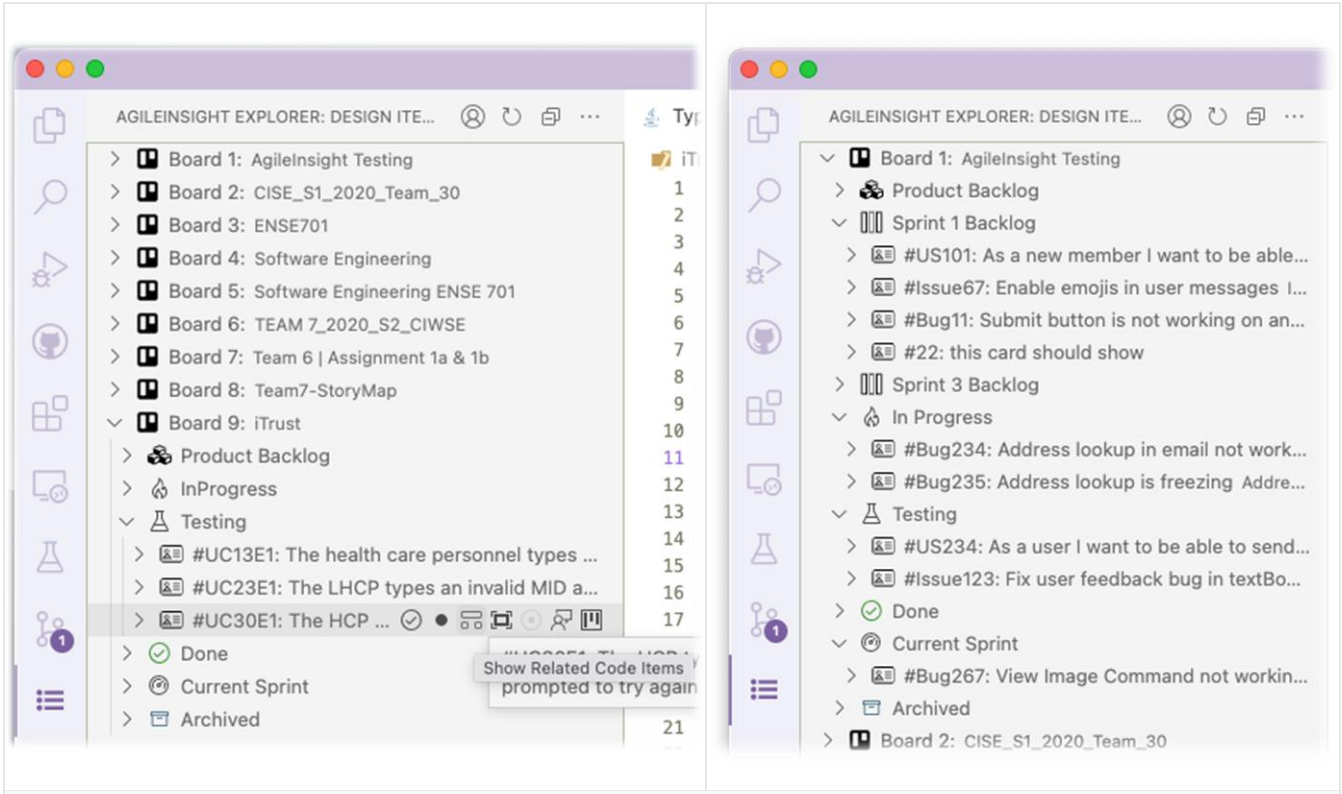


Figure 6.14: Two views of Dashboard View Pane illustrating the dynamic implementation of the display structure as implemented for Trello. On the left, no sprint backlogs are showing as the board has no such active lists. Item-specific actions are also showing for the item hovered on. On the right, a slightly varying structure is displayed, which matches the actual Lists that are active on this particular board. Note that List naming is identical to that on the actual board (see Figure 6.15). The TreeDataProvider must be able to reflect the actual structure on user's board in a live and on-the-fly manner. Lastly, while the mixed nature of design items showing on the left is not strictly complying to a specific practice, they are solely presented here to illustrate the Design Item concept and the applied heuristic to dynamically identify such items.

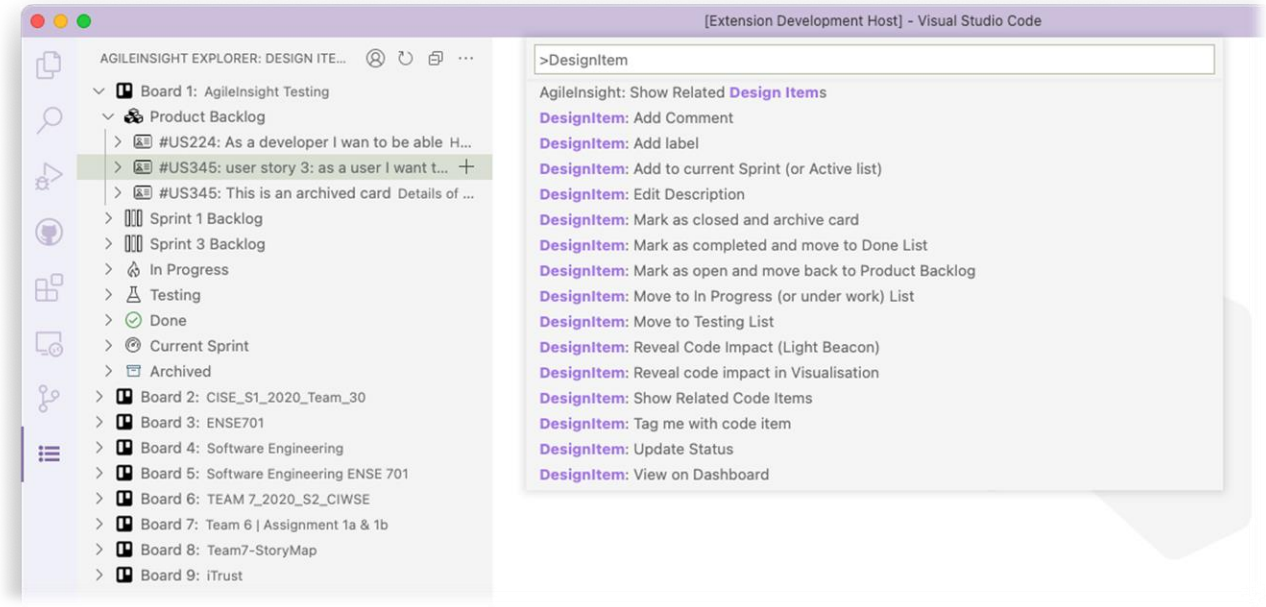


Figure 6.15: Illustration of some contextual action commands as seen in Vscode's Command Palette. Note that a broad list of actions is being shown here for demonstration purposes. In actual implementation, only those applicable to the selected Design Item will show.

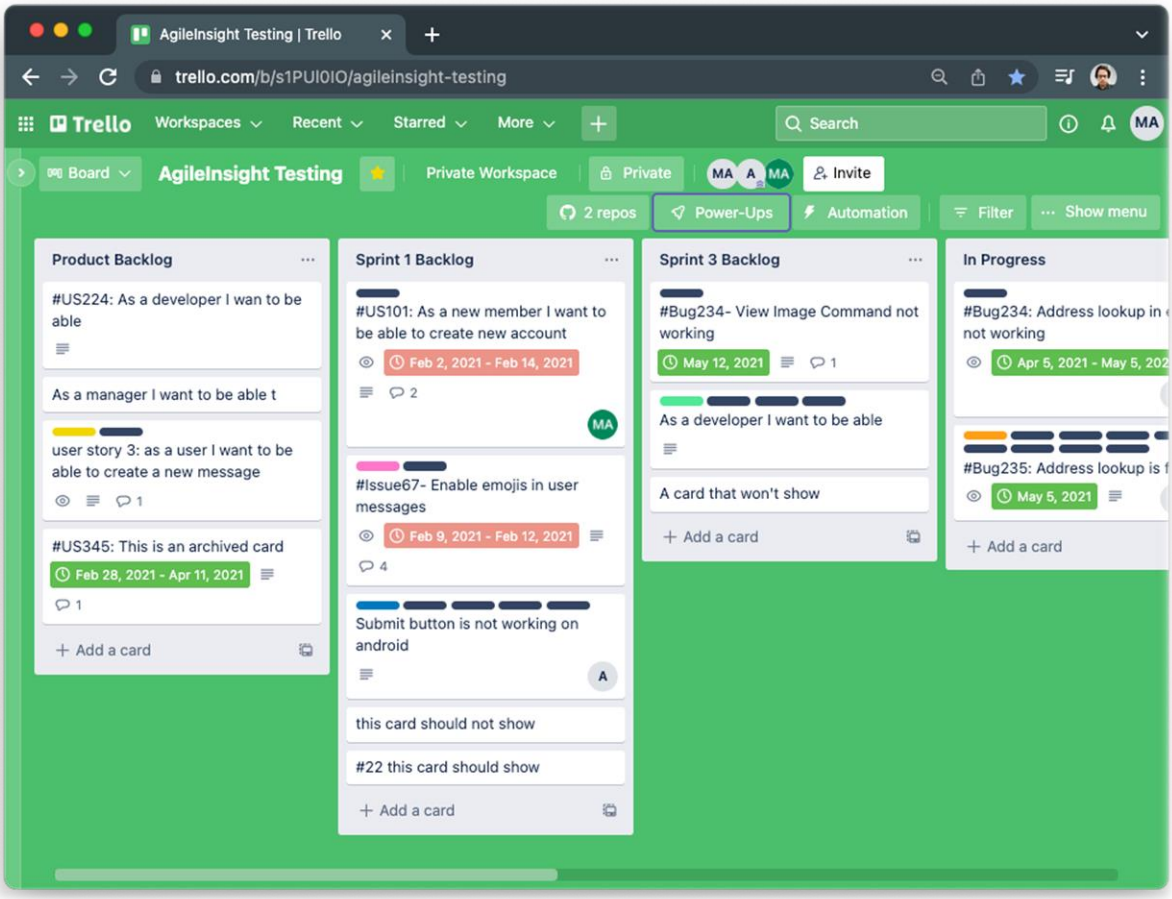


Figure 6.16: A partial view of the actual Trello board contents corresponding to the Design Items display structure showcased on the right side of Figure 6.14.

6.4.3 Design Items Engine

The purpose of this component should have become clear at this point—it helps the TreeDataProvider in provisioning the design items. It is where the abstraction of varying types of user requirements into a common concept—the Design Item—takes place, allowing the tool to work and function in a homogeneous manner regardless of the agile practice under use. The Design Item concept introduced in Chapter 4 is a general theoretic concept, which can be implemented differently depending on the application purpose. This section presents the implementation adopted in this work. It consists of a general rule that needs to be adhered to for integrating any dashboard product. A second part that is specific to Trello's implementation is then presented to shed light on the technical details that are of interest, and to work as guidance for other implementations.

6.4.3.1 Design Items—Abstracting User Requirements

In concept, and as has been broadly covered in Chapter 4, a design item is any written or documented software requirement intended to be eventually actioned and turned into software code. In practice—

and in the specific context of agile software development that is using digital dashboards, which is at the centre of this work—it is that small actionable requirement that is well-demarcated and that resides on some medium waiting for a developer to turn it into code. Ideally, it will have a unique identifier. This simple definition is all that is required to turn the concept into a practical reality. It is seen as generic enough to encompass almost all agile software development practices—and probably some of those more traditional approaches. The pragmatic motive behind this idea is simply to be able to handle any type of functional user requirement and be able to tightly link it to its specific code manifestation: in other words, to be able to manage it as a well-bounded requirement entity with a well-bounded effect on the source code—which is largely possible thanks to the small-sized requirements advocated by agile practices. Thus, any form of user requirement that fits this definition can be abstracted and referred to simply as a Design Item. It makes it possible to develop tools such as AgileInsight that can work with a wide range of development practices, rather than being constrained to a particular one. The simple diagram in Figure 6.17 illustrates a workable abstraction of this concept.

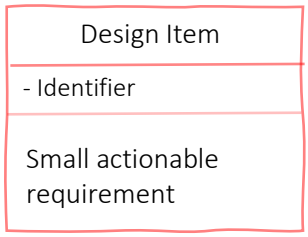


Figure 6.17: A simple practical approach to abstract actionable user requirements into a common Design Item concept that can be applied to wide range of development practices.

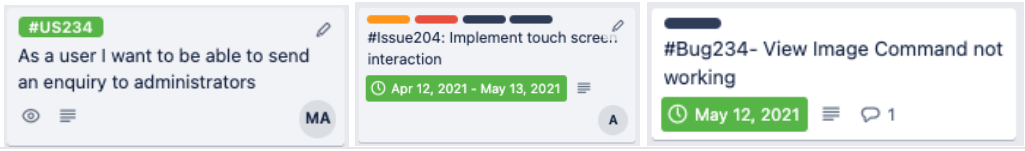
As for the dashboard products that were examined in this work, Jira, Asana, and Trello all feature readily defined objects that closely fit this definition. These are; Issues, Tasks, and Cards, respectively. Jira enforces unique identifiers as well, which greatly simplifies the process. For an agile software development context, these objects should represent the starting point where Design Items—those small pieces of user requirements—can be looked up and abstracted.

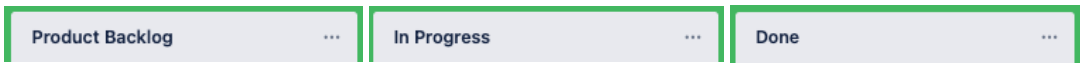
The Design Item structure above is a minimalist definition that is just sufficient to operationalise the concept in this work. In practice, it is an abstract object that can be extended with a few additional elements to expand its functionality or to better suit a specific dashboard product. For example, parent and child reference elements can be added to support such relationships—which for instance, can be suitable to incorporate sub-tasks for Asana's Task object.

The following discussion describes how the abstraction was implemented for Trello.

6.4.3.2 Implementing Design Items for Trello

Even though dashboard products are greatly influenced by agile software practices, they are nonetheless designed to be used in a wide spectrum of general project managements applications. This is very true for Trello. Thus, to be able to identify Design Items that exist on a Trello board, it is imperative to establish a few assumptions. However, these assumptions are minimal, and are largely guided by the common practices of these boards in the agile software industry. The assumptions are applicable to the objects Card and List of Trello's data scheme.

Assumption 1: Design Item Cards Use Unique Identifiers
For a card on a Trello board to be recognised as a Design Item, it must have a unique Identifier[79]. This can be in a label on the card or in the card's title. An identifier must start with the symbol '#' followed by alphanumeric characters. Ideally, the length of the alphanumeric characters should be between 5-6 digits but this is currently not enforced.
Valid Examples: 

Assumption 2: Actionable Lists Use Common Naming
For a list on a Trello board to be recognised as an Actionable List, it must follow naming conventions common in agile software development practice. Those common namings are used by the Design Items Engine via regular expressions to identify such lists on a board, and eventually to represent them on the Display Structure.
Valid Examples: 

The two assumptions above represent the wider framework based on which Trello's Design Items Engine recognises Design Items on a Trello board. The detail of this implementation is presented next.

[79] The identifier must be unique across a board. Further elaboration is discussed in section 6.10.

1. Unique Identifiers In Trello's Design Items

Unlike Jira and some other products, Trello does not enforce unique identifiers for its cards[80]. However, developers who use Trello would normally use an agreed system of numbering to their cards. A quick inspection and later feedback from experts revealed that such numbering is either used as a separate label or as a prefix to a card title. The symbol '#' is also commonly used as a prefix to numbering items on development platforms (e.g., Issues on Github) and has come to be a widely popular practice among developers. Hence, this assumption aligns well with the actual practice in the developer community and is not expected to require any change to already established ways of working. In other words, this assumption was carefully chosen so AgileInsight would work out of the box with most existing agile boards.

The actual heuristic implemented for Trello's Design Items Engine works by checking all labels on a card to look for a valid identifier. If this step fails, then it inspects the card's title to check if it is prefixed with a valid identifier. If both attempts fail to recognise a valid identifier then the card is not recognised as an actionable design item and is ignored[81]. If a card has more than one valid identifier then the first match is considered. While this heuristic is presently hardcoded, it is intended as a workable solution that could easily be expanded to support user customisation in future work. The following pseudocode presents this heuristic as implemented by AgileInsight.

```
set CardIsDesignItem to False
set CardIdentifier to NULL
    For Label of LabelsOnCard
       If Label starts with '#' then
         set CardIdentifier to Label
         set CardIsDesignItem to True
         exit loop
       EndIf
     EndFor

If CardIdentifier is Null
   set Title to trim(CardTitle)
   If Title starts with '#'  then
     set SplitChar to  Index of (':' Or '-' Or ' ')
     set CardIdentifier to split(Title, SplitChar)
     set CardIsDesignItem to True
   EndIf
EndIf
```

Code Block 6.7: A simplified heuristic method to identify a Design Item on Trello and set its identifier

[80] To avoid obscurity, the unique identifiers referred to here are meant to be those user-facing ones, and not those implementation-related hidden fields.

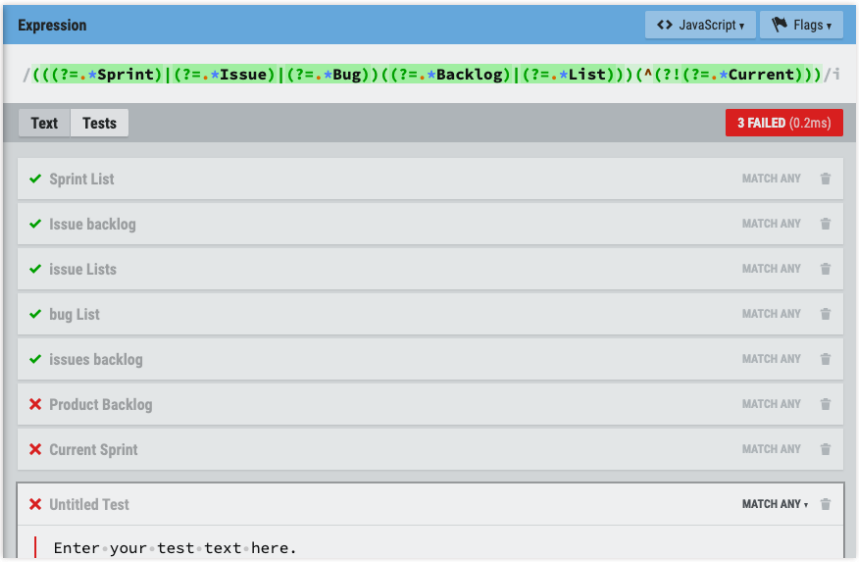
[81] An experimental fail-safe approach was also developed to identify actionable cards based on common terminologies (e.g., 'as a user', 'issue', etc.) when they do not have a unique identifier that declares them as Design Items. This was later dropped however for consistency reasons.

As a last remark, for dashboard products that do not enforce unique identifiers, users are expected to declare their Design Items by using alphanumeric identifiers that start with the hash symbol. Products that enforce user-facing unique identifiers readily simplify this step as no text processing is required to recognise and extract identifiers. They can be simply fetched through the product's APIs.

2. Common Naming for Actionable Lists

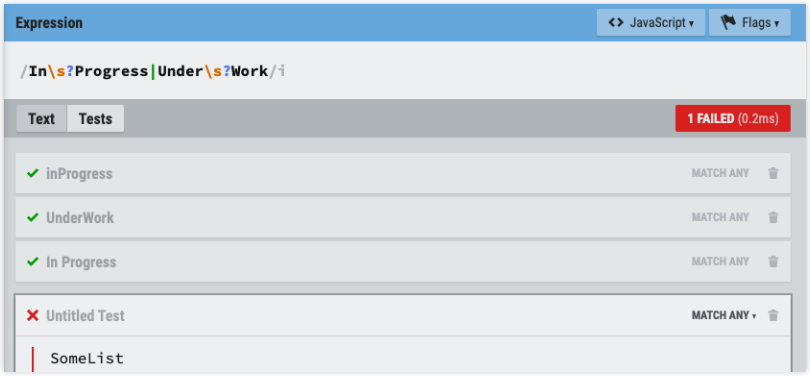
The Dashboard View Pane shown in earlier figures (see Figure 6.14) showcased different categories of Lists such as Product Backlog, Current Sprint, In Testing. Presentation of these lists was also stated to be dynamic based on a board's content. Dynamic identification of such lists is made possible through text processing heuristics implemented using regular expressions (regex). In a similar fashion to the approach used for the unique identifier, the present implementation is hardcoded but is intended to be exposed as user-configurable parameters in future work. As AgileInsight is aimed to be independent from any particular agile practice, it is necessary to rely on text processing heuristics to enable it to support a wider spectrum of users and practices. Ideally, a default implementation would encompass a bank of terminologies that are common among currently popular agile practices so that the tool would work out of the box for the largest number of users. Custom terminologies can then be accommodated easily through Vscode's user settings. Examples of the regular expressions implemented in AgileInsight to recognise actionable lists on a Trello board are presented in Code Block 6.8 and Code Block 6.9. They serve to illustrate how flexibility with efficient processing can be achieved by utilising regular expressions.

```
/(((?=.*Sprint)|(?=.*Issue)|(?=.*Bug))((?=.*Backlog)|(?=.*List)))(^(?!(?=.*Current)))/i
```



Code Block 6.8: Example of a regular expression as implemented by AgileInsight to identify actionable lists on Trello boards using common naming used by practicing community (top). The bottom row shows results of the regex being applied to a few tests.

```
/In\s?Progress|Under\s?Work/i
```



Code Block 6.9: An example of a simpler expression used to identify an in-progress list using common naming used by the practicing community.

The two text processing heuristic approaches discussed above for identifying design items and actionable lists are provided to illustrate the key operations that render the concept operational. The full implementation of Trello's Design Items Engine includes a few other text processing operations that extract secondary information and append it to a Design Item after one is recognised. This includes other categories of labels that are used as tagging, and other categories of identifiers that are used to link code items (covered in section 6.6). The engine is also responsible for augmenting the Design Item with auxiliary information such as short and full descriptions, status, due date, assigned users, and so on. In summary, each Design Item Engine can be written to best suit its dashboard product as long as the key requirement is met—which is that it exposes an abstract Design Item object with a unique identifier.

Closing Remark. In closing this section, it is important to note that none of the processing results of all operations discussed above are stored or cached. A Design Item must always be a true live copy of its corresponding object on the dashboard at any moment in time—even more so when it is about to perform an operation. This is fundamental to the correct operation of the tool, as objects in a dashboard product are accessible by multiple users and can be modified from multiple front-ends. The development approach and the design of the various components must thus be built around this central requirement.

6.5 Codebase Parsing and Modelling – Achieving a Language-Agnostic Solution

In section 2 of this chapter three components were introduced; Language Servers, Language Clients, and a Live File Parser. Also described was how the language servers and clients are the underlying infrastructure in the Vscode platform that made it possible for this research to build a language-agnostic file parser.

In principle, programming languages provide their services in Vscode through a common interface—a Language Server Protocol. Vscode is thus able to consume the services from all these language servers through a single common API. But given Vscode's modular architecture, it also exposes this API for other extensions to tap into. The end result is that extensions can utilise this common API to request information about any file, regardless of the language it is written in. As long as Vscode has the language support package for the underlying language—and that language's server is up and running[82]—then the request will ultimately be passed to that server to fulfil the request. To illustrate with an example, an extension can request for documentation or a definition of a particular element in the file, and the language's server of that file's language will eventually provide a response (if all goes well).

As for the purpose of this research, AgileInsight needs to get all structural elements composing a file; in other words, all the syntax elements making up a source file, but in their actual structural composition. It also needs some properties of each element such as their position in the document and what type of element it is. Fortunately, Vscode also needs a similar service to implement its Outline View (see Figure 6.18). To unify its implementation so that it works with any language server, Vscode came up with the Abstract Object Model, named `DocumentSymbol`, which is fulfilled by an API Interface called `DocumentSymbolProvider`. Through the use of this API, Vscode's Outline View is able to provide a hierarchical structure of any opened file in its editor, regardless of its underlying language. In fact, Vscode offers a

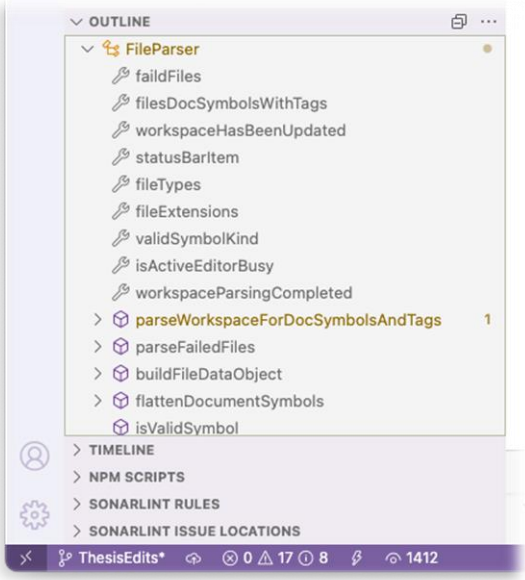


Figure 6.18: Vscode's Outline View displays the structural elements of a file opened in its text editor. It uses the `DocumentSymbolProvider` interface to achieve this irrespective of a file's language.

[82] Vscode will start a language server process automatically when needed, based on specific conditions set by the implementation of that server.

collection of Language Services through similar interfaces, from hovers, to item definition and completion. Figure 6.19 presents an illustration demonstrating a partial aspect of the architecture of language services API offered by Vscode and how communication takes place between the various servers and the service consuming parties.

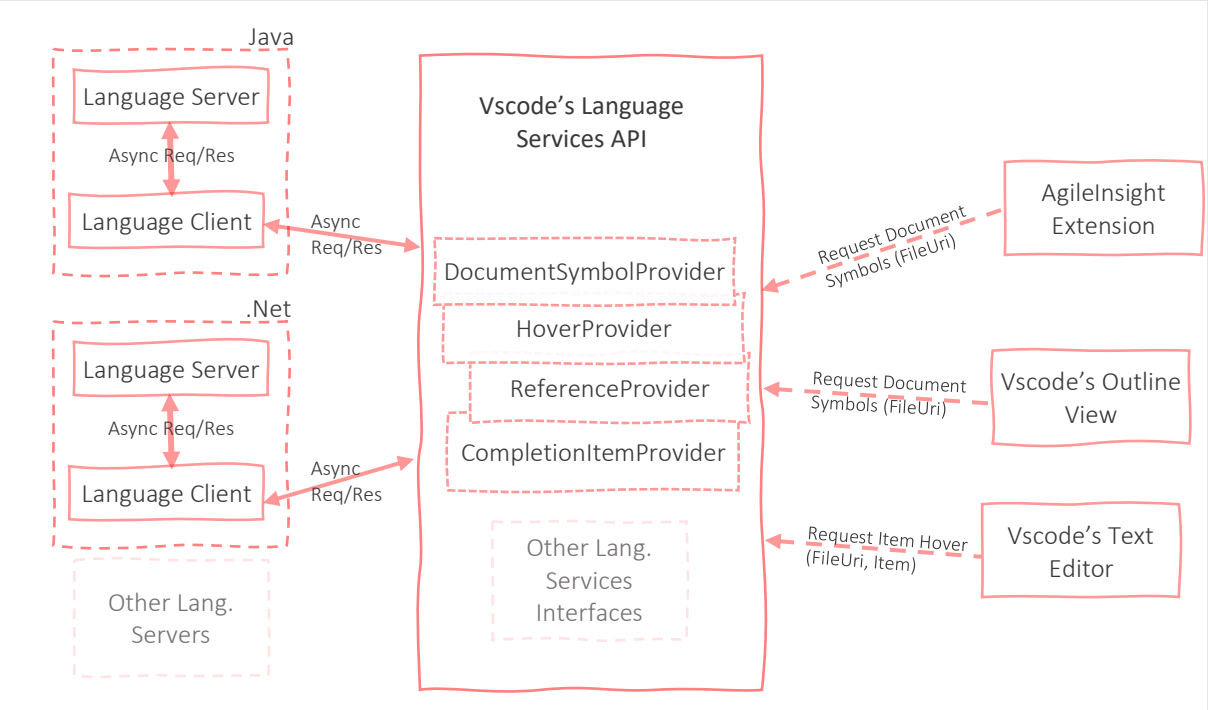


Figure 6.19: An illustration of how language services in Vscode are offered by LSP servers and eventually consumed by various parties. Providers are interfaces to which clients register their concrete implementations to Vscode to indicate that they offer that particular service[83].

The conclusion is that AgileInsight no longer needs to bother itself with what language a source code file is written in. If it can access its hierarchical structure through this abstract `DocumentSymbol` model—just like Vscode's Outline View is doing—then this solves the problem from its root. While in principle this appeared as an excellent and straightforward solution, achieving a practical implementation proved to be extremely challenging.

There was a combination of reasons contributing to this difficulty in implementing a practical solution:

1- Vscode does not expose the required API through direct method calls. Access to this API is exposed through the more sophisticated command execution approach, which requires knowledge of the exact command identifier string. This was further aggravated by inaccurate

[83] For more details see: https://code.visualstudio.com/api/language-extensions/programmatic-language-features

and outdated documentation of the `DocumentSymbolProvider` execution command, requiring long trial and error effort to arrive eventually at the correct usage[84].

2- While the `DocumentSymbolProvider` interface will in principle provide the Document Symbols of any file, the request will only be fulfilled if the language server is running. The server is only loaded when a client needs it. For practical matters, Vscode's text editor is the usual client and the pair of client-server of any language will normally be loaded when Vscode detects a file of that language. This was further complicated after discovering that loading a server-client pair had a different timing from one language provider to another—and consequently fulfilling the request for Document Symbols differed significantly in response times[85].

3- To provide an efficient and consistent experience to the user, AgileInsight needs to perform parsing of the source code files in a parallel and live manner—parallel processing is important given that many codebases will consist of hundreds to thousands of files and performing the operation in sequence will result in considerably increased wait time. The variable timing between different language servers in fulfilling the requests has resulted in significant complications to achieve the parallel parsing of files.

4- In theory, all language servers should implement the LSP protocol and adhere strictly to its design so that the exposed API has a universally consistent behaviour. While this was largely true, a few intricate discrepancies were discovered when testing the behaviour of a few languages[86]. The discrepancies were of minor effect though, and are certainly expected to be smoothed out over future releases as the implementation of that language's server gets perfected. It did mean, however, that AgileInsight had to (unnecessarily) account for these minor discrepancies to ensure a homogeneous behaviour for the user—until these imperfections are corrected.

Ultimately these challenges have been overcome through a series of experimentation, trial and error, and deeper study of LSP's intricacies in Vscode. For example, it was realised that the activation and

[84] At the time of development, return types were mixed up in Vscode's documentation between the `DocumentSymbolProvider` command and that of the `DocumentHighlightsProvider`, resulting in attempts that used the wrong command identifier and wrong casting of return type. Examination of the platform's open-source repository ultimately uncovered the error in documentation and led to discovering the right command identifier.

[85] For example, TypeScript's server (the native language of vscode) always had the fastest response time, typically fulfilling requests within milliseconds. Other languages, such as .NET's OmniSharp server and Python's server could take up to a few seconds under some circumstances.

[86] For example, TypeScript's server returns method names in this format `[getAddress]`, while Java's server appends the parameters returning methods as in `[getAddress(Person)]`. The OmniSharp server would yet append the namespace and class name, returning methods as `[namespace.ClassName.getAddress]`. The discrepancy appears to be non-agreement on 'default' behaviour, as the protocol does have variant methods to return signatures of methods if desired.

loading of a language's server-client pair—and consequently the response time of fulfilling a request—is significantly improved when using a workspace nature that is specific to that language. Nonetheless, an approach was ultimately arrived at that circumvented those difficulties and is able to parse any file (obtain its `DocumentSymbol` data object) as long as the Vscode environment has that file's language package added.

6.5.1 On Demand Live Parsing of Source Code

Section 6.2.5 has presented a descriptive overview of how the parsing is accomplished in AgileInsight. In this section a few selected technical highlights are further elaborated for interested readers.

1- Devising a Language-agnostic Parser

Until the time of writing, Vscode does not appear to offer any API that provides a similar functionality to the `DocumentSymbolProvider` interface but that is targeted for batch processing of files. The `DocumentSymbolProvider` is designed for one-off requests—and some implementor servers appear to expect the file to be opened in the text editor—and attempting to execute a large number of requests in parallel is certainly not something that the interface was designed for. While it is somewhat likely that, in future, Vscode will provide a matching API for handling batch files, an improvised solution had to be created to make AgileInsight a reality.

The solution came in the form of a background non-blocking process that works in two stages. In the first stage, it runs parallel asynchronous requests to attempt obtaining the `DocumentSymbol` model of all valid files that are in the current open workspace or project folder. Valid files are defined as those belonging to the common programming or scripting languages. For some of the languages that were tested like TypeScript, JavaScript, and Python, this stage will typically be able to process all the parsing required without opening any file (unless the language server was not loaded, in which case a single file would be opened). Those files that fail to provide their `DocumentSymbol` objects in the first attempt, are then dispatched to the second stage. In this stage, a process will loop over the failed files and apply a sequential request-and-wait approach to obtain their DocumentSymbol objects—performing a brute-force opening and closing of the file in the background if required—until no failed file is remaining. Since servers have a varying response time, the waiting time for each request-and-wait operation was calibrated over several experiments and was eventually set to 6 milliseconds. This was found to be enough for most language servers to provision their files' `DocumentSymbol` objects. Those that might take longer will eventually succeed in subsequent attempts. The second stage is thus applying a queue-based approach where each server is given 6 milliseconds to respond with a file's DocumentSymbol

object before the round is handed to the next file/server. The process keeps offering 6 milliseconds to each file/server until all files have been successfully processed. The entire two-stage process took less than 30 seconds on average for our real-world dataset, iTrust (more on this covered in Chapter 7). However, the process is found to vary largely based on the number of different languages involved, and their language servers implementation as explained earlier. Since the process is running in the background, its progress is made visible in Vscode's bottom Status Bar.

The implementation above has been tested with several collections of files and with projects containing mixed collections of source files from different languages. The programming (or scripting) languages involved in the tests included: TypeScript, JavaScript, HTML, XML, Java, C#, Python, PHP, JSP, CSS, Go, Swift, C++, and JSON. Table 6.1 shows example processing times of some real-word codebases that were tested. More examples are provided as part of AgileInsight's benchmarks in section 7.4.

Table 6.1: Example file-parsing processing times of real-world codebases featuring different languages

Codebase	Total Files	Type of Files	Average processing Time
iTrust	582	Java, JSP, html, JavaScript	23.6 sec.
ASP.NET Boilerplate	1401	C#, JavaScript, Typescript, CSS	1 mins, 51 sec.
ISTIO	1391	Go	3 mins., 11 sec.
AngularJs	849	JavaScript	36 sec.
Keras	661	Python	1 min, 15 sec.
Vscode	3280	Typescript, JavaScript, html, CSS	22 mins, 16 sec.

As has been described earlier, the parsing process is automatically launched upon the opening of a workspace or folder in Vscode to obtain a true live image of all present source files and their inner structural elements—now made possible by the DocumentSymbol model. Each time a file undergoes a change in its contents, or a new file is created, a controlled background process will ensure that the DocumentSymbol model of that file is kept up to date. This time, since the operation is running on a single file, the parsing happens almost instantaneously. Detection of such file changes is accomplished via a file system watcher that is applied to an open workspace, and which is implemented using Vscode's API interface `FileSystemWatcher`. Code Block 6.10 provides a code fragment illustrating how the interface could be put to use. With the above processes in place, a true live image of the entire codebase is maintained at any moment in time. This is crucial for the correct functioning of many AgileInsight features. As remarked elsewhere, AgileInsight does not store data in any intermediary files or databases, and all its functionalities rely on live data.

```
//extentionsToUse: array of valid file extensions
//workspaceUri: URI of user's open workspace
//FileParser: an AgileInsight class
const blob = `${folderToUse}/${extentionsToUse}`;

const watcher = vscode.workspace.createFileSystemWatcher(
    new vscode.RelativePattern(
        vscode.workspace.getWorkspaceFolder(workspaceUri), blob
    ),
    false, false, false
);

watcher.onDidChange((uri: vscode.Uri) => {
    if (uri.scheme === 'file') {
        FileParser.controlledFileDataUpdate(uri, true);
    }
});
watcher.onDidCreate((uri: vscode.Uri) => {
    if (uri.scheme === 'file') {
        FileParser.controlledFileDataUpdate(uri);
    }
});
watcher.onDidDelete((uri: vscode.Uri) => FileParser.removeFileData(uri));
```

Code Block 6.10: Code fragment showing example usage of Vscode's `FileSystemWatcher` interface.

2- Flattening the DocumentSymbol Model

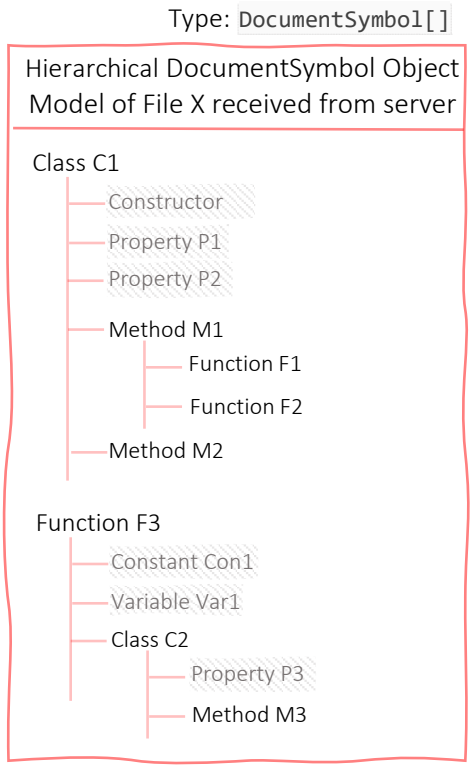
The resulting DocumentSymbol Model from the above operation has a hierarchical structure. For example, methods and properties will be nested as children of their class object—and similarly for any other nested elements, child objects will always be nested under their parent objects. This is perfect to represent the true structure of a file which is required for the visualisation part of this work. However, this is not ideal when searching operations are needed—which is required for this work. For instance, to implement the tagging mechanism (covered in the next two sections), which is a crucial part of the traceability solution offered by AgileInsight, the user must be able to quickly lookup a code item using an auto-complete feature. The nested hierarchy will require a recursive search operation which would considerably slow the process.

A simple flattening of this hierarchy would solve the issue. A sub-process to perform the flattening of each file's `DocumentSymbol` model was thus integrated into the parsing operation. This generates a dictionary-like `Map` object for each file with the name of a code item as the key. But as a file is likely to have multiple code items with the same name (e.g., similarly named methods under different classes), a long name is created and used as the key instead. With this configuration, the code item lookup operation is able to offer a responsive and almost instantaneous auto-completion capability (see section 6.8 for an example[87]). The long name is created by chaining the names of parent elements.

[87] A live demo is also available at the tool's profile page at: https://blaiski.github.io/agileInsight.page/

Figure 6.20 illustrates a hierarchical vs a flattened DocumentSymbol structure along with examples of the long name format.

Valid Symbols. While a file's `DocumentSymbol` object contains the complete structural elements of that source file, not all elements are relevant for the purpose of AgileInsight. Only those objects matching the Code Item definition (see Chapter 4) are considered by AgileInsight. In practice, this will typically encompass all named code blocks and exclude the likes of parameters and variables. Table 6.2 presents the full list of elements relevant to AgileInsight work, which are referred to as Valid Document Symbols. To maximise efficiency, AgileInsight only processes those valid symbols in its operations. Nonetheless, those can be easily set to be modified in Vscode's user settings.



Type: `Map<string, DocumentSymbol>`

Additional Flattened Version of File X's DocumentSymbol model created by AgileInsight	
Key (string)	**Value** (object)
`ClassC1`	Class C1
`ClassC1.MethodM1`	Method M1
`ClassC1.MethodM1.FunctionF1`	Function F1
`ClassC1.MethodM1.FunctionF2`	Function F2
`ClassC1.MethodM2`	Method M2
`FunctionF3`	Function F3
`FunctionF3.ClassC2`	Class C2
`FunctionF3.ClassC2.MethodM3`	Method M3

Figure 6.20: Language Servers produce a hierarchical DocumentSymbol as an array object (left). AgileInsight creates an additional flattened version—discarding non-relevant objects as greyed out in diagram (right). Both versions of the model are required for delivering AgileInsight functionalities. Note that while the example structure above might not align with some object oriented languages such as Java, it is nonetheless permissible and common in JavaScript.

Table 6.2: The set of Valid Document Symbols as appearing in AgileInsight implementation. The set is defined using the `enum` property `DocumentSymbol.kind` of each `DocumentSymbol` object.

`SymbolKind.Class`	`SymbolKind.Method`
`SymbolKind.Function`	`SymbolKind.Module`
`SymbolKind.Interface`	`SymbolKind.Struct`
`SymbolKind.Namespace`	`SymbolKind.File`

3- Extending the DocumentSymbol Model

To achieve the full functionalities required to build AgileInsight, a few features had to be introduced to the original DocumentSymbol object model of Vscode. These are as follows:

Introducing File Objects: The `DocumentSymbol` objects returned by language servers omit the file as an enclosing entity—only the inner elements are included. Accounting for the file itself as an entity is important, however, for the visualisation part of this work, as well as for fulfilling the tracelink feature (see the next section). Files need to be represented as a top-level code item entity, so a user can choose to bind an entire file if desired when creating a tracelink. To solve this, a new DocumentSymbol object is created by AgileInsight for each file, with the original `DocumentSymbol[]` object being enclosed then under this newly created object. In a similar fashion, DocumentSymbols are also created by AgileInsight for folders and packages, but these are only used for the visualisation part and are covered in due course in section 6.9.

Introducing Code Item Identifiers: The original DocumentSymbol object has a name property which is typically the naked name of an element without any extra appendments or signature part (in case of methods or functions). As was remarked earlier though, language servers will slightly vary in fulfilling this property. Moreover, to offer a proper tracelink capability, a full identifier is required so that any code item will have a unique identifier across a codebase that always leads to the exact same element. This is also important to ensure users are able to choose the right code item when creating a tracelink. To create such an identifier, it is a common convention in development communities to chain-append the enclosing parent elements with a dot separator that can extend up to the file and folder structure until reaching the root folder. AgileInsight introduces two versions of identifiers, a short version that extends to the file level, and a full version that extends to the root folder. Code Block 6.11 shows the format adopted by AgileInsight for the short and full identifiers of code items.

Code Item Name	`getCurrentNode()`
Code Item Short Identifier	`ScrumScene.java#ScrumScene.getCurrentNode()`
Code Item Full Identifier	`ScrumCity/src/nz/ac/aut/scrumcity/scene/ScrumScene.java#ScrumScene.getCurrentNode()`
Code Item Name	`Main(string[] args)'`
Code Item Short Identifier	`CSharpTest.cs#CalculatorProgram.Program.Main(string[] args)`
Code Item Full Identifier	`TestProject/src/CSharpTest.cs#CalculatorProgram.Program.Main(string[] args)`

Code Block 6.11: Code item identifiers as implemented in AgileInsight, with a Java example at top section and a C# example at lower section.

In addition to the two properties introduced above, a reference to the parent element is also added to the extended DocumentSymbol model, which is required for the traceability features—for example, to infer indirect parental relations.

4- A Complete Source Code Model

A number of objects were presented above as part of the process of building the source code model of an individual file. A containing object is now needed to manage those scattered objects under a single enclosing entity that represents the complete model of a file. This data object holds the extended `DocumentSymbol`, the flattened `DocumentSymbol`, in addition to a number of other supplementary pieces of information. Finally, to keep track of the model objects for a complete codebase, a simple `Map` object is introduced, where files' model objects are stored and indexed by their corresponding file URI. This global map becomes then the fundamental live data structure where the entire source code model of a codebase (or project) is maintained and kept up to date at all times.

To make better sense of the array of objects appearing above, a visual representation becomes imperative. A complete data structure is presented in Figure 2.21, which provides a depiction of the multitude of objects, and how they fit together to make the global map of the source code model of a codebase.

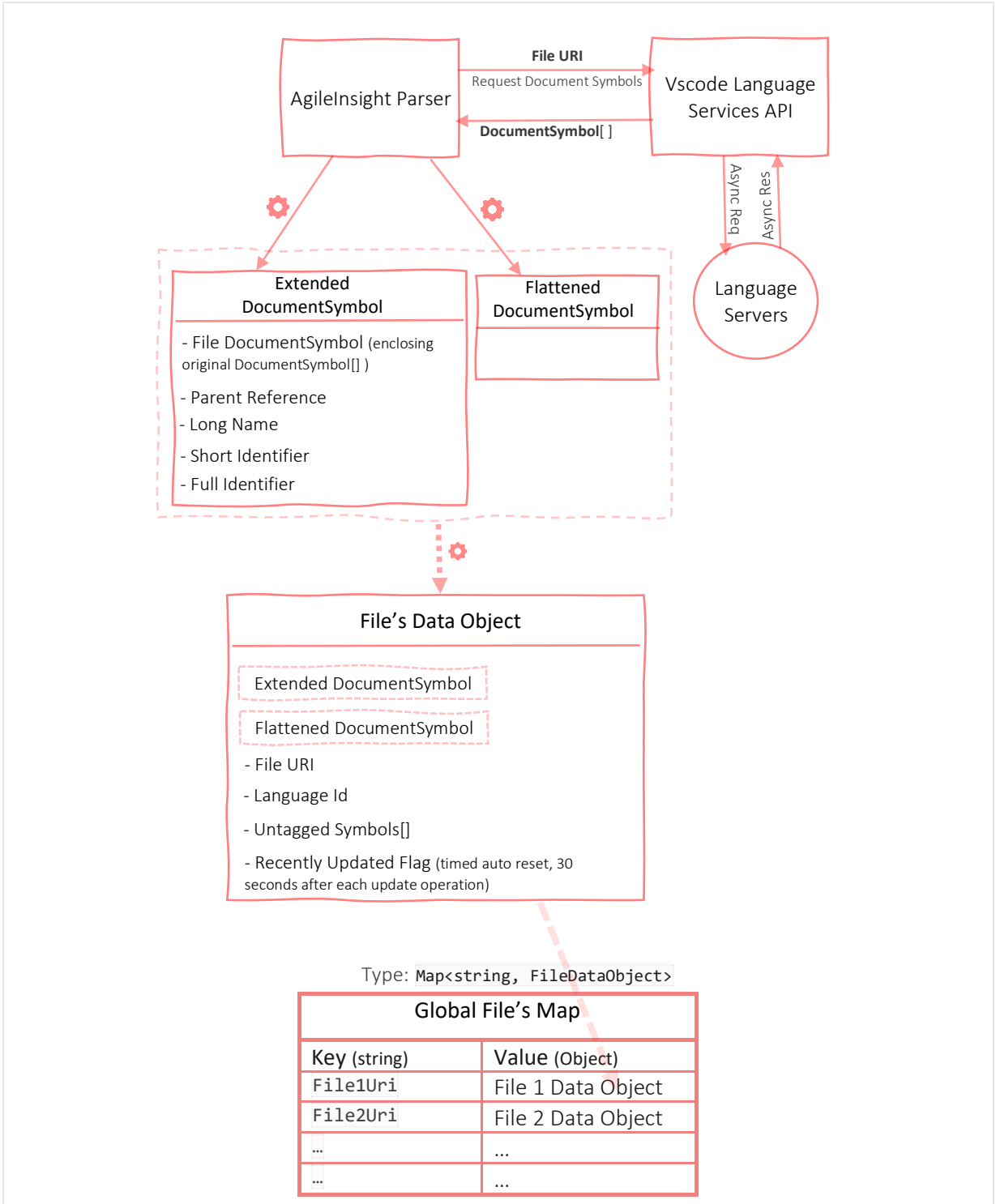


Figure 6.21: An overall diagram showing the data structure design used by AgileInsight to manage a full source code structural model of a complete codebase or project.

6.6 Towards Solving the Traceability Problem—Three-legged Automated Active Tracelinks

In a 2015 conference[88] on software traceability of safety-critical agile projects, internationally acknowledged traceability expert Jane Cleland-Huang closed her presentation with the following remark: "*If you are working in an agile project environment and you want to know the impact of a new user story, how much more agile can you get than giving them a tool that will just kind of tell them on the fly.*" Section 2.3 elaborated on modern approaches in software traceability and argued that most present techniques are remedial in nature—attempting to rediscover the tracelinks after development has already taken place, and link-knowledge has been lost (or as Cleland-Huang put it in her presentation, 'in … after-the-fact manner'). Section 4.5 then presented the concept of active tracelink collection, a novel mechanism to capture the tracelink-knowledge directly from the developer—the original source of this knowledge—before it is lost[89]. This section presents the technical implementation of this introduced mechanism—a proactive approach aimed to eliminate the problem from occurring in the first place, rather than devising a remedy after the issue has already occurred. It further eliminates the inaccuracy issue where human intervention is typically required to boost the confidence level in results obtained through automated rediscovery-based techniques (Gauerhof et al. 2022; Cleland-Huang and Habrat 2007; Mirakhorli and Cleland-Huang 2016; Bella et al. 2018). Once accurate links have been collected—as per the technique that this section shortly presents—a number of valuable benefits can then be offered to the user to support long-standing common questions faced in daily development activities—including being able to point to a user story (or any design item thereof) and reveal its exact impact on the final product—the source code. Needless to say, many of the work activities and components devised in the earlier sections were paving the way for this traceability solution to be realised.

A goal of this research has been to arrive at a practical solution that is relevant and in touch with the practicing community. The eventual solution arrived at here is strongly guided and informed by findings and lessons obtained across two phases of collaborative interview sessions with experts, as is broadly covered in Chapter 8. It is also highly aligned with and driven by the latest research and literature in the area of software traceability in agile projects. This has informed the underlying technologies selected, as well as the devised techniques. Special attention was given to make the footprint of the solution as small as possible (in terms of user disruption), and to integrate transparently in the way of working of agile-practicing teams. Following is an explanation of the individual parts that together come to make

[88] See: https://www.youtube.com/watch?v=1C3Di2ilNh0

[89] Due credit must also be acknowledged to the work of (Delater and Paech 2013a), which bears some similarity (in terms of intent, but different in concept and approach). Full discussion of their work has been in section 2.3.

up the overall solution. An overall diagram is presented at the end to provide a high-level view of those parts and how they interplay to achieve a full tracelink between a design item and its implemented code items. It is worthy to note that in discussing the technical implementation here, an understanding of the underlying concepts is implicitly presumed at this point. For that, the reader is kindly referred to Chapter 4 if necessary.

6.6.1 Automated Three-legged Tracelink Operation (TTO)

Substantial effort was expended to design a practical solution that fulfils the above described objective in a generic framework, but most importantly, one that can be implemented in the Vscode platform. The design arrived at consists of three steps that are automated to create a strong full-circle tracelink bond between the design artefacts (i.e., design items) on agile dashboards and their concrete source code implementation (i.e., code items) in the source code editor. The process involves as well automatically triggered commit operations to permanently record the tracelinks on source code repositories. Overall, a complete three-legged operation is performed in an automated way, without the user leaving their Vscode environment, and only requires user intervention at one step to provide input. To further reduce the operational footprint, input is collected via auto-complete and multi-select data options. Likely to be more familiar to users, this process is referred to as a 'tagging' operation in AgileInsight vocabulary. More importantly, all three steps of the tagging operation are executed as one autonomous process. The following is a description of the implementation of those three steps.

6.6.1.1 Tagging Design Artefacts on Dashboards

This method allows for the automatic tagging of design items such as a user story, or an issue on any of the boards that the user has authorised AgileInsight to access. The design item is linked with the specific code items that fulfil its intention. It accomplishes this via three data points:

1. Full identifier as a primary tracelink

The previous section introduced how each code item (e.g., a method, a function, or a whole file) is assigned a unique identifier during the parsing process, and how a full and a short version are created. The identifier provides a physical identity to each single code item that makes up the overall structural composition of a source code file, including the file itself. It becomes akin to an address to those well-bounded entities, so they can be interacted with as individual units, pointed to, navigated to, and even visualised. Effectively, it gives the ability to physically treat these units as the fundamental building blocks that realise a functioning intention of a design item.

With that view in mind, a practical tracelink concept starts to emerge; a small actionable design item is transformed by a developer into a limited number of individual and identifiable code item units. A small block of original intention on one side, producing few small blocks of concrete implementation on the other. All that is required to create the tracelink is to draw the link lines—to bind the design item block with the full identifiers of its code items. The developer is the person who best knows those exact code items at the time they are produced, but the knowledge can become quickly lost if not captured immediately.

Thus, a mechanism has been created to capture this information from the developer before they conclude a development session. The exact mechanism is covered in the next section, but what is important to note at this point is that once the developer has declared (or confirmed) a bond between code items and their original design item, then the full identifiers of those code items must be inserted and attached to the design item that the developer is working on or has just completed.

How the identifiers are inserted or attached to a design item card on a dashboard is of secondary importance. As has been discussed earlier, almost all agile dashboards have a well-designed and mature data structure format for managing their object models. It is a matter of practicality of finding the best property or field under which to store those full identifier data points. In future, such dedicated properties or fields might well be readily offered by a dashboard's object model, or some might allow the creation of custom properties or fields[90].

As for the current implementation that AgileInsight integrated for Trello, full identifiers are appended to the lower section of a card's description field. It was the best option arrived at that offered the required flexibility after experimenting with few other properties and custom fields. The identifiers are inserted in a complete automated manner, including being automatically updated when required. Figure 6.22 shows an example of full identifiers attached to a design item card.

2. Permanent Link to enhance accessibility

The full identifier above is the primary data point that binds a design item to its code items. However, they are no more than long paths in textual form. It is helpful to provide users with a more accessible and direct navigation mechanism to enable them to directly navigate to the actual code item to inspect it. GitHub, as described earlier, provides a helpful feature called 'Permalink', which is a permanent link

[90] It is worth mentioning that during our Trello card integration work, it was noticed that a "PowerUp" feature did exist that allowed Trello users to manually link a card to their GitHub issues and commits. (PowerUps are small extensions that users can add to their Trello). This serves to demonstrate users interest in such functionality in the real world, endorsing our approach.

to the specific location of a code item in source code in a specific version of a file. It is a relatively recent feature that is highly useful and utilised by developer communities.

When the user is connected to GitHub (see section 6.2.6), this useful but optional data point is also created and attached next to a full identifier. It enables a user who is inspecting the design item card to directly navigate to that code item by clicking on the link. Figure 6.22 shows an example of a GitHub permanent link attached to an identifier and the result of clicking on it.

While Vscode's GitHub extension does not readily expose this feature for consumption by other extensions, the extension is open source and it was possible to tap into this functionality to reuse it.

3. Short identifier for better readability

In addition to a long-formatted string path, a more readable and shorter version of the full identifier can be attached to the design item's card to offer the user a mechanism by which they can readily spot linked code items via their familiar names. The idea is that full identifiers, given their inconvenient long nature, are not supposed to be the first contact for users to come across. Users should instead see the shorter version of the identifier—which highlights code item names that are more familiar and readily recognisable to a developer—as the readily exposed tracelink data points. This is expected to be sufficient in most situations to quickly answer a developer's question and they can move on to their more important actual work task. The full identifier should be relegated to some less-exposed section of a design item, probably tucked away where it will not create unnecessary visual clutter on a card, and is only expected to be of use when a deeper analysis is required (e.g., to inspect the source code at the specific point of time when a link was created). If possible, the field where full identifiers are stored should feature restricted access to discourage normal users from accidentally altering or deleting it. After all, it is the point where the actual bond is created between the two worlds.

For Trello's implementation, AgileInsight currently utilises a card label to display this more recognisable short version of a code item tracelink. With proper coding practice, a design item should normally result in a limited number of code item links (high cohesion), and consequently, a card would ideally have very few code item identifier labels displayed on it. For convenience, AgileInsight refers to those short identifier tracelinks as code item tags, or CI tags for short. Figure 6.22 shows a card from a real-world dataset with code item tags appearing as labels.

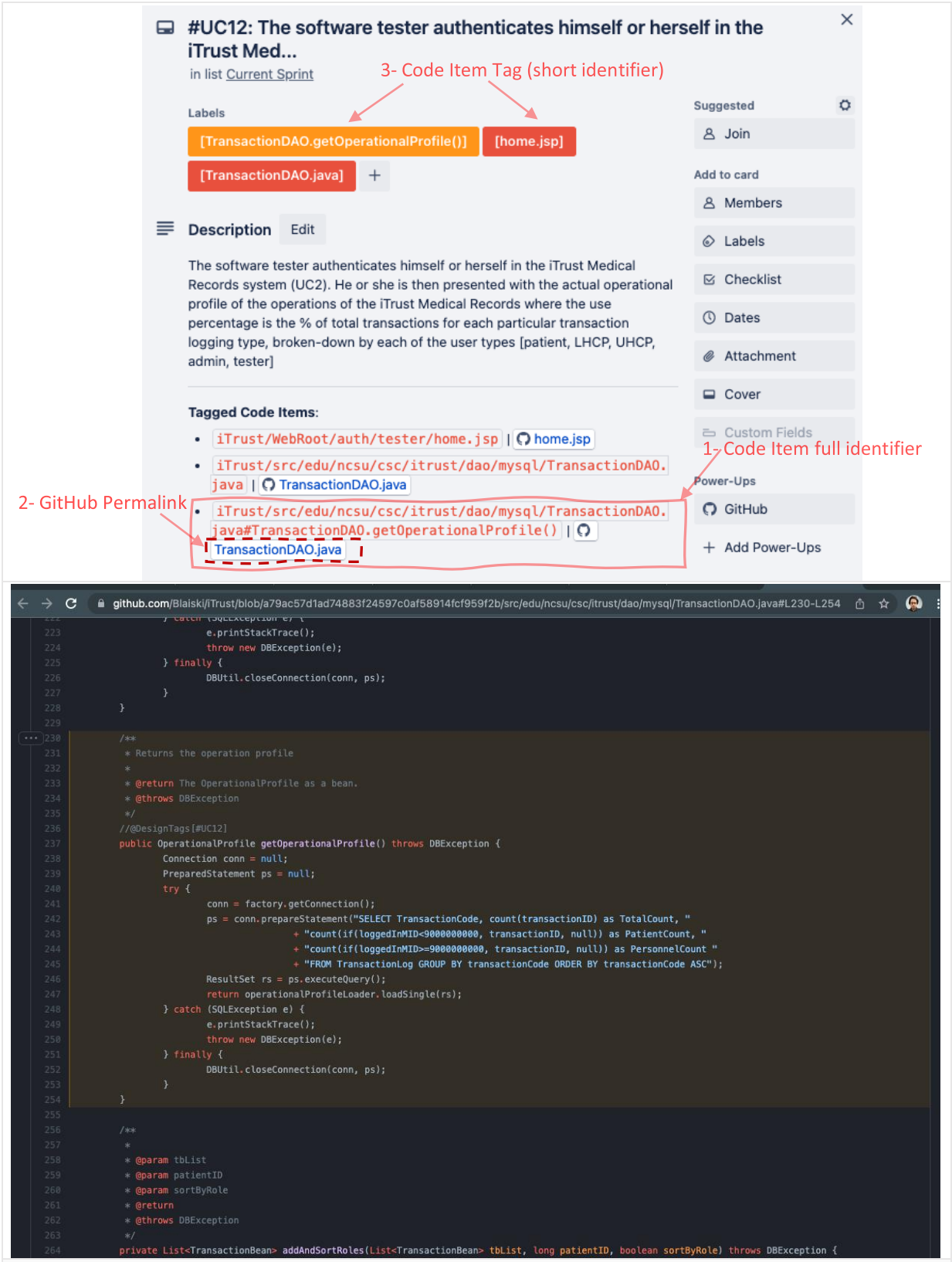


Figure 6.22: Top: A design item card on Trello showing the three data points of code item tracelinks. The full identifier is one that creates the actual bind to a code item, while the other two provides better accessibility. The color of the label indicates whether the code item tag is a file-level (red) or an inner level (orange). Bottom: By clicking the GitHub Permalink, the user can navigate to the code item block in the specific version of the source file where the tracelink was last updated. A highlighted background clearly marks the line range of the linked code item.

Remark. It is worth pointing out that as long as the user's Vscode environment has a remote repository configured (e.g., GitHub) then as part of the tagging process, the full workflow of a git commit operation (commit, stage, and push) would be taken care of by AgileInsight in a fully automated way. This is possible thanks to the special commands and the API methods that Vscode's git extension exposes for other extensions to re-use (see section 6.5 and 6.8 for more details).

6.6.1.2 Tagging Code Artefacts in Source Code

This is the second leg of the three-legged automated tracelink operation introduced by AgileInsight. It augments the previous step to create tracelinks in the source code itself, offering a different—and hopefully highly useful—view to software traceability.

1. Attaching Design Item Identifier to Code Items

Code item tags offer accessible tracelinks for design items as displayed on the dashboard—offering a design-to-implementation perspective. To provide developers with the other corresponding view—that is, tracelinks showing an implementation-to-design perspective—design item identifiers come into play. As described in section 6.4.3, design item identifiers allow for the unique identification of those original requirements across a user's board. In a similar fashion to the role identifiers played for code items, the design item identifiers enable the developer to bring those original and fundamental building blocks of software design more visibly to the surface, where they can be handled and interacted with as tangible entities.

In similar fashion, design item identifiers, or DI tags for short, are inserted and attached to the specific code items that implement them. The tagging process will automatically locate the concerned code items (in the developer's open workspace or project folder) and append the DI tag to each. A special but simple comment block has been introduced to attach design item tags to their code items—an action that is completely automated, including auto-updating if necessary. A DI comment block (which hosts DI tags) is always attached immediately before a code item's block starts (e.g., at the first line preceding a class or a function definition). If the code item being tagged is a whole file, then the DI comment block is placed as the very first line of that source file. Figure 6.23 presents the format of a DI comment block.

```
//@DesignTags[#Issue204, #Issue67]
1      2              3     4
```

Figure 6.23: The format of the comment block used to keep Design Item tags attached to a code item. 1: comment character as per a programming language syntax, 2: A standard fixed label used to identify the DI comment block, 3: comma as a separator when required, 4: a design item tag. A square bracket is used to enclose design item tag/s.

Since DI tags are mere alphanumeric string of characters, a hover mechanism has been implemented to allow users to conveniently access the most relevant information by hovering over a DI tag. This introduces traceability (and design) information to the developer in-context, right in their source code editor, and without the need to switch to another tool or application. To make the most of Vscode features, our building of AgileInsight experimented with actionable hover commands to offer the developer more advantages; to perform quick actions on their dashboard cards without leaving their source code—an example of secondary fruitful outcomes of integrating agile dashboards into the development environment. Figure 6.24 demonstrates design item tags attached to their code items, along with examples of their 'Peek' information and quick actions that are exposed upon a hover event.

While AgileInsight had to devise a special comment block to host the design item identifiers, if the introduced concept finds acceptance and adoption among the development community, then design item identifiers (or DI tags) might well be integrated in future as official annotations in a programming language.

```
    # Do a CD and load the contents.  If there is no directory name, skip
    # the CD.
    # @DesignTags[#Issue204, #Issue67]
    def load_dir(self, dir):
        "Change to a directory and load its content into the listing window."
        if dir:
            try:
                self.conn.cwd(dir)
```

```
//@DesignTags[#US367, #Bug267, #Bug345]
async showInputBox<P extends InputBoxParameters>({ title, step, totalSteps, value, pro
    const disposables: Disposable[] = [];
    try {
        return await new Promise<string | (P extends { buttons: (infer I)[] } ? I : ne
            const input = window.createInputBox();
            input.title = title;
```

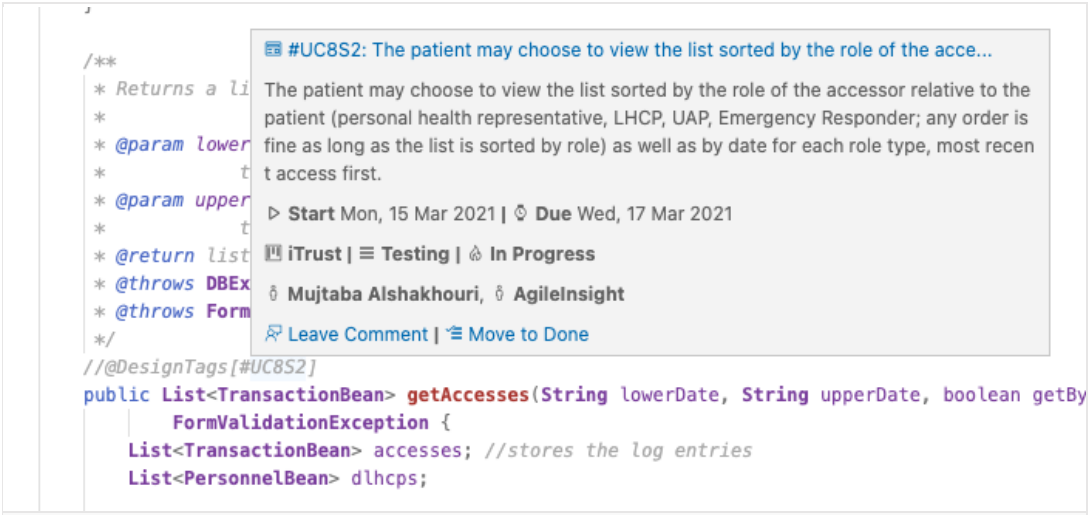


Figure 6.24: In first two rows, examples of Design Item Tags appear attached to Code Item blocks in Python and TypeScript source code, respectively. In bottom block, an example of a hover Peek is illustrated, featuring relevant details of the design item and some quick actions (blue-colored).

6.6.1.3 Recording Tags in Source Code Repository

To complete the full circle of a tracelink bound, the three-legged tracelink operation performs one last step; to create a permanent traceability record in the source code repository. This includes recording the DI tags that were attached to the code items in the source code by creating a commit operation, and placing a corresponding tag in the commit message. The following points explain the details of how this last leg is implemented.

1. Intercepting a traditional commit to create tracelinks

Source code repositories have become indispensable in software development. With the design item tags attached to code items in the previous step, it is only a logical next step to ensure these tags are permanently reflected into the source code repository. However, rather than creating an artificial commit operation, AgileInsight attempts to transparently integrate this action into a typical commit operation that a developer would normally perform. The design of this integration is a key part of the traceability concept introduced in this work. It scaffolds on the presumption that developers would typically pick (or be assigned) one design item at a time, and work towards fulfilling its implementation in one or a few limited sessions—depending on its complexity. Few commit operations would normally then permeate through those sessions, most likely at every occasion a developer perfects a few parts. Or, in other words, every time the developer contributes some new code items—or makes changes to existing ones. It is at these stages that AgileInsight presumes a tracelink operation should occur in order to preserve the developer's knowledge of the code changes they have contributed—i.e., to capture those valuable tracelinks.

Thus, rather than forcing an artificial commit transaction, AgileInsight simply acts in the middle of that natural workflow and intercepts it in an as-transparent-as-possible manner to incorporate the tracelinks into that git commit operation—which a developer is expected to undertake either way. In fact, since the whole tagging operation is one autonomous process, the previous two steps are already incorporated into this interception. AgileInsight offers three methods for this operation, that each were designed to fit into a particular way of working for developers—especially those practicing an agile methodology. Section 6.7 provides details on the implementation and working of each of these methods.

The important aspect that is relevant here, is that after attaching DI tags to code items in the source code, AgileInsight will automatically perform a commit operation to push the changes to the repository. It will also place the identifier of the design item concerned into the commit message to create a visible traceability record at the repository side[91]. Figure 6.25 illustrates these two aspects of maintaining a traceability historical record on a repository with an example from a real-world dataset.

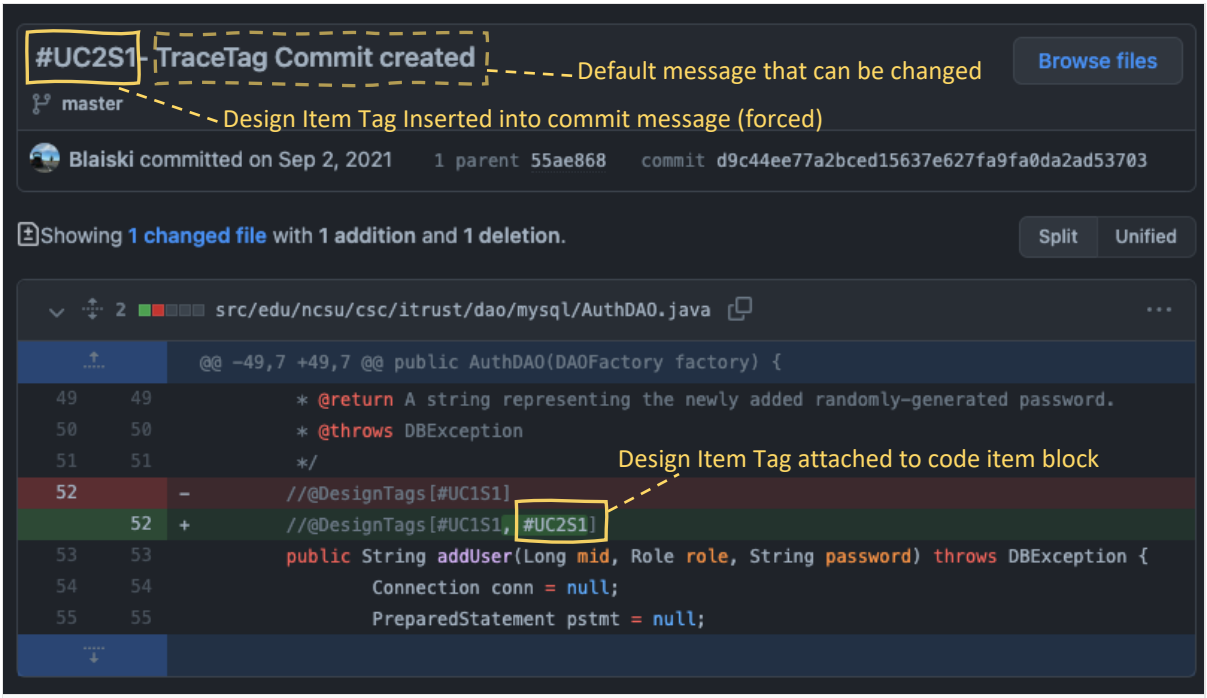


Figure 6.25: An example illustrating the results on source code remote repository after AgileInsight has completed the automated commit operation, creating a permanent tracelink record binding a design item to its related code item.

[91] As noted in related literature discussion, placing a reference to related GitHub issues has recently become a common practice in development communities. AgileInsight has picked up inspiration from this practice.

2. Option to amend previous commit

As well as intercepting a typical commit operation to capture and record the tracelinks, AgileInsight also provides a commit amend operation to allow developers to integrate the tracelinks as part of the last commit operation. While commit amend operations are not popular practice, this flexibility could offer developers the ability to retract commits that failed to declare implementation tracelinks. This amend feature is offered by Vscode's git API and AgileInsight simply repurposes it.

An Overall Picture. A brief summary of the above discussions now concludes this section. In brief, AgileInsight introduces a new mechanism to capture traceability knowledge from developers while they are actively working. It is guided by feedback from both developers and recent research, and is designed to be transparent with minimal operational impact. Once initiated, the process triggers three fully automated steps to create a strong tracelink bond between each original requirement and its actual code implementation. It tags the design item card on the developer's board, it tags the concerned code items in the source code editor, and it takes care of the complete commit workflow for the developer, ensuring a permanent traceability record is created on the repository. Figure 6.26 illustrates this process with a diagram as a summary depiction. The next section illustrates the different ways in which the tagging operation can be initiated by the developer and how it fits within their way of working.

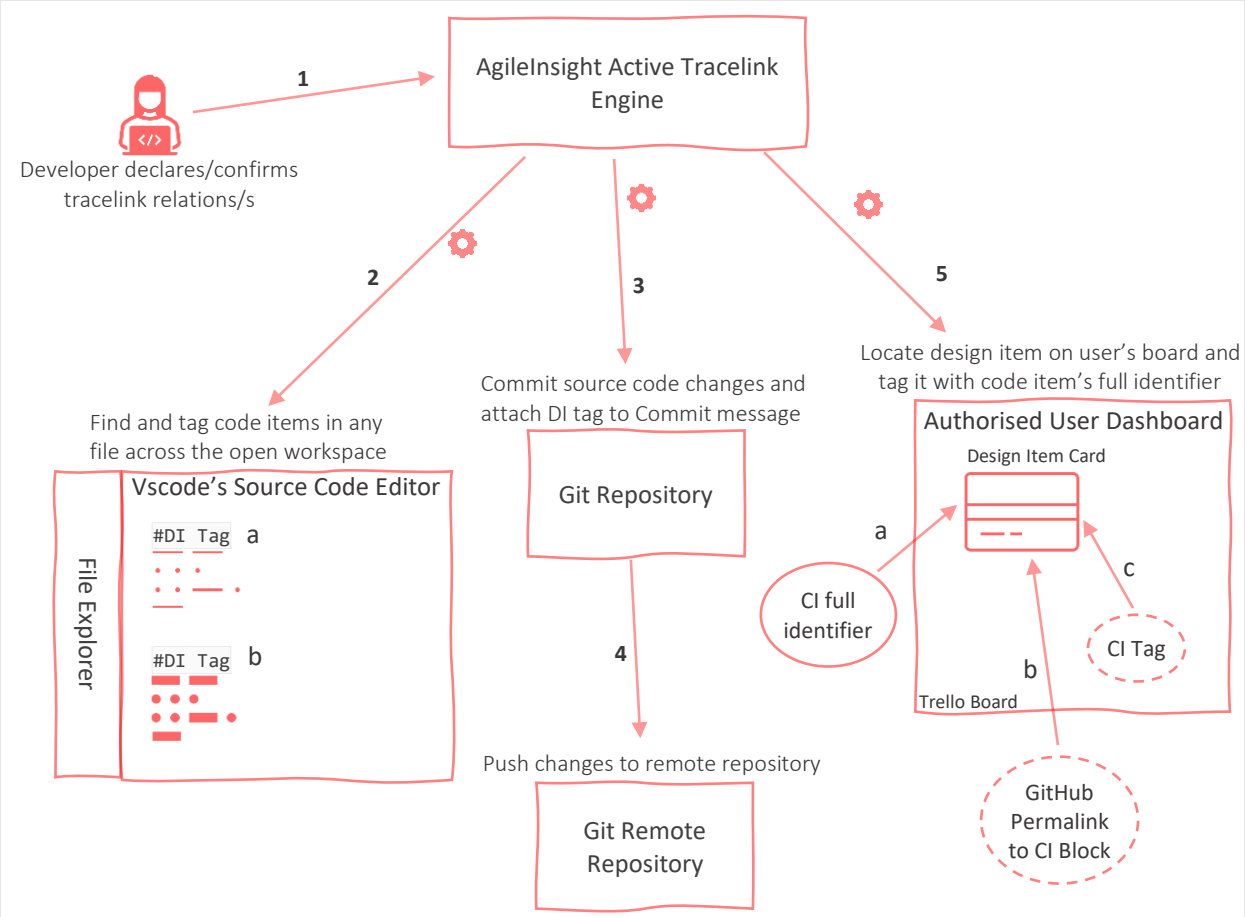


Figure 6.26: A diagram summarising the automated three-legged tracelink operation as introduced and implemented by AgileInsight

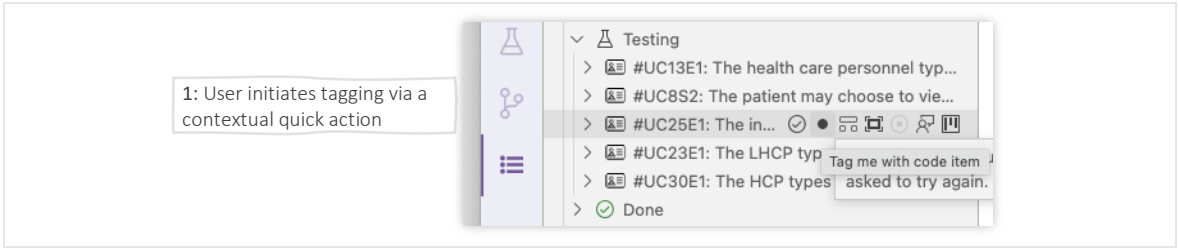
6.7 Techniques for Capturing the Tracelink Knowledge from the Original Developer

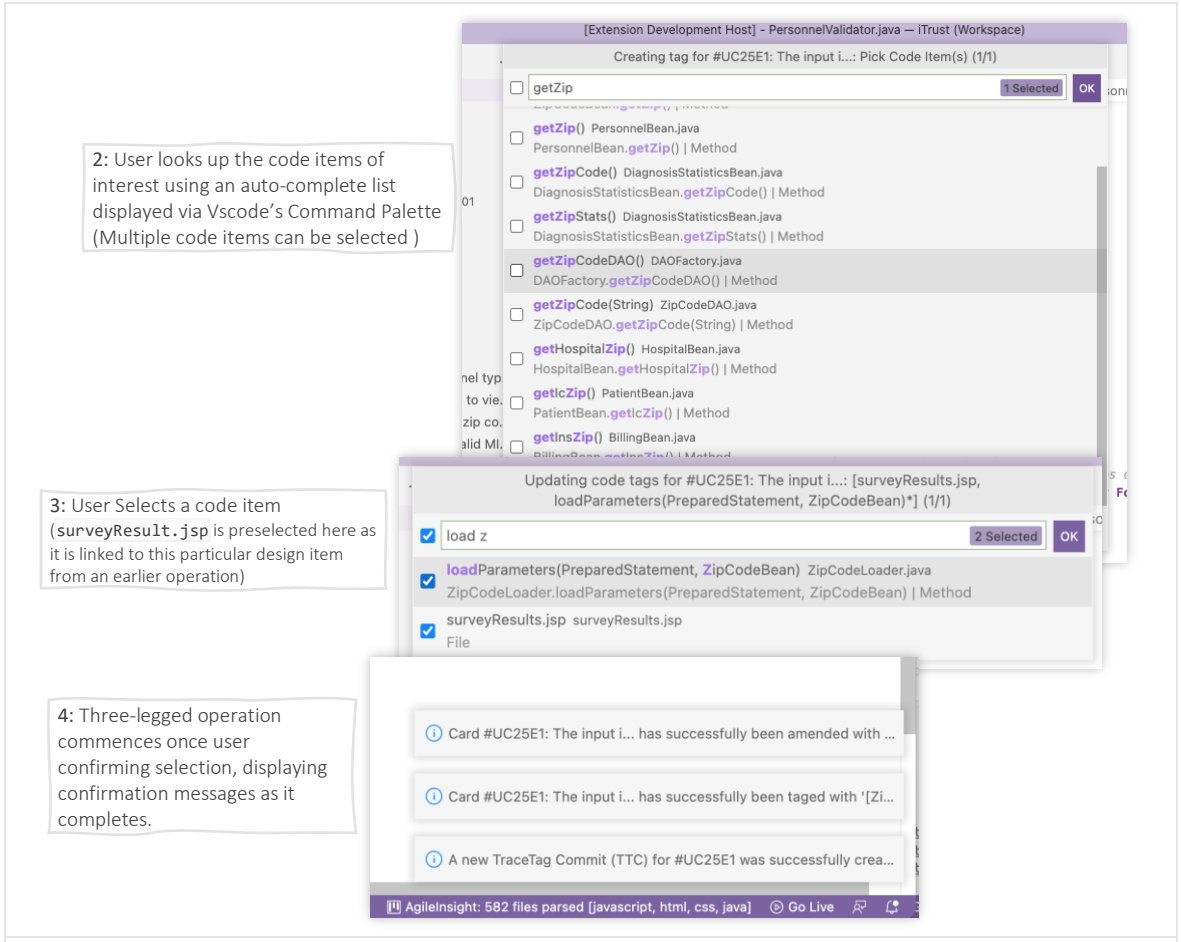
After collaborative sessions with expert developers (see the next chapter), three primary methods were deliberated to capture the specific code items they produced (or added changes to) in response to completing a design item; in other words, to freshly capture the tracelink knowledge from the original developer before it becomes lost. These three methods are presented individually below.

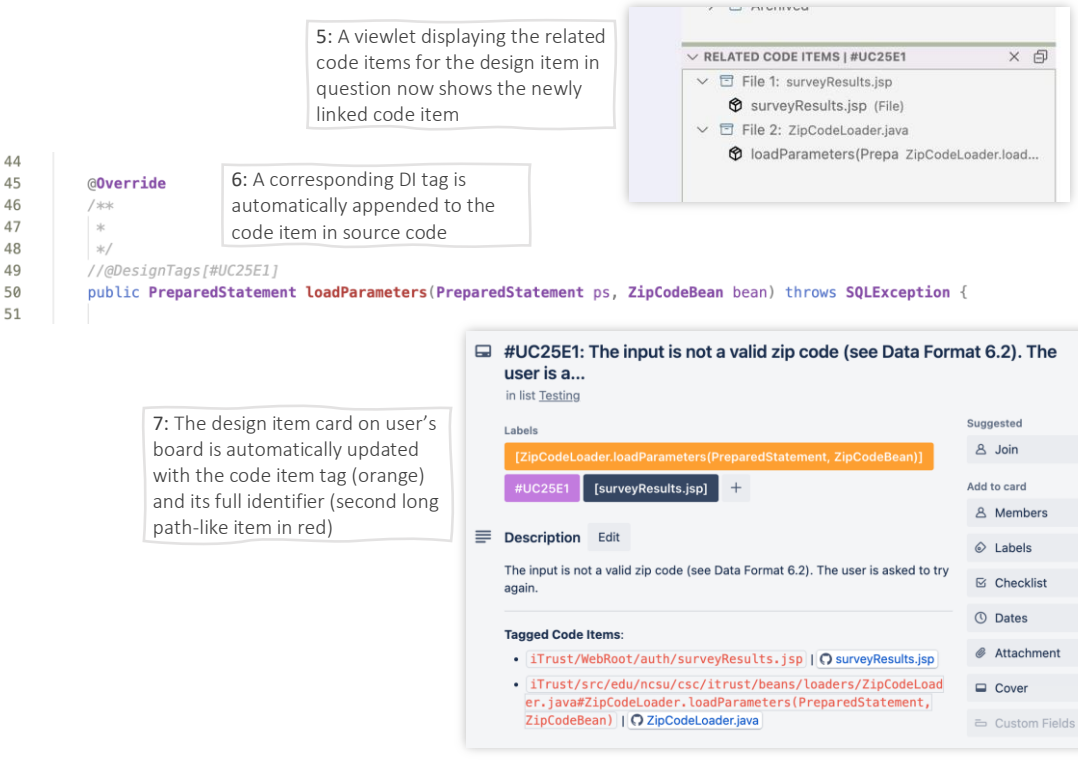
6.7.1 DI Initiated—Tagging a Design Item with its Related Code Items

This is the most straightforward method to initiate the tagging operation. After a developer concludes a working session on a particular design item and decides that it is time to commit their changes, then instead of performing the routine workflow of a commit operation, they are encouraged instead to simply point to the design item they were working on and invoke the tagging command. Through the AgileInsight Explorer (see Figure 6.11), the design items are always readily available on the left side of the source code editor. The command may also be invoked directly through a shortcut for those who are not inclined to use their pointer devices. Once the action is invoked, the developer is presented with Vscode's central palette where they can quickly lookup and select the specific code items that they have worked on as part of their implementation of the design item in question. An auto-complete and multi-select mechanism helps to expedite this process with the intention of making it more convenient. Once the code items of interest are selected and confirmed, the tagging operation commences, performing automatically and in sequence all of the three legs of the process described in previous section. The developer is offered the chance to edit the commit message if desired, but the DI tag is enforced. Figure 6.27 demonstrates this process and highlights the limited action required from the user to initiate it.

While ideally this method should be used each time a developer concludes a session—whether a task is fully completed or just partially—it is possible to launch the tagging process in a retrospective manner, where development has been already completed at an earlier time.




2: User looks up the code items of interest using an auto-complete list displayed via Vscode's Command Palette (Multiple code items can be selected)
[Extension Development Host] - PersonnelValidator.java — iTrust (Workspace)
Creating tag for #UC25E1: The input i...: Pick Code Item(s) (1/1)
getZip
1 Selected
OK
getZip() PersonnelBean.java
PersonnelBean.getZip() | Method
getZipCode() DiagnosisStatisticsBean.java
DiagnosisStatisticsBean.getZipCode() | Method
getZipStats() DiagnosisStatisticsBean.java
DiagnosisStatisticsBean.getZipStats() | Method
getZipCodeDAO() DAOFactory.java
DAOFactory.getZipCodeDAO() | Method
getZipCode(String) ZipCodeDAO.java
ZipCodeDAO.getZipCode(String) | Method
getHospitalZip() HospitalBean.java
HospitalBean.getHospitalZip() | Method
getIcZip() PatientBean.java
PatientBean.getIcZip() | Method
getInsZip() BillingBean.java
3: User Selects a code item (surveyResult.jsp is preselected here as it is linked to this particular design item from an earlier operation)
Updating code tags for #UC25E1: The input i...: [surveyResults.jsp, loadParameters(PreparedStatement, ZipCodeBean)*] (1/1)
load z
2 Selected
OK
loadParameters(PreparedStatement, ZipCodeBean) ZipCodeLoader.java
ZipCodeLoader.loadParameters(PreparedStatement, ZipCodeBean) | Method
surveyResults.jsp surveyResults.jsp
File
4: Three-legged operation commences once user confirming selection, displaying confirmation messages as it completes.
Card #UC25E1: The input i... has successfully been amended with ...
Card #UC25E1: The input i... has successfully been taged with '[Zi...
A new TraceTag Commit (TTC) for #UC25E1 was successfully crea...
AgileInsight: 582 files parsed [javascript, html, css, java]
Go Live


5: A viewlet displaying the related code items for the design item in question now shows the newly linked code item
RELATED CODE ITEMS | #UC25E1
File 1: surveyResults.jsp
surveyResults.jsp (File)
File 2: ZipCodeLoader.java
loadParameters(Prepa ZipCodeLoader.load...
6: A corresponding DI tag is automatically appended to the code item in source code
44
45 @Override
46 /**
47 *
48 */
49 //@DesignTags[#UC25E1]
50 public PreparedStatement loadParameters(PreparedStatement ps, ZipCodeBean bean) throws SQLException {
51
7: The design item card on user's board is automatically updated with the code item tag (orange) and its full identifier (second long path-like item in red)
#UC25E1: The input is not a valid zip code (see Data Format 6.2). The user is a...
in list Testing
Labels
[ZipCodeLoader.loadParameters(PreparedStatement, ZipCodeBean)]
#UC25E1
[surveyResults.jsp]
Description
Edit
The input is not a valid zip code (see Data Format 6.2). The user is asked to try again.
Tagged Code Items:
iTrust/WebRoot/auth/surveyResults.jsp | surveyResults.jsp
iTrust/src/edu/ncsu/csc/itrust/beans/loaders/ZipCodeLoader.java#ZipCodeLoader.loadParameters(PreparedStatement, ZipCodeBean) | ZipCodeLoader.java
Suggested
Join
Add to card
Members
Labels
Checklist
Dates
Attachment
Cover
Custom Fields

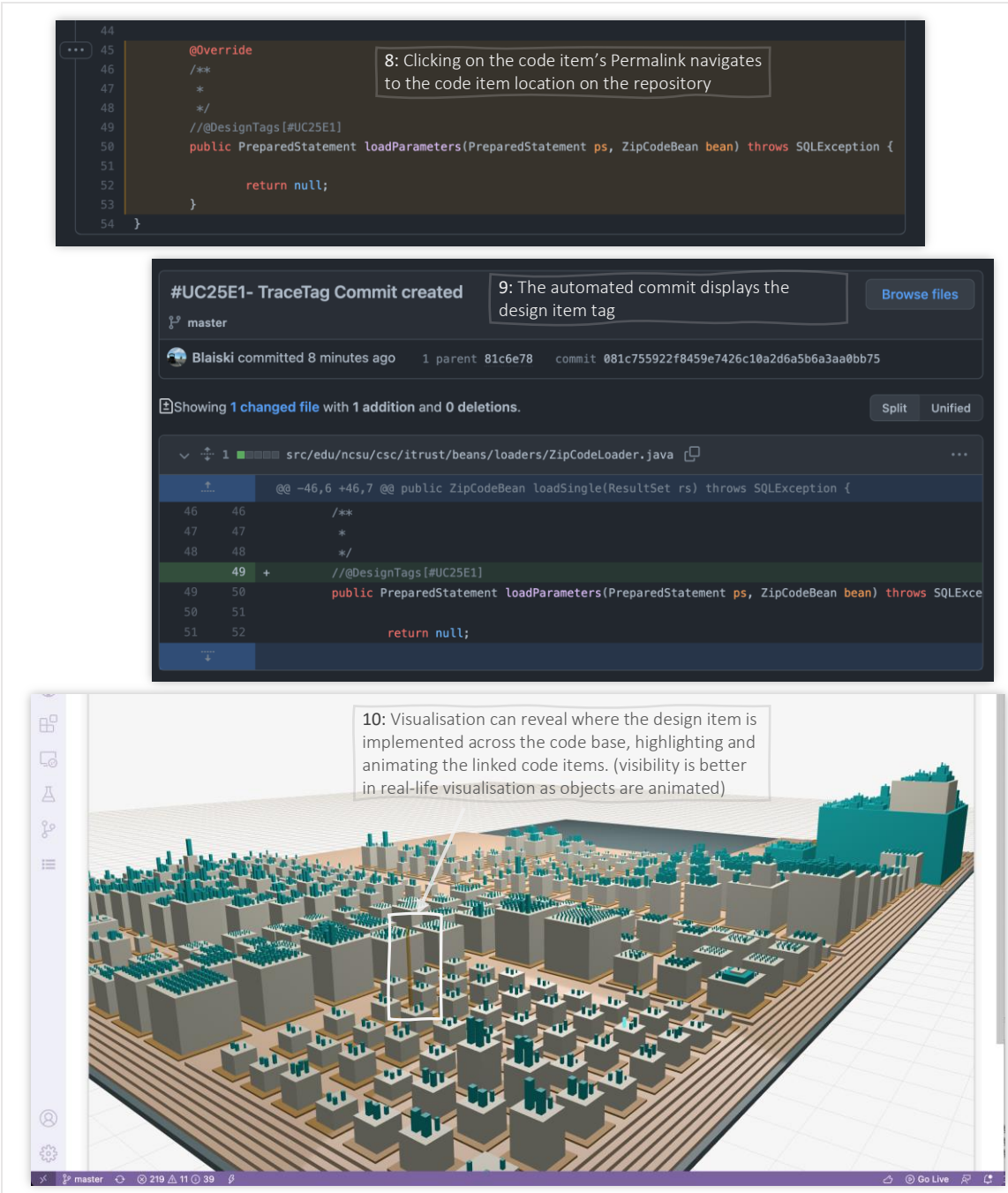


Figure 6.27: A demonstration of the DI initiated tagging process, showing its resulting automated three-legged tracelinks in source code, repository, and user's dashboard. Step 5 and step 10 show some immediate usage benefits.

6.7.2 On-Commit Initiated—Reminder Approach

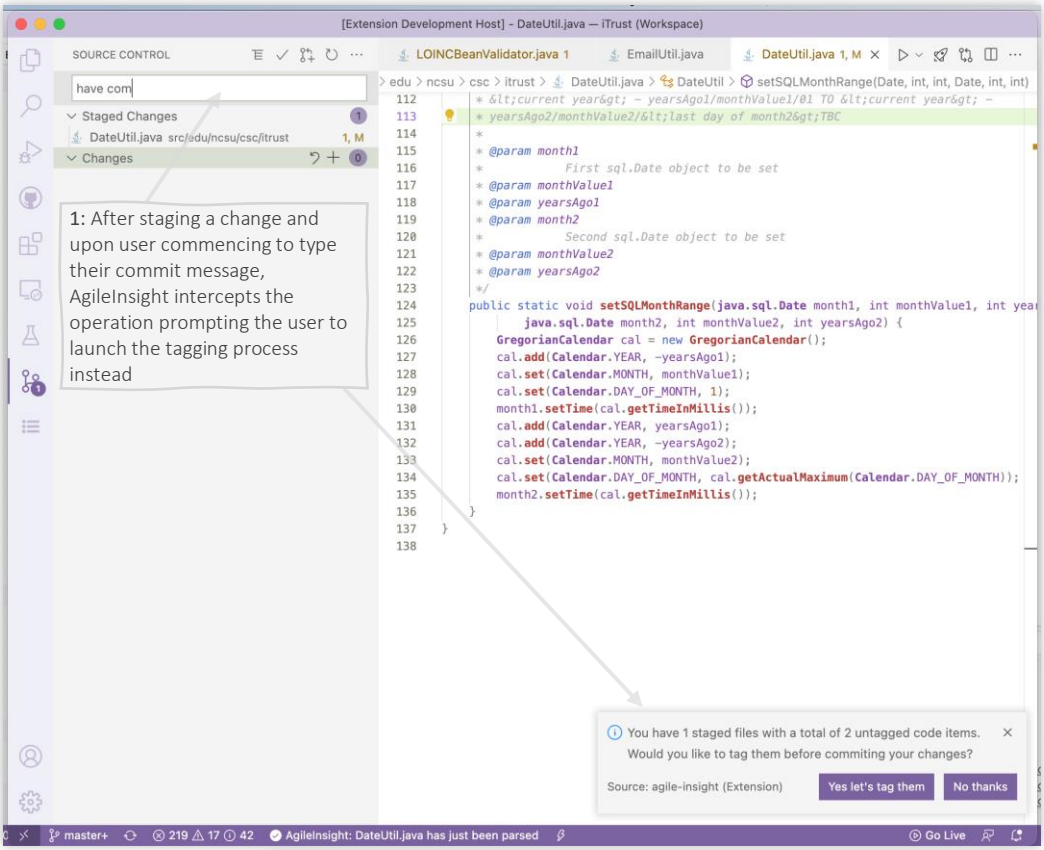
This method is a natural development from the previous one, directly inspired by feedback from the expert developers participating in the collaborative sessions. A strong preference was expressed to have the tagging process triggered when the developer is about to commit their changes. This was also seen as a good approach to encourage developers to develop the habit of—and gradually adopt—the tagging functionality.

The method works once the developer heads to the git module of Vscode and goes to perform the traditional commit workflow. If AgileInsight detects that their staged changes contain contributed (or modified) code items, and that these are not tagged, then as soon as the developer proceeds to interact with the git commit message box, AgileInsight will display a prompt to remind them if they wish to tag those code items. If the user accepts the prompt, they are presented with the list of those recently contributed code items, for which they can then initiate the tagging process. The initiation follows the same workflow described in earlier method[92]. Figure 6.28 illustrates the workflow of this method in pictures. The technical details of how the untagged code items in the staged changes are detected by AgileInsight are presented in section 6.8.

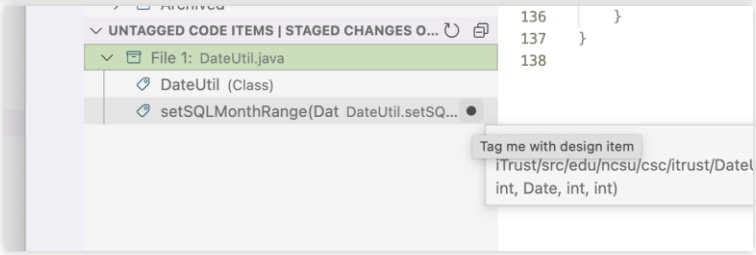
Given that the tagging process takes care of and automates the committing process, it is hoped that developers would gradually find that launching the tagging process directly instead of performing the commit action manually, will simplify their work and save them some time. It is worth to mention that presently AgileInsight will stage and commit all the changes in the set of files containing the code items of interest. Other changed files are not affected by the process.

Another improvement that should greatly simplify the process, and is set to alleviate the developer from almost any intervention, has been recognised towards the end of the last development cycle of this work. This improvement is presented briefly in the last section of this chapter.

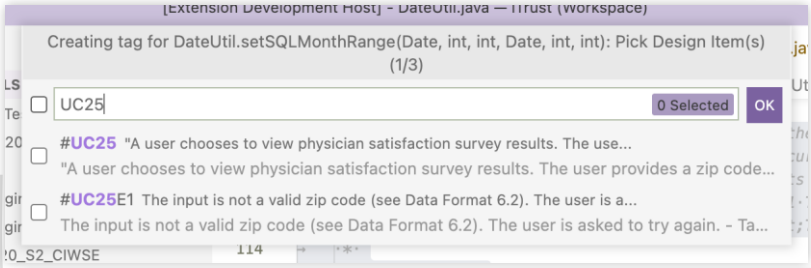
[92] The user can also use the third method that will be described next. A slight improvement to select all code items directly from the list (which is inspired from the third evaluation stage feedback) is planned for future development of this work.

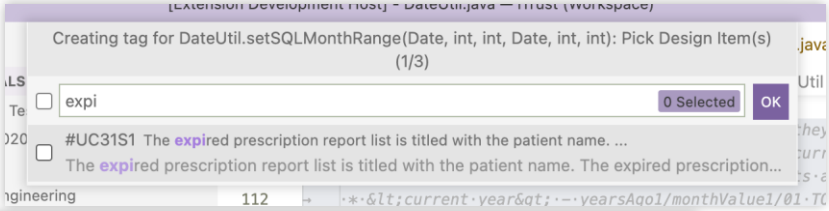


2: Once user accepts the prompt, AgileInsight identifies the code items in the staged changes that need to be tagged, and presents them to the user, who can then tag those desired. Checkboxes will be introduced in future once Vscode APIs permits it.

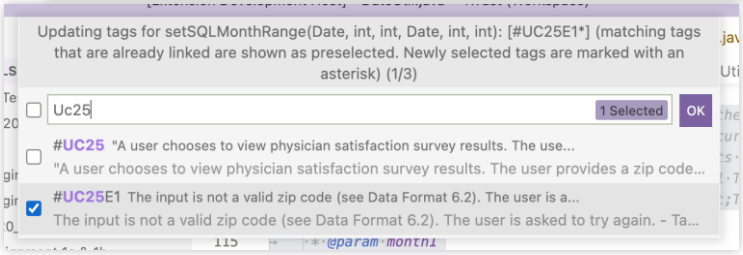


3: User can then look up the design item (or items) to link to the staged code item being tagged. They can search using design item identifier or any text that appears in its title or description





4: User selects the design item in question



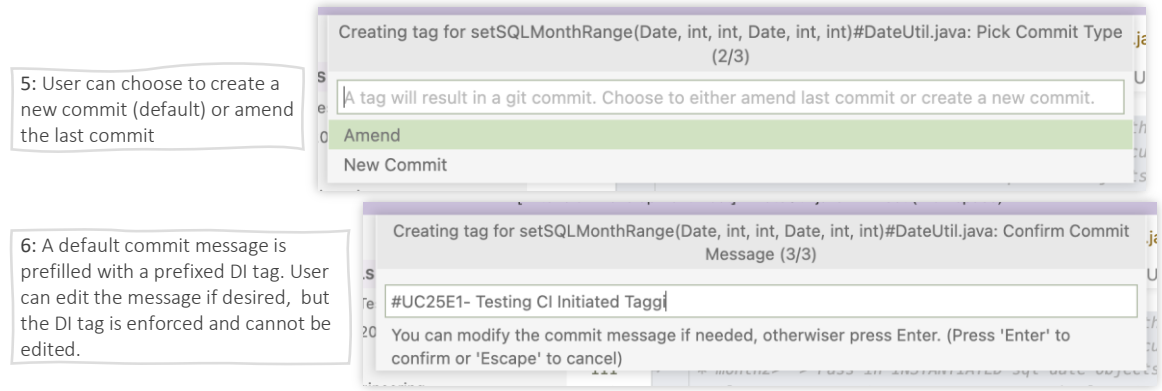


Figure 6.28: A demonstration of the on-commit initiated tagging process. Once user confirms the last action, the three-legged tracelink operation commences automatically as illustrated in the previous method.

6.7.3 CI Initiated—Tagging a Code Item with its Related Design Items

While the previous two methods are expected to be those most naturally fitting into a typical workflow of developers, a third method was found to be imperative—if only to provide the necessary flexibility to account for those less typical situations. This method enables developers to initiate the tagging process from the code item side. It allows them to accomplish the counter process of the first method—that is, to start at a particular code item of interest and select one or more design items to bind them to it. This could be particularly suitable for those situations where an enhancement has been introduced to the codebase that affects a number of existing design items. Other relevant scenarios are also examined in Chapter 8.

This method can be initiated in a number of ways. The most straightforward method is to point to the name of a code item in the source code. This displays an actionable hover tooltip on which the developer can simply click to launch the process. It can be similarly launched by right-clicking on the code item in the source code or via a keyboard shortcut. Once it is launched, the developer is presented with exactly the same mechanism described in the first method, except that this time the user can lookup and select the related design items instead of the code items. Figure 6.29 illustrates this workflow with pictures.

Note that, regardless of the various approaches implemented to initiate the tagging, the underlying processing will always run the same core tagging process for all three methods. Needless to say, whichever method the tagging is initiated by, the resulting outcome will always be the same—a design item is strongly bound to one or more code items on the user's dashboard, in their workspace source code, and on their source code repository.

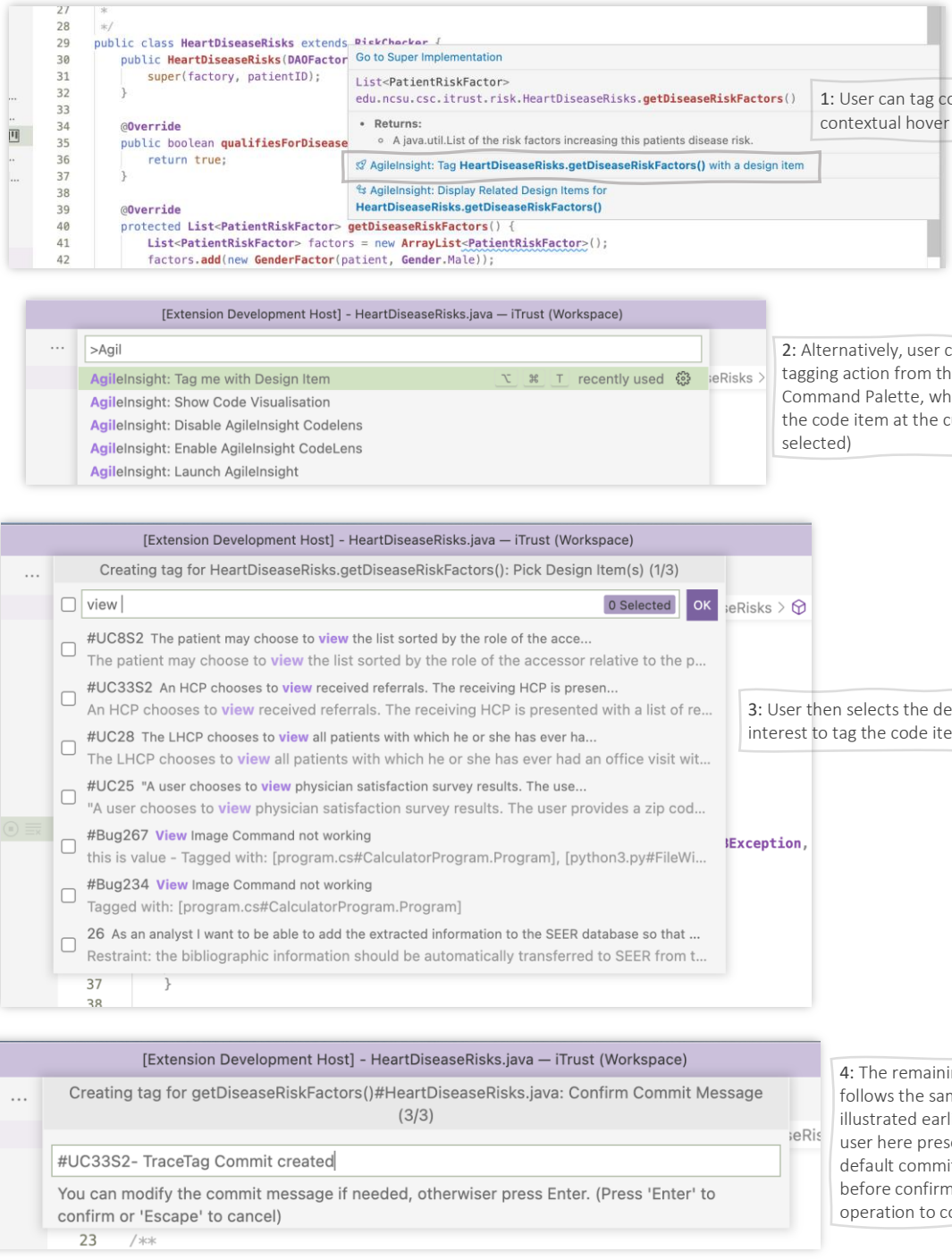


Figure 6.29: A demonstration of the CI initiated tagging process, showing here the variant method of initiating the three-legged tracelink operation (first three pictures).

6.7.4 Other Secondary Methods to Initiate the Tagging

In addition to the three primary methods described above, a number of secondary methods were also implemented to provide more flexibility to the user and to evaluate their usefulness. Rather than fitting into a developer's daily workflow routine, these methods are instead developed as accessibility mechanisms intended to support activities that address the quality of a codebase by introducing (or increasing) traceability coverage across the source code. They include, 'CodeLens' quick actions displayed within source code, and three viewlets that list recommended code items for tagging based on their change status. These secondary methods are described in Appendix II, along with an illustration of each.

Closing Remarks. In concluding this section, it is worth drawing links to the arguments presented at its beginning. It was noted then that the vast majority of techniques appearing in recent software traceability research are based on analysis and rediscovery methods—many of which utilise advanced machine learning algorithms—in an attempt to reconstruct the lost tracelinks, which then require human intervention to confirm those links. This factor alone—that no matter how advanced and accurate the developed technique is, there will always be a margin of inaccuracy requiring a confirmation from a human agent to establish trustworthiness—is reported to drive many organisations from adopting such solutions (Wang et al. 2018; Cleland-Huang, Rahimi, and Mäder 2014; Bella et al. 2018; Lin et al. 2018). The technique developed in this work and presented in this section is motivated by taking another look at the problem, and in doing so introduces a prevention mechanism so the tracelink knowledge is not lost in the first place.

The tagging techniques presented are thus based on a philosophy to promote a way of working—particularly for agile developers—where tagging while coding becomes a good habit. If developers start to see the benefits of tagging their code, and if they are presented with tools that enable them to accomplish it with the least disruption to their normal way of working, then it is possible that such a habit can start to be gradually picked up by the practicing community. While this work represents an initial attempt in this direction, future researchers are likely to introduce improved and advanced techniques. The ultimate goal is that development teams will come to consider it as a standard practice to declare the code changes that were introduced in fulfilment of a requirement, and that implementation of a user story or an issue cannot be considered done until the developer has acknowledged where in the source code the implementation has occurred. As stated above, early glimpses and seeds of this can be found in some development communities on the GitHub platform (as

well as Jira). If such a way of working comes to be gradually adopted by the development community, then it could practically solve a large aspect of the outstanding traceability problem.

The above can only happen when a high degree of modularity is achieved at the requirement side and at the source code side (the implementation). Each needs to have a framework that allows them to be structured based on smaller and simpler constituent units, based on which complex concepts are then built. This way, boundaries for the requirement and boundaries for its effect can be feasibly set. The abstract concepts of Design Item and Code Item are exactly meant to achieve that. The Design Item is enabled by modern agile practices, while the Code Item is made possible by abstracting structural elements of programming languages. In Chapter 7 and Chapter 8, some early benefits of synchronising the design with its implementation are presented, with some initial attempts for enabling the user to instantly switch between the two views.

6.8 Technical Aspects of the Tagging Module—Leveraging Vscode Git Extension

In the previous sections, a number of discussions referred to AgileInsight capitalising on the features of Vscode's native Git extension to achieve certain capabilities. This section seeks to present key technical aspects of where and how this utilisation is taking place.

6.8.1 Git Events and Git Commands

The tagging mechanism presented earlier is heavily reliant on utilisation of git internal features—some exposed directly through their official API and others through tapping into their internal commands. Thus, as indicated earlier, a local git repository is a prerequisite for the tagging operation to work successfully. The primary utilisation of the git system happens first in facilitating the automation of the commit workflow, and then in detecting relevant code items in source code changes. This subsection discusses aspects related to the automation while the next treats the code item detection.

Commit Workflow Automation. The automation of the commit workflow clearly requires the ability to invoke essential commands such as a commit or a push operation. Many such actions are easily achievable via the directly exposed git API interfaces. However, until the time of writing, it appears that some actions were intentionally kept private. The staging cannot, for example, be invoked through this API—but AgileInsight requires it. Thanks to Vscode architecture, it is possible to tap into other extensions' internal commands and request their execution, even when they are not explicitly exposed. Code Block 6.12 demonstrates an example of both of these utilisation approaches, and how to first acquire the git API.

```
const gitExtension = vscode.extensions.getExtension<GitExtension>('vscode.git').exports;
const git = gitExtension.getAPI(1);

//workspaceUri: the URI of current active workspace in vscode environment
const branch = await git.getRepository(workspaceUri).getBranch("HEAD");
git.getRepository(workspaceUri).commit(msg);
```

```
await vscode.commands.executeCommand('git.stage');
```

Code Block 6.12: Example code fragments showing how the API of Vscode's git extension can be acquired and reused (top). The bottom row illustrates how to perform direct invocation of git commands through the execute command method. Use of the later method is more sophisticated and requires knowledge of the exact return types and any parameters required, as well as their type. AgileInsight utilises both methods to achieve its functionality—as not all features are exposed via git's direct API interface.

A second essential need to automate the commit workflow is the ability to listen to and act upon events occurring in the git system, and specifically to *state* changes of the local repository. For example, after

attaching a DI tag to a code item block, it takes few milliseconds for the git system to detect the source code change, and invoking a stage command earlier will cause a failure. AgileInsight must thus wait and listen for when the change has been picked up by the git repository before requesting to stage the changes. A typical commit operation will normally involve a stage, a commit, and then a push operation. However, to prevent possible invalid states between the local and remote repositories, AgileInsight precedes those with a pull operation to ensure a clean state. The timing of each of those operations is crucial to ensure a smooth automation of the workflow, and this requires a combination of event listeners and careful wait-and-check approaches. The operation as a whole had to be calibrated over extensive trial and error and experimentation efforts under various circumstances to ensure a smooth execution. However, covering the details of this automation is beyond the scope of this thesis.

An important aspect to report here, however, is that listening to git events can be achieved in two ways. The first is to use the listener subscription interface made available directly through the git API, which will detect and fire events internally through the git system. Code Block 6.13 presents a code fragment showing how these can be utilised. The second is to create an external file system watcher on the local repository folder (similar to the method illustrated in Code Block 6.10). Unfortunately, in both cases, the event itself is ambiguous, only indicating that a change has occurred, but it is not possible to directly identify the nature of the event or the file concerned. For that, the developer will need to retrieve and manually examine the changes by accessing the properties `workingTreeChanges` or `indexChanges` of the `RepositoryState` interface after receiving a change event. The behavior of the external watcher was also found to differ from the internal git listener in that events are only fired when the user actively performs a git operation, such as a stage action—which turns out to be more suitable in some situations. Its behaviour, however, is tightly dependent on the underlying operating system, which can cause irregular results. On the other hand, the internal git listener will fire every time the git system detects a change such as user modifying a file by typing—which might be too excessive for most purposes, unless regulated. Ultimately, AgileInsight had to make use of both methods to achieve the desired behaviour under different functionalities.

```
const gitExtension = vscode.extensions.getExtension<GitExtension>('vscode.git').exports;
git = gitExtension.getAPI(1);

git.getRepository(workspaceUri).state.onDidChange(() => {
    /***
     * perform some action
     */
}
```

Code Block 6.13: Code fragment illustrating how to listen to Git state events on a local repository through the `RepositoryState` interface (the `state` object in the code) of the Git Extension API.

6.8.2 Git File Changes Blobs

The Git system is fundamentally built on binary objects called blobs. However, not to steer away from the focus of this work, blobs in this discussion can be considered as string objects containing parts of a source file that are organised in special arrangements. When the git system detects changes in a file, such as parts being modified, deleted, or even changes that have been staged ready to be committed, they are stored in blobs, and Git API makes it possible to retrieve these blobs in string format.

In earlier discussion it was indicated that AgileInsight employs a special text processing technique to examine such blobs in order to identify particular code items that are of interest. For example, the untagged code items viewlets presented in section 6.7.4 (also see Appendix II) rely on this process. This process is also important for the tagging operation for detecting when a code item has been successfully tagged, and that the change is ready to be staged. It is also the mechanism that enabled AgileInsight to implement a Tagging Reminder System (discussed in next section).

Presenting the technical details of the implementation is beyond the scope of this writing, however, the mechanism works by scanning the textual content of the blob to recognise the name of any code item that is registered in AgileInsight's Global Source Code Model. If one is identified, it is then possible to examine further if the code item is tagged or not, and by which DI/s it has been tagged. The difficulty lies in understanding the special line arrangements, their meaning, and interpreting the symbols used, in order to decide whether an encountered code item is of interest or not. Code Block 6.14 presents code fragments illustrating how such blobs can be obtained from git, along with an example of a returned result. In brief, the methods `diffIndexWithHEAD()` and `diffWithHEAD()` of the `Repository` interface can be used to retrieve the staged changes and recent changes[93], respectively.

```
//workspaceUri: URI of user's currently open workspace
//git: the Git Extension's API object

let recentChangesBlob = await git.getRepository(workspaceUri).diffWithHEAD(filePath);
let stagedChangesBlob = await git.getRepository(workspaceUri).diffIndexWithHEAD(filePath);
```

[93] In Git terminologies, these are called working tree changes and refer to changes that have not been staged.

```
Changes are diff --git a/src/python3.py b/src/python3.py
index 0c3a79e..0ea67ab 100644
--- a/src/python3.py
+++ b/src/python3.py
@@ -50,7 +50,7 @@ class LoginWindow:
     # width and places its contents into the reference $ref.  If $ispwd,
     # treat it as a password entry box.  Returns the text variable which
     # gives access to the entry.
-    #@DesignTags[#Issue67]
+    #@DesignTags[#Issue67, #Bug235]
     def genpair(self, row, text, width, ispwd=False):
         "Generate a label and entry box pair."
         tbut = Label(self.main, text=text)
@@ -85,15 +85,16 @@ class LoginWindow:

         # Make sure we're all filled in.
         if not host or not acct or not password:
-            showeror(self.main, "You must provide a user name and password.")
-             return
+            showero(self.main, "You must provide a user name and password.")
+              return

         # Attempt to connect to the remote host and log in
         try:
             self.conn = FTP(host, acct, password)
             self.conn.set_pasv(True)
+            #some comment
         except:
-            descr = sys.exc_info()[1]
+            descr = sys.exc_info()[0]
             showerror(self, descr)
             return
```

Code Block 6.14: Code fragment showing how to retrieve git blobs of recent and staged source code changes (top). The bottom image illustrates a printed example output of a returned changes blob.

6.8.3 AgileInsight Tagging Reminder

As indicated earlier, expert developers expressed during the collaboration sessions that the best way they would like such a tagging mechanism to be integrated into their IDEs is through the commit workflow. The feedback seemed to point that the commit event presented the best candidate stage where traceability knowledge should be declared by the developer and captured by the system. This feedback has guided the design and influenced the final shape of the tagging operation in this work, leading to the incorporation of the third leg—the integration of the version control system—into the process. In principle, the implementation should be straightforward, to have AgileInsight intercept a normal commit operation and prompt the developer to declare the knowledge that needs to be captured.

Unfortunately, Vscode's Git Extension API does not provide any mechanism to listen to when a user is about to commence a commit operation. Vscode's general extension development API also does not offer such a mechanism[94]. Thus, after some design deliberations, the best workarounds selected at the time came in two forms; an on-commit prompt (that was presented earlier), and a timed reminder prompt. Relevant details are presented next for both. Needless to say though, the earlier discussed DI-

[94] In fact, Vscode did at some point offer a listener interface called `onDidExecuteCommand` that could have potentially allowed such interception. However, the interface was soon voted by the community as undesirable for performance and security reasons, and was hence removed.

Initiated and the CI-Initiated methods have already been designed to integrate an automated commit action, so both of these work to intercept a commit action as well, albeit implicitly before it materialises.

1. Implementing the On-Commit Prompt

Despite the impression that the naming might suggest, and as was described earlier, this method fires a prompt when the user begins to interact with the input box of Vscode's git panel, and before they launch the actual commit operation. To implement this, AgileInsight had to implement its own listener process. It uses a timer that gets triggered whenever the user stages a file change—which typically indicates that the user is likely to commence a commit action soon. The timer launches a background listener that starts watching git's input box for the commit message, for a period of time (currently set at five minutes). If within that period, the process detects that the user has begun to type into the input box (by regularly reading its value), then it will display the tagging operation prompt as was illustrated earlier. If no interaction was detected within that time period, then the process times out and is killed. This implementation was only possible thanks to git exposing its commit input box value to be read through the interface property `Repository.inputBox`. As stated earlier, this prompt process is only enabled when the staged file changes are found to contain code items that are recommended for tagging—a process that is made possible thanks to the Git File Changes Blob discussed above.

2. Implementing a Timed Reminder Prompt

An extension to the above process was considered to be potentially useful to help encourage developers to embrace the tagging practice. Rather than waiting for the user to stage their changes, if AgileInsight detects that some ***recent*** file changes contain untagged code items that are eligible for tagging, then a similar process will launch a prompt to advise the developer that it is probably time to tag those newly-contributed code items. If the developer accepts the prompt, then they are taken to the list of the untagged code items in recent changes, (see Appendix II), where they can initiate the tagging operation.

The reminder employs a user-configured time value that determines the interval of when this reminder is displayed—or it can be completely disabled if desired. The argument behind this reminder feature is that if such a newly introduced practice is to find a receptive audience and enjoy adoption among developers, then a way is needed to present it to them in a transparent and non-intrusive manner—and at the times when its value becomes relevant to them. It could also help teams who decide to introduce the practice—or try it out—in their work environment. Results of the evaluation reveal more on this in the coming chapter. Figure 6.30 illustrates some aspects of this reminder mechanism.

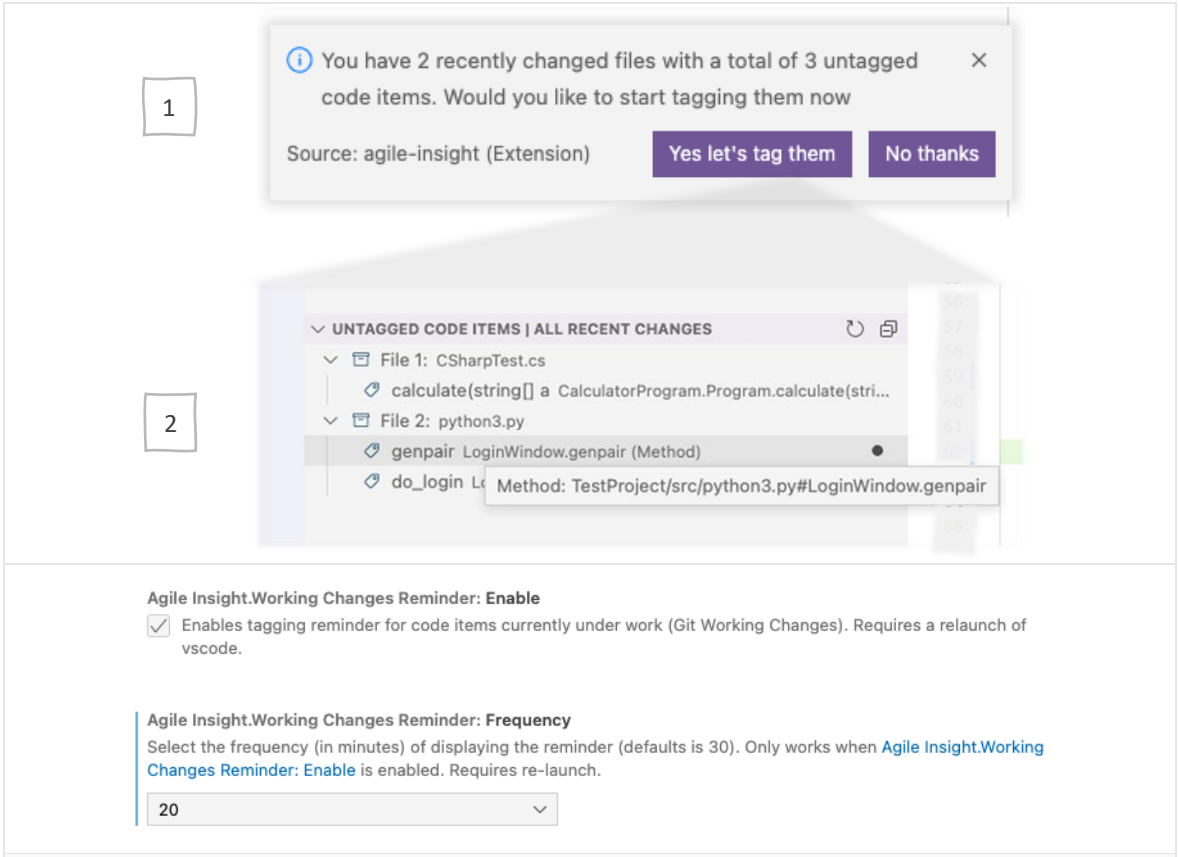


Figure 6.30: An illustration of the timed tagging reminder implemented in AgileInsight. A prompt is triggered according to a user-configured timed period (bottom row), alerting the developer to the existence of newly added code items that are eligible for tagging. Accepting the prompt takes the developer to a list view (step 2) where they can take action (also see Appendix II).

6.9 Scenography–A Language-Agnostic Source Code Visualiser

In earlier sections the underlying technicalities and technology were discussed to explain how the integration of—and user interaction with—a 3D scene is achieved within the Vscode environment. Earlier chapters have also presented the arguments and theoretical motivations behind this direction. This section now reports some technical details with regard to the process of creating the visual representation of a codebase, that are considered to be of particular interest.

6.9.1 A Generic Source Code 3D Modeller and Layouter

The city metaphor and software maps techniques were presented earlier (see section 2.1) as popular and highly versatile approaches for visualising the structure of software systems. Arguments for their cognitive advantage and application versatility were also laid out as justifications behind their adoption in this work. Two facts are of particular interest to the discussion here. The original city metaphor as introduced by (Wettel and Lanza 2007a, 2007b) was designed to visualise primarily object-orient source code. The majority of publications employing it appear also to target Java systems in particular (Jeffery 2019). These two facts appear to also apply to the software maps metaphor, which takes a slightly variant layouting technique presented by that of the city metaphor (and some researchers might also argue that it is an independent, but rather parallel, development to the tree-map visualisation metaphor, which is the forefather of all containment-based visualisation approaches (Johnson and Shneiderman 1991; Shneiderman 1992; Johnson 1993)). In either case, both visualisation approaches appear to have been predominantly confined to object-oriented source code, and demonstrated in publications with primarily Java systems only[95]. As far as this research has revealed, no work has presented or claimed a generic application of either approach (or some other similar approach) that is able to handle and visualise source code in an agnostic manner, and irrespective of the underlying language or paradigm.

Thanks to the previously introduced language agnostic source code parser—and by some modification applied to the original city metaphor layouting algorithm—AgileInsight is able to present a completely generic and language-agnostic source code visualiser. Two key steps are of interest to cover here. The generic 3D source code modeller, and a corresponding generic 3D node Layouter.

[95] After deeper examination of the literature, 3 exceptions were found. These are (Viana, Hora, and Valente 2017) who applied the metaphor to JavaScript, (Nunes et al. 2017) who applied it to Swift, and (Brito et al. 2019) who applied it to the Go language.

1- A Generic Source Code Containment Model

Section 5 of this chapter introduced how an abstract source code model is built by AgileInsight regardless of the underlying language. It also highlighted how AgileInsight introduces a new DocumentSymbol object to represent a file as an enclosing entity, under which the original DocumentSymbol array object is kept. A revisit to the diagrams in Figure 2.20 & Figure 2.21 could be helpful at this point. With this abstract model being now available, the process of generating a 3D visualisation of the source code becomes possible, and without necessarily worrying about the nature of those DocumentSymbols. What is important is to be able to account for each valid document symbol (i.e., code items) of the source code in a generic manner that intrinsically reflects the original source code structure. While AgileInsight can expose and visualise all categories of document symbols, elements such as variables or parameters have little value for the purpose of this work and visualising them would only defeat the purpose of the visualisation as it would lead to aggravated visual clutter. Thus, AgileInsight only visualises the valid document symbols as presented in Table 6.2.

Constructing The Folder Structure. As explained earlier, inner constituent entities of files are readily provisioned by Vscode's language services API, and AgileInsight then adds the enclosing file entity. In order to account for a complete codebase structure, one additional part is remaining—that of the folder (or package) entities. These elements are not generally accounted for by the language services API[96], and variations in implementation were found across different language servers. Therefore, AgileInsight implemented a common process to create those elements itself. It achieves that by parsing the folder/package of any codebase up to its root folder, and then recursively create a corresponding DocumentSymbol for each encountered folder/package.

Constructing A Full Containment Hierarchical Model. However, to transform that source code model and its constituent DocumentSymbol entities into graphical objects that are render-able onto a 3D scene, one last step is needed. Each DocumentSymbol needs a few extra properties to tell the renderer how it should be visualised—e.g., what dimensions it should have, and where it should be positioned across the scene (i.e., its XYZ values in the 3-axis world coordinate system of the 3D scene). This is where the layout algorithm comes into the picture, and which will be discussed shortly. Moreover, for the layout algorithm to work, the complete source code model needs to be presented in

[96] No standard or common practice was found to be adopted by the various language servers here. Java was found to return the packages (as packages play a fundamental role in the language), however it returned them as flat entities completely separate from the file structure. Other languages completely ignored folder entities.

a tree-structured hierarchical containment model, with a single parent node. Each parent needs to physically contain its children inside of it. To achieve both goals above, the original source code model is traversed—alongside the created folder entities—to create an abstract container object that encapsulates the DocumentSymbol object for each element and adds its children under it. The process runs recursively, reconstructing the exact hierarchical source code structure from its root folder (or root folders) down to the contained files and their inner elements such as functions and classes. This container object holds those extra properties mentioned earlier, and it is where the layout algorithm is applied to calculate and assign those properties. The abstract containers are eventually transformed into the visual nodes that are rendered in the visualisation scene—i.e., the Three.js Mesh objects (see Figure 5.9).

In summary, to create the visualisation, AgileInsight generates a containment model of the codebase irrespective of its underlying language or programming paradigm. As long as Vscode can serve the `DocumentSymbol` objects of the source code files, AgileInsight is able to produce an abstract hierarchical containment model that encapsulates those `DocumentSymbol` objects, which can then be visualised.

A particular distinction to be highlighted here is that in traditional containment models that particularly target Java systems, classes are seen as synonymous to files. File entities are thus typically seen as redundant and dropped from the containment model. However, to enable a truly generic and paradigm-free visualisation, AgileInsight accounts for files as concrete enclosing entities. This was one reason why `DocumentSymbol` objects had to be introduced to account for files. The other reason was to enable file-level tracelinks to be created, as was detailed earlier.

Figure 6.31 presents a diagram to help illustrate this language-agnostic containment model.

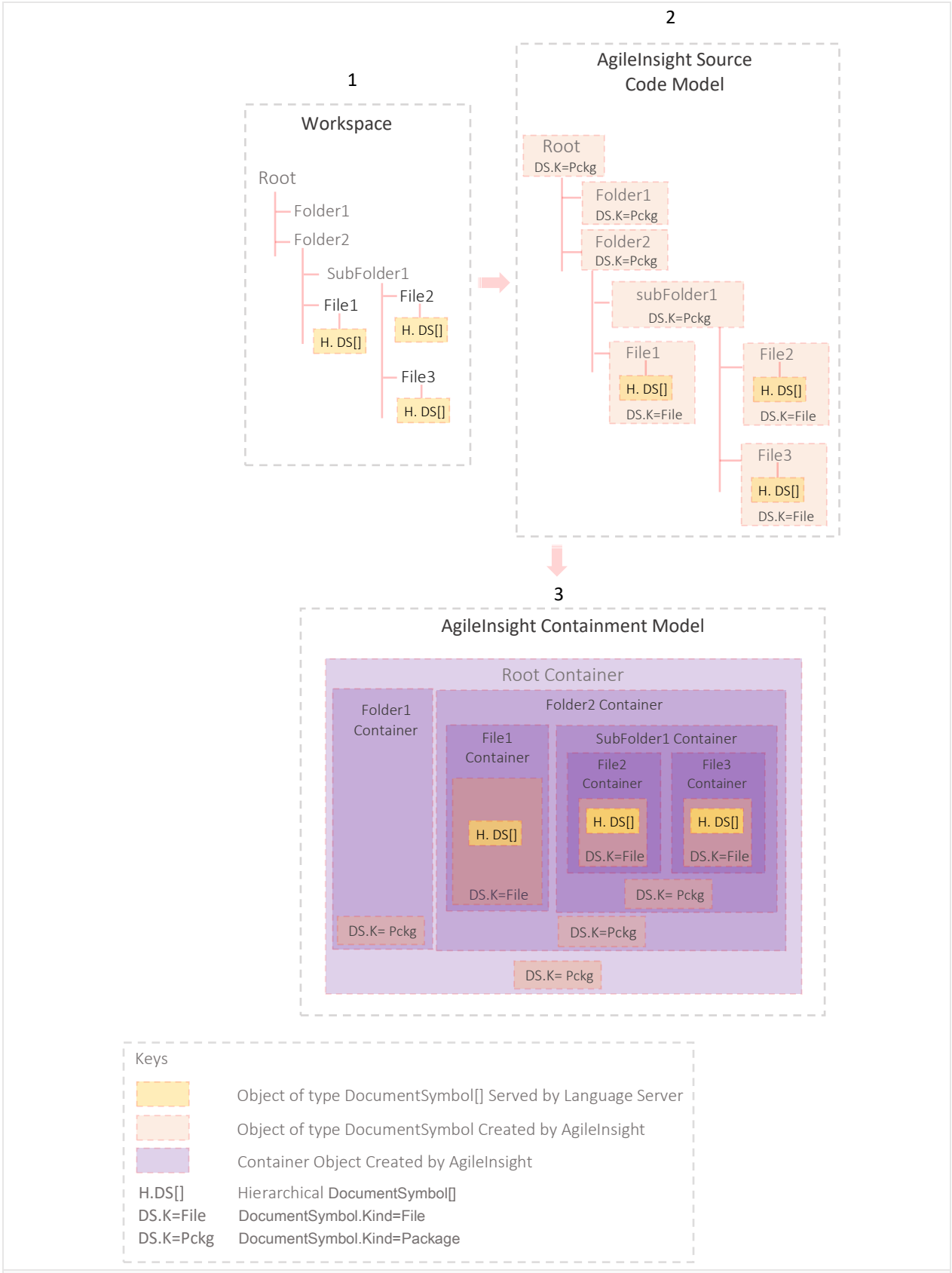


Figure 6.31: This diagram illustrates the modelling steps taken by AgileInsight to create the source code containment model that is required to produce the visualisation. Note that step 2 is created at the parsing stage (see Figure 6.21) and only the folder objects are added here. The yellow DocumentSymbol[]s are the only objects that are readily available from the language servers. The purple container objects are eventually transformed into the Mesh objects of Three.js for rendering (see Figure 5.9).

2- A Generic 3D Node Layouter

To enable AgileInsight to visualise any source code irrespective of its language, a simple modification was introduced to the original layouting algorithm. However, before explaining this modification, a quick introduction of the original layout algorithm is necessary.

Layouting Algorithm. Without diving into unnecessary details, the city metaphor (or software maps) visualisation technique is based on looking at source code as a hierarchical containment model, where elements are structured in a strict parent-child relationship. For example, and in the traditional software visualisations where object-oriented code is predominant, packages are presented as containers that hold classes, which in turn hold methods or inner classes. A binary tree algorithm is then typically employed to traverse through the entire containment model to calculate the dimensions and positions for each node, which eventually results in the visible city (or map) layout seen in the final visualisation. The algorithm works inside out, recursively expanding the area of a parent container to accommodate the total size of its child containers. This research builds on the same layouting algorithm employed in our earlier work (Alshakhouri 2013), which itself was based on the original algorithm introduced by Richard Wettel (Wettel 2010). The reader is referred to Wettel's original work for detailed discussion of the underlying algorithm.

One distinction in the original algorithm is that it is normally implemented to only permit certain categories of objects to work as parent containers. This is typically limited to packages and classes. Moreover, the category of permissible parent containers is predefined in the original model. No new object category can be accounted for in the visualisation, let alone to work as a parent container. Parent containers are crucial because it is where the layout algorithm works to compute the dimensions and positions, and thus where the visual structure emerges.

AgileInsight introduced a simple change to free the visualisation and its underlying layouting algorithm from any predefined model restricting its functioning. In AgileInsight, any element in source code can be visualised as long as its language server serves a corresponding DocumentSymbol for the element. Any element can further be a parent container. The layouting algorithm does not restrict its layouting process to a certain category of containers. Instead, it runs the layout process against any container that happens to have children. Running this process recursively enables AgileInsight to layout any hierarchical parent-child containment model. It does not matter if a method is inside a class or a class is inside a function, this layout approach can expose the structure of the source code irrespective of its underlying paradigm—whether it is object-oriented, functional, mixed, and so on.

Since the language-agnostic parser takes care of providing an abstract hierarchical model of any source code, AgileInsight is able to produce a corresponding containment model in its image. In this manner, AgileInsight is able to model the structure of any source code into an abstract containment hierarchy, which can then be visualised. However, to ensure the final visualisation is meaningful and relevant, and to prevent unnecessary visual clutter, AgileInsight limits its visualised source code entities to only those defined as valid elements (as discussed earlier). This prevents elements such as variables, properties, and parameters from cluttering the visualisation— neither is considered to add any cognitive value for understanding the structure of a codebase, which is the original drive of software structure visualisation.

Figure 6.32 and Figure 6.33 provide a graphical illustration of the working of the traditional layouting approach versus the generic layouting approach introduced in this work. Figure 6.34 shows an actual source code visualisation that demonstrates the flexibility of the generic approach.

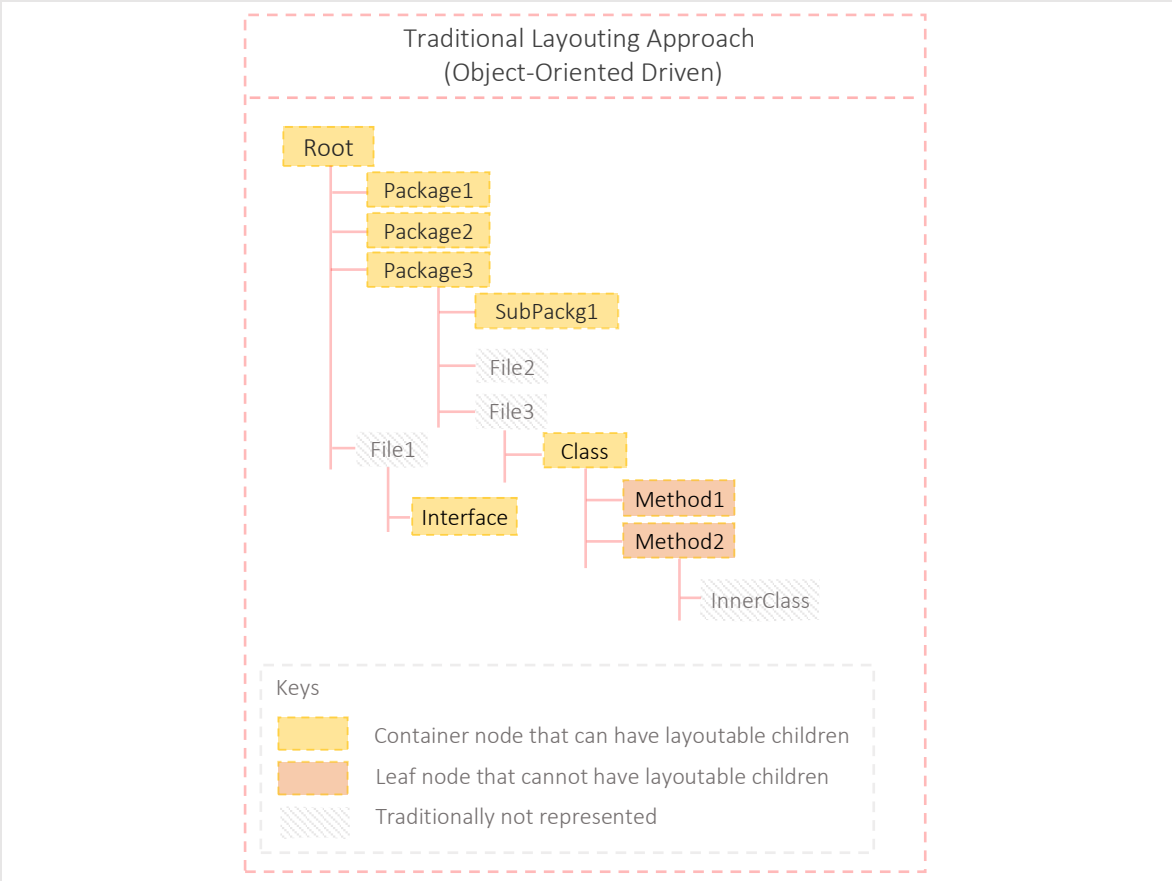


Figure 6.32: An illustration of the traditional layouting approach, which dominates software visualisation research. The approach is strongly influenced by an object-oriented perspective to the code structure—making it unsuitable for functional or other nonconventional programming paradigms.

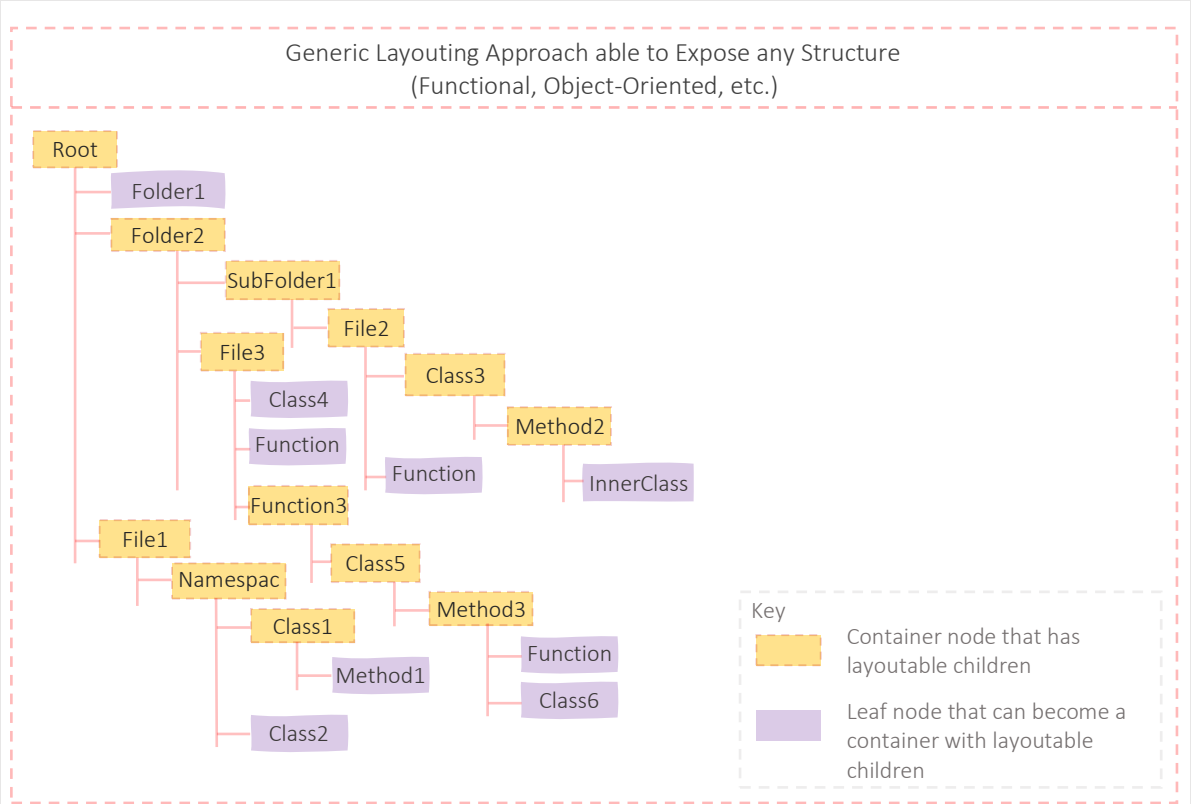


Figure 6.33: An illustration of the generic layouting approach adopted by AgileInsight, which is able to expose any code structure irrespective of the language paradigm. Any node is potentially eligible to become a container with layoutable children—as long as it has ones. Note that a tree structure (rather than an actual containment) is used in the demonstration for more clarity. While, some depicted code structure might not align with recommended practices—it is still allowed in languages that permit a mixed style of functional and object-oriented designs. The goal of the illustration is to demonstrate the flexibility of the model and it's ability to expose any structure.

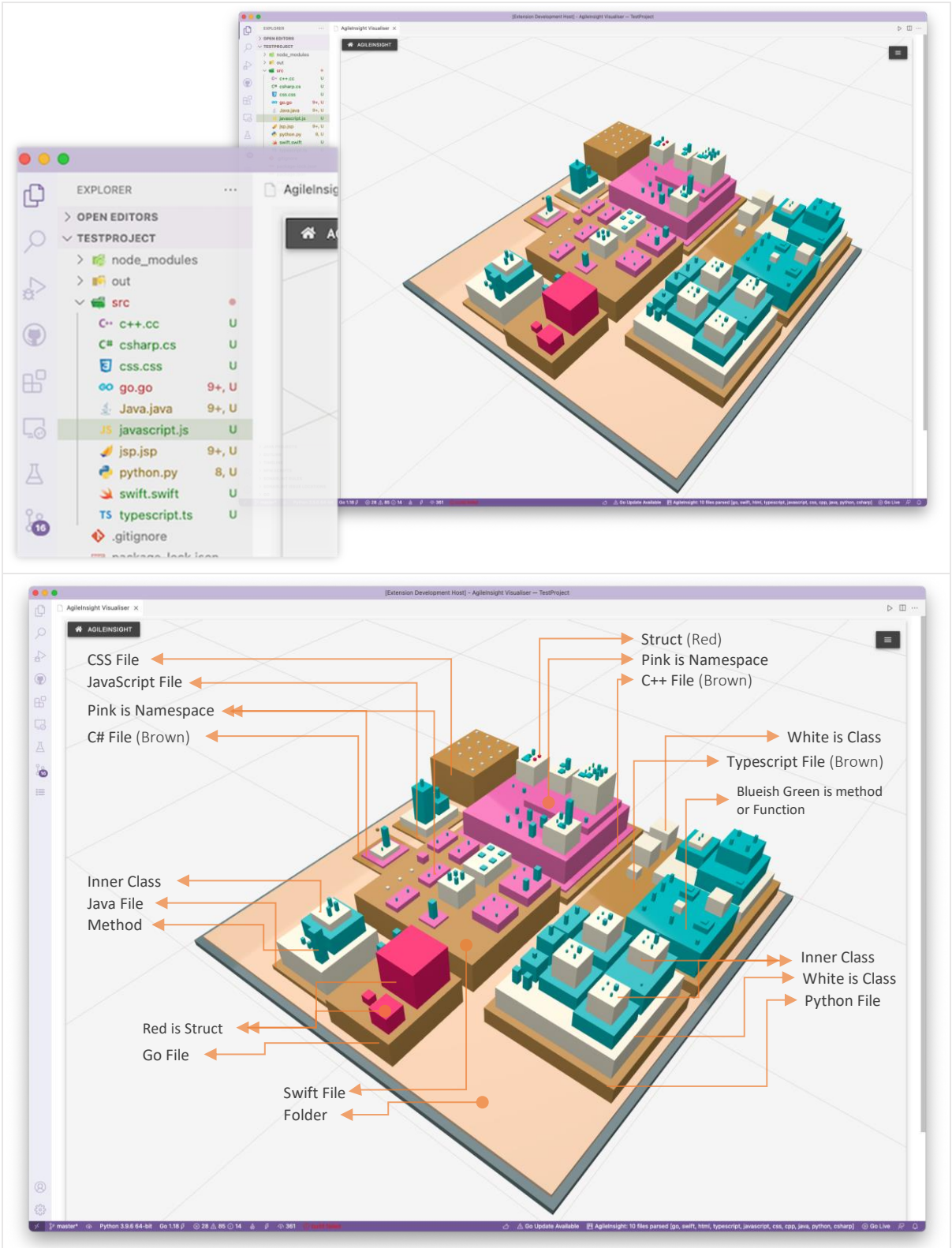


Figure 6.34: **Top**: a demonstration of AgileInsight's language-agnostic visualisation capability, illustrating the flexibility of the introduced generic node layouting technique. The visualisation is generated using 10 files of different languages, obtained from real-word open-source systems and placed in a single project folder for demonstration purposes (see section 7.4 for more details). **Bottom**: explanation of the visualised code items detailing the color mappings used; White for classes, brown for files, blueish-green for methods & functions, red for structs, pink for namespaces, and beige for folders. The grey platform is the workspace folder.

6.9.2 Dynamic Mapping Functions

A key characteristic of software visualisation techniques is how the source code elements are mapped to their corresponding visual objects that are rendered in the scene—i.e., the meshes. Earlier software visualisation research was generally focused on representing software metrics starting with simple indicators such as lines of code (LOC), to more advanced quality measures such as cohesion and cyclomatic complexity. These metrics are normally mapped in a linear fashion to shaping characteristics of the visualised meshes, such as their height and width—and sometimes to other attributes such as their color or texture. Later research became more focused on communicating the structure of the software and how components are distributed or situated across the corpus of the source code, in order to help the user better comprehend their code. Metrics are then looked at as secondary features that can be projected on demand at a later stage.

The interesting point is that in order to visualise the structure of the source code—at least when using the metaphors concerned in this research—some metrics must still be used as a basis for computing the dimensions of individual objects, and for the final visual appearance to eventually emerge. The LOC, number of attributes (NOA), and number of methods (NOM) properties are the most popular candidates for this purpose in the majority of research works in the field. Regardless of the chosen metrics though, they are almost always mapped linearly to an object's dimensions. Our earlier work followed the same approach as well.

In this work, an improvement is implemented by introducing dynamic mapping functions to map the selected metrics to an object's dimensions. The argument behind this is that linear mapping often results in an awkward-looking landscape that tends to defeat the purpose of the visualisation—many times resulting in a visually cluttered and disharmonised scene. For example, the scene can become dominated by a large number of very differently-shaped buildings, increasing the visual complexity of the scene. A visualisation should aim instead to deliver a pleasant and natural looking structure of the source code that can better make its imprint onto a user's memory. So, rather than being faithful to representing metrics accurately, this work instead sought to find a way to produce a more harmonised, meaningful, and natural look to the final visual landscape. After all, a user who is looking at the visualisation is less likely to be interested in reading accurate metric values, but rather to gain an overall sense and impression of those metrics.

Normalised & Dynamic Mapping Functions. To achieve a harmonised look, a method is needed to control how an object grows in dimension. For example, if a class or a method has an exceptionally large LOC value, then it is important to reflect this quality, but at the same time, it is even

more important to prevent that large LOC value from creating a massively tall building in the final landscape. Thus, a method is needed to reflect this variation in metrics, but to do it in a normalised mechanism so not to allow uncontrolled sharp variation across the scene. It is enough for the user to spot a noticeably large building to realise that it is more complex than its surroundings, without having to faithfully represent its KLOC into a sky-high building. To achieve this, some simple experimentations were run to test a few mathematical functions to examine their effect on the visual appearance. In particular, two functions were being sought:

1- Height Dimension: a function[97] was sought for the height dimension that had a broad rate of variation at the lower band of inputs, but that would allow the variation to smooth out as the input values start to reach the higher band. This is desirable for mapping the height of the buildings as it will allow differences in LOC to be expressed out more visibly for value ranges in the lower band—for example, from 0 to 500 LOC—which is presumed to account for a large population of the source code items. As LOC values keep increasing further, the function would smooth out the variation until differences become less visible and the height practically stops increasing—which is a desired result. At this point, it is enough to identify those items as exceptionally large, and interest in distinguishing their actual 'largeness' probably becomes less relevant.

2- Width Dimension: a similar function was sought for the width of buildings, but this time, the rate of variation needs to smooth out faster so as not to allow excessive variation in the area of buildings, as this directly impacts the layouting—and consequently the final shape of the landscape. Moreover, AgileInsight maps NOC (number of child elements) to a building's width only as an initial value. During the layout process, it allows the area of a building to expand to accommodate its children—an approach introduced and justified in our earlier work (Alshakhouri 2013). Therefore, there is less need to expose the variation and differences in the widths, but rather to cap the value from growing too large.

To illustrate the above discussion visually, Figure 6.35 presents a graphical representation of some of the functions that were tested. Figure 6.36 demonstrates then two contrasting examples of using a linear mapping versus a normalised dynamic mapping function.

After examining the effect of a number of functions, the one producing the most desirable results (in terms of visual appearance and variation) was chosen for each of the two properties above. These are

[97] Such function must be uniformly continuous.

presented in Equation 6.1. However, to offer the user more control and flexibility, a capped linear mapping function is also made available. The user can switch between linear and normalised dynamic mapping during runtime, and can choose that separately for the primary objects (i.e., files, classes, namespace, and so on) as well as for the secondary objects such as methods and functions. Other new dynamic functions can be set as configurable user settings.

It must be noted, however, that this approach to applying normalised mapping functions to produce a harmonised and more meaningful visual landscape, constitutes mere preliminary work. It is meant to demonstrate a direction for improving and naturalising the landscape of the visualisation. Future work conducting more elaborate work in this area, and applying robust experimentation, is certainly expected to lead to better results.

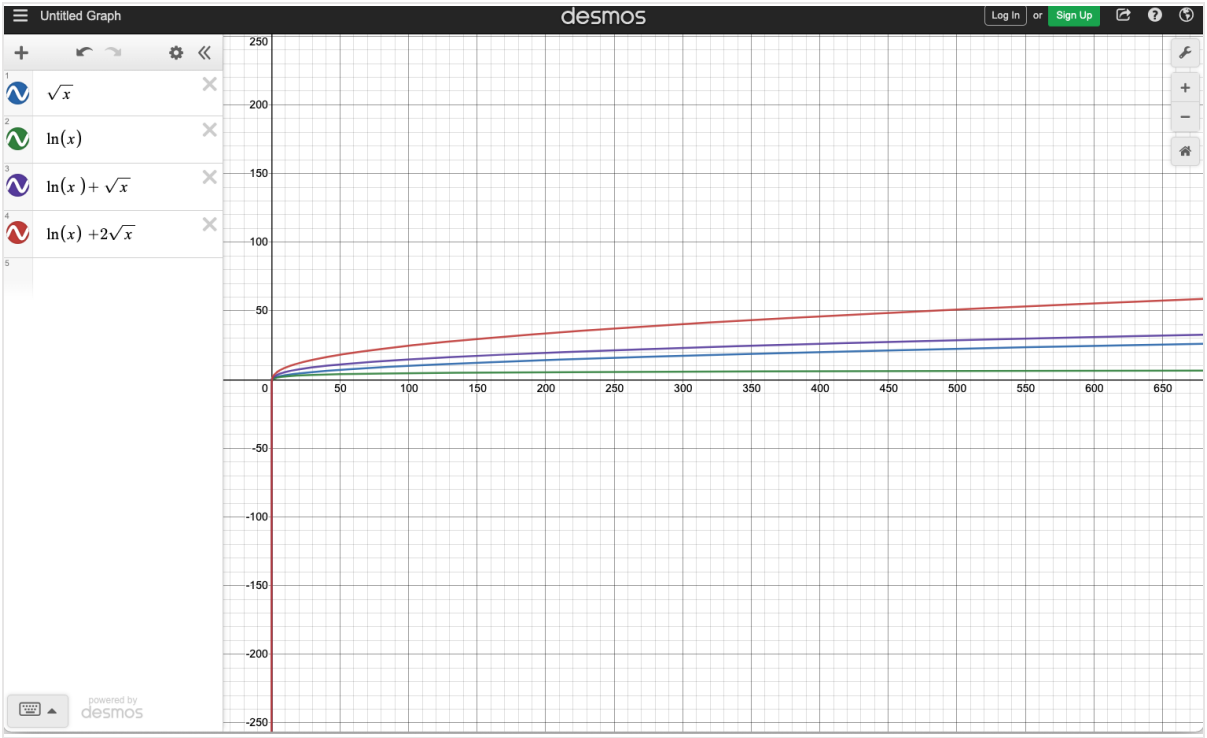


Figure 6.35: Example of the functions tested to examine their normalising effect on the visualised source code landscape. The corresponding graph for each function help illustrates the variation rate and speed of growth.

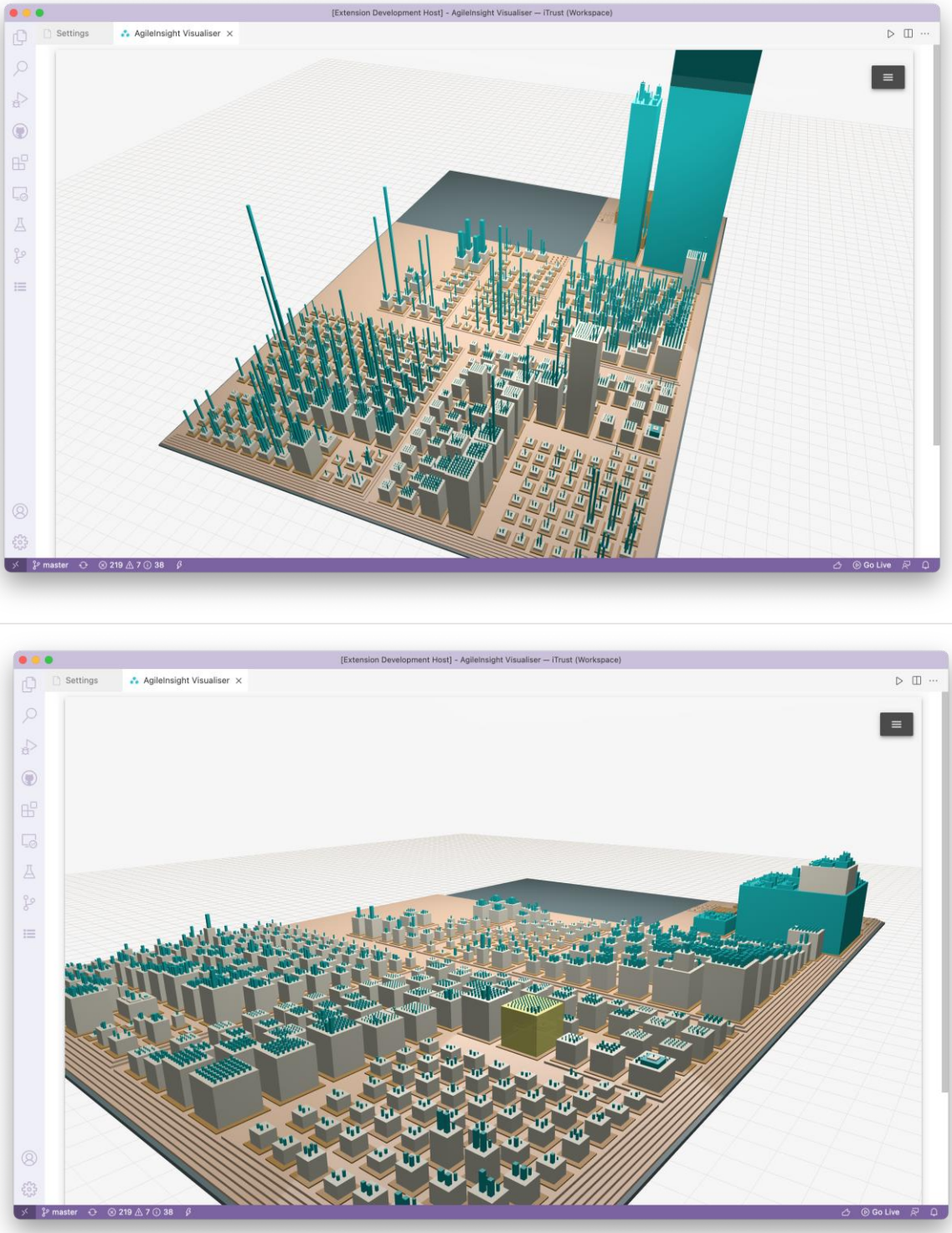


Figure 6.36: Two views of the same codebase (iTrust) as visualised by AgileInsight. The top image shows the result when using linear mapping method, while the lower image demonstrate a more harmonised and natural-looking result obtained after applying the normalised dynamic mapping functions.

$$Height = ln(NOC) + \sqrt{LOC}$$
$$Width = \sqrt{NOC}$$

Equation 6.1: Normalised dynamic mapping functions adopted by AgileInsight for the width and height dimensions of visualised buildings.

6.10 Conclusion

This chapter set out to cover the key technical aspects that characterise this work. While it has resulted in a significantly dense chapter, the work features the integration of a number of technologies—some of which are quite recent in the industry, while others are brought together for the first time. The density could thus be seen as a reflection of the complexities involved. The ultimate goal of this chapter is to enrich the field and document lessons and expertise gained from this work, and to equip researchers interested in building on it with sufficient grounds to start off where this work has stopped.

This chapter has made a number of contributions, the highlight of which is AgileInsight in itself as a tool that reduces the gap between the design and development processes. It brings the software design artefacts right into the development environment; it introduces a novel mechanism to bind the design artefacts with their implementation system artefacts; it enables the developer to instantly switch between a design view and implementation view; and it introduces a new traceability concept that holds promising potential towards solving a key outstanding aspect of the traceability problem.

On another important level, it introduces a language-agnostic and live source code parser, with a corresponding language-agnostic software visualisation solution that is able to expose the structure of any codebase with no restrictions or paradigm limitations. It introduces a software visualisation environment that is tightly integrated with a highly popular IDE—bringing the technology closer to the users who need it. It introduces an open architecture to integrate popular agile dashboards into Vscode. Lastly, it demonstrates a practical implementation to the abstract concept of Design Items and Code Items, that could allow tools to be developed without binding themselves to a specific language or development practice.

Nonetheless, the work has some shortcomings and aspects that require improvements. Some key areas include improvements of the visualisation aesthetic features (e.g., implementing a progressive rendering technique such as that introduced by (Limberger et al. 2017), introduction of cognitive scaffolding techniques (e.g., dynamic aggregation), introduction of performance improvements (e.g., finding alternatives to instanced meshes), and more application functionalities that better utilise the visualisation itself—now that it is readily available to the user.

Another shortcoming is the fact that the work proceeded with Trello integration. While integrating Jira is now a straightforward process, the fact that it is not yet complete limits the reachability and audience of the tool. For this reason, this aspect is currently prioritised for future work.

Another key improvement relates to the tagging operation. As was stated earlier, a valuable improvement was recognised that can significantly decrease the required user intervention in the tagging operation, and consequently further increase the appeal of the process and the likelihood of its adoption. The technique has also received considerable support from expert developers participating in the final evaluation (as reported in Chapter 8). It involves the developer simply checking out a design item before they start a session, which would eliminate the need to manually declare code items. Instead, AgileInsight will be able to detect related code items automatically, and the developer needs only to confirm the relation.

One last important improvement to highlight is the unique identifiers for design items. The current implementation enforces a unique identifier across a single board. There is no present mechanism to prevent the same identifier being used for a different design item in a different board. A correct implementation must ensure the uniqueness of a design item identifier across the codebase itself, so that no other design item can use that same identifier again from any other dashboard that connects to this codebase.

Chapter 7
MINI-CASE STUDIES

7.1 Prelude

Part of the evaluation activities that were carried out in this work involved some laboratory mini-case studies conducted to verify the functioning of the research tool and give initial feelings of its claimed benefits. Despite being mini case studies, they are conducted using real-world datasets as well as popular and real-world open source systems. Some of these datasets were also utilised to carry out demonstration interviews with experts from industry over three phases. This chapter presents these mini-case studies, while the actual evaluation activities with experts are covered in the next chapter.

7.2 A preliminary Mini-Case Study—Early Validation

This case study was conducted at the very early stage of this research as part of Phase One evaluation (see section 78.5 in next chapter). Its purpose was to verify the validity of the research concept at its early form. This was accomplished by utilising the prototype tool of our earlier work, as it captured a preliminary form of the concept underpinning this work.

7.2.1 Selecting a Real-World Dataset

Briefing. To test the proposed concept using the earlier-developed prototype tool, three key datapoints were required; the source code artefacts, their original design artefacts (i.e., user stories in this particular case[98]), and lastly, the tracelinks connecting each user story to its related code artefacts. Unfortunately, while the software engineering literature is abundant with publicly available source code datasets, finding a pair of source code artefacts and their original requirements is highly limited—even more so, when the requirements need to be in the format of user stories (or any of the similar agile methodologies). Even more scarce yet, are the tracelinks between those two forms of artefacts to be available. In fact, the practice of systematically keeping such tracelinks between, let us say a user story, and the exact class and method objects where that user story has been implemented, seems virtually non-extant among industry practitioners—at least at the time of the earlier research, and as far as the extent of that research has revealed[99]. (Moreover, one of the objectives of the earlier

[98] ScrumCity was an early proof of concept of synchronising the design artefacts with their code implementations, and it adopted user stories, in particular, to demonstrate this concept.

[99] Admittedly, the earlier research has not focused on safety-critical research contexts where such granularity of tracelinks might have been considered. Nonetheless, use of agile methodologies in such contexts has traditionally been avoided, and only very recently agile is slowly finding acceptance in those contexts (Cleland-Huang et al. 2021). Additionally, the search carried out in late 2017 for a dataset with such tracelinks has failed to find a match. This statement must not however be confused with the practice of developers mentioning the number of a GitHub issue in their commit messages—a practice that is relatively recent and still not yet well-disciplined.

research was to demonstrate the value of keeping record of such tracelinks—a matter that this evaluation phase set out to validate with the end users, and which was then implemented and facilitated through this particular research discourse).

The non-availability of those tracelinks should not thus be considered as a surprise, but rather as a well expected position. Additionally, agile practitioners are known to afford little attention to maintaining user stories after their successful development (Saito et al. 2018), let alone maintaining tracelinks to their code artefact implementations (Cleland-Huang and Vierhauser 2018; Cleland-Huang, Rahimi, and Mäder 2014). As far as our investigation has showed, such trace links appear to be rarely—if at all—collected in practice, despite the contested claim in this research of their potential high value for a project's development and sustainment. Similar dispositions are also found in some recent research such as Tian et al. (2021). In fact, a key objective of this work, as elaborated earlier, is to establish this value for agile practicing teams in particular, and to enable the collection of the relevant data within their work routines. Thus, expecting to find a publicly available dataset with such tracelinks seemed distantly achievable. In fact, during the course of this research, only two close matches were found—and in both cases, researchers had to ask the developers and offer them incentives in order to provide those tracelinks in a post-development activity. This phase of the evaluation uses the first of those datasets, which fortunately was made available in late 2016. Another close-match dataset was made available in a timely manner in 2021, which was utilised in phase three of the evaluation. None of those datasets were a perfect match—for example, none were the result of an agile development practice—however, both offered a good enough case to demonstrate and verify the desired objectives, as will be detailed in due course.

Cassandra Dataset. A 2017 artifact paper by Mona Rahimi and Jane Cleland-Huang had made available a dataset of Apache's open source Cassandra database that happened to present a suitable case to test ScrumCity and its underlying concept. Feature requirements were extracted from the documentation of 27 versions of Cassandra and made available along with their corresponding source code artefacts (Rahimi and Cleland-Huang 2017). More importantly to this work, a subset of 48 of those features had their tracelinks reconstructed to produce feature-to-file mappings—a key datapoint to enable the testing of the prototype and its underlying concept. While those feature requirements did not represent true user stories, they nonetheless fit the general concept of small units of actionable requirements (see Chapter 4). Moreover, given the Java nature of the dataset, the file-level mappings corresponded effectively to class level mappings, which is sufficient to demonstrate the efficacy and the practical utility of the concept, as the prototype tool supported class level and method level

tracelinks. However, this meant that the method-level traceability facilitated by the prototype could not be tested.

Technical Details. For the purpose of conducting the demonstrations and the subsequent case study on the prototype tool, only the last version of dataset source code was used, namely version 2.2.0. This version contained a total of 3214 classes organised across 68 packages. To load the feature-to-code artefact data to the prototype, additional manual work had to be first performed to convert the 48 features and their tracelinks into the XML format required by the tool. Moreover, the tracelink information were only made available as top-level class names. The prototype tool required fully qualified names (similar to the unique identifiers introduced in Chapter 6, which are used to uniquely identify code artefacts of various granularities). Those fully qualified names had thus to be manually resolved and appended to the original mappings. Furthermore, the prototype tool adopted a Scrum practice as stated earlier, and hence its XML format required the data to be structured in accordance with the Scrum practice (i.e., Releases, Sprints, and user stories). Thus, and strictly for the purpose of running the evaluation, the features were treated as user stories and loaded under a single Sprint data object—their original wording format was left unaltered. This is perceived as a sensible approximation given that those 48 features represented the actual requirements of Cassandra's first version 1.0.0-beta1. Figure 7.1 below shows the result after converting one of Cassandra's features (feature 24) data into the XML format of ScrumCity.

```
<!-- feature record 24-->
<Feature>
  <ID>F24</ID>
  <Title>TRUNCATE statement</Title>
  <Description>A TRUNCATE statement results in the immediate, irreversible removal
    of all data in the named column family</Description>
  <Owner>Ted</Owner>
  <Category>Category</Category>
  <Type>Feature</Type>
  <Priority>Normal</Priority>
  <MethodRefs>
    <MethodRef>org.apache.cassandra.db.TruncateResponse</MethodRef>
  </MethodRefs>
  <ClassRefs>
    <ClassRef>org.apache.cassandra.db.TruncateResponse</ClassRef>
    <ClassRef>org.apache.cassandra.db.TruncateVerbHandler</ClassRef>
    <ClassRef>org.apache.cassandra.db.Truncation</ClassRef>
    <ClassRef>org.apache.cassandra.exceptions.TruncateException</ClassRef>
    <ClassRef>org.apache.cassandra.service.TruncateResponseHandler</ClassRef>
    <ClassRef>org.apache.cassandra.cql3.statements.TruncateStatement</ClassRef>
  </ClassRefs>
  <Tasks>
    <Task></Task>
  </Tasks>
  <CommenceDateTime>2012-06-22T12:00:00</CommenceDateTime>
  <CompletionDateTime>2012-06-27T12:00:00</CompletionDateTime>
</Feature>
```

Figure 7.1: Example of Cassandra's Features being represented in the XML format of the earlier-developed prototype tool, ScrumCity. Note that other data elements such as Tasks and Dates/Times are not relevant to this study.

With the required meta-data ready, an eclipse environment (see Table 7.1) was prepared with the ScrumCity plugin, and the source code corresponding to Cassandra's version 2.2.0, along with its features XML file, were imported into the workspace. Figure 7.2 depicts the resulting scene produced by ScrumCity's visualiser, showing a city landscape representation of Cassandra's system exposing its code structure to the user. The GUI panel on the right side displays the list of user stories (or features) grouped by parent Sprint and parent Release respectively. The pink-illuminated buildings (representing classes) correspond to the set of classes that are related to the particular Sprint that is selected—in other words it shows the implementation locality of those original 48 features[100].

Table 7.1: Environment Specification of Phase One Evaluation (inc. Cassandra Case Study)	
Machine	HP Elitebook 840 G5/Intel Dual Core i5 2+, Intel HD Graphics 620, 3 MB cache and 2 cores, Windows 10 Pro 64
Eclipse	Eclipse IDE for RCP and RAP Developers. Version: 2017-10-02 Release 4.7.0 (Oxygen)

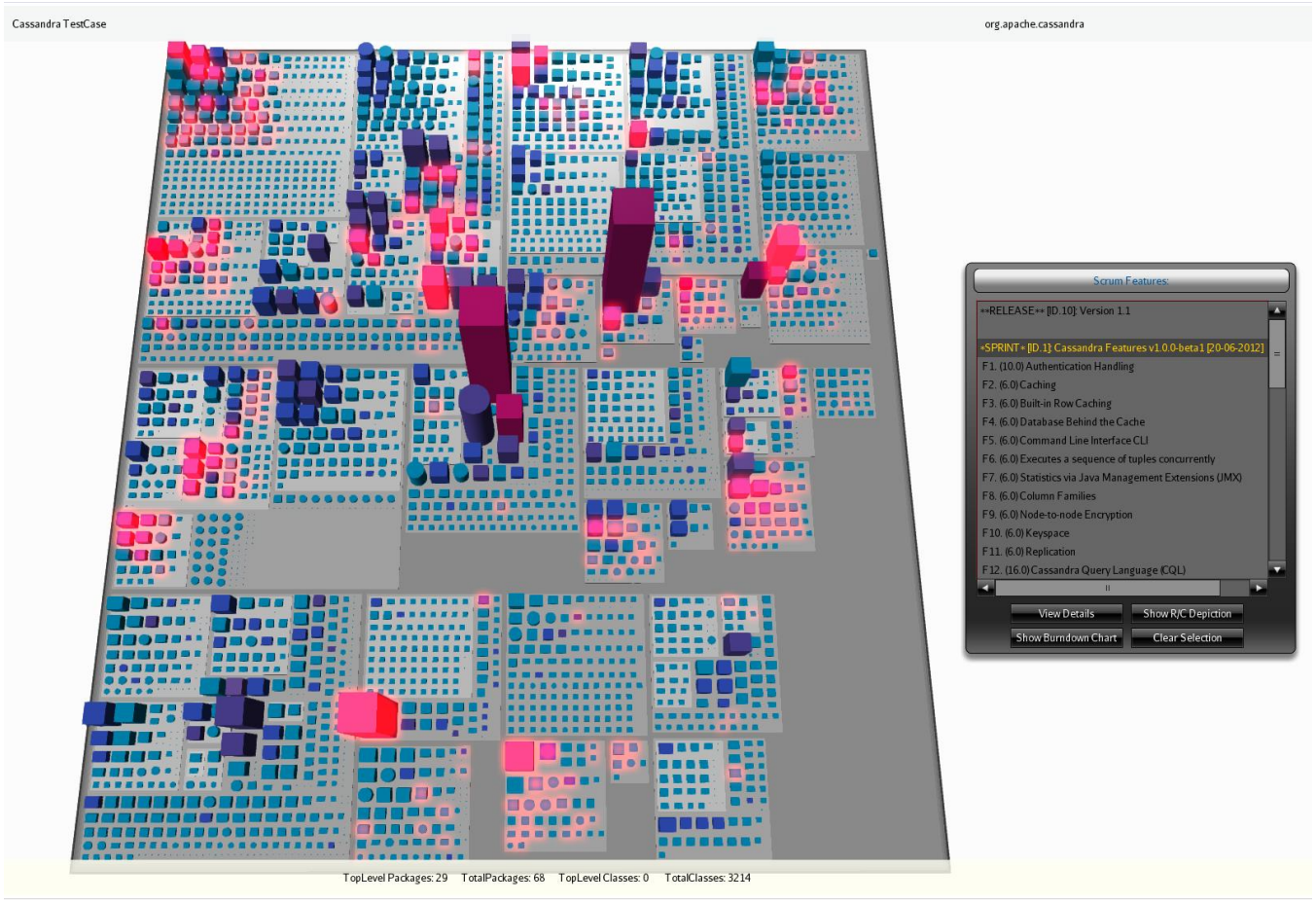


Figure 7.2: A visualisation of Cassandra system as appearing in the prototype Eclipse plugin, ScrumCity, ready for expert interview demonstrations.

[100] For a full description of ScrumCity prototype tool, its features, and implementation details, the reader is referred to (Alshakhouri 2013) for an elaborated description, and to (Alshakhouri, Buchan, and MacDonell 2018) for a concise coverage.

7.2.2 Usage Scenarios—An Early Examination of Value and Efficacy for Real-World Tasks

This section presents and explores some of the envisioned usage scenarios that could benefit from this research. These scenarios translate to real world activities and tasks commonly undertaken by agile practitioners and form the basis for conducting the case study and expert interview sessions that are covered shortly after. It is worthy to highlight that these components of preliminary evaluation—the usage scenarios, the case study, and the interview sessions-had been featured in our earlier cited work (Alshakhouri, Buchan, and MacDonell 2018). They are being readapted here to give the reader a perspective of the early evaluation conducted to assess the potentiality of this research. They serve to demonstrate how it could contribute to the enhancement of agile practice and likely lead to practical results. The activities covered illustrate potential benefits arising from integrating the design concepts—that is, design artefacts, to use a more pragmatic term—with their corresponding code artefact implementations, all in a unified, contextual, and interactive environment. Broadly speaking, the benefited tasks showcased here cover aspects of design exploration, inspection, and reasoning, but also feature applications of requirement-to-code traceability. Needless to say, they only present a very early—and rather premature—picture of the research potential. They are not to be taken as an exhaustive perspective of what the developed approach could possibly deliver, as it is still under its early research and advancement stage at this point. Being inspired by our earlier prototype tool, the scenarios necessarily reflect aspects of that particular implementation, but this should not lead the reader at this stage to any presumptions about the prospective technique that is to unfold at the end of this work.

Scenario 1—Identification of Original Purpose and Design Intent. This scenario demonstrates how refactoring activities can benefit from the introduced technique by enabling developers to quickly identify, inspect, and learn about the original design intents (represented by user stories in this case) behind a given system artefact. It could aid in discovering and assessing potential impact by a prospective modification activity. Given an existing class *C1* that dates back to earlier versions of a system, a developer *D1* who is tasked with refactoring the code for quality purposes needs to quickly identify C1's original intended functionality. The user selects the concerned project folder and launches the Visualiser tool, resulting in an interactive visualisation scene being presented to them depicting the entire project as a 3D city. Using the search functionality, D1 looks up C1's name which allows them to instantly locate its metaphorical building from among the other city's buildings—as it is now illuminated in a distinctive glowing red colour. The user is next transported automatically to that building in a manner aimed at promoting retention of the object's locality (using visual cues). Selecting

C1's building results in all related user stories to be highlighted in the accompanying Development Artefacts Explorer (labelled 'Scrum Features' in the UI). D1 can now choose one of the highlighted user stories and invoke the Details pane, allowing them to inspect and study the feature's description, its development, and activity information in context (see top & lower parts of Figure 7.3). The Details pane may include anchors to other classes or methods that are relevant to the selected user story (if any), which the developer can further review to fully understand C1's original purpose and its potential inter-dependencies.

Scenario 2—Pre-emptive Change Impact Analysis. This scenario demonstrates how the technique could inform a developer about possible impact on system artefacts before a planned change activity is actually carried out. A developer *D2* needs to implement some new functionality that requires her to modify method *M1*. Using similar steps as in Scenario 1 to locate M1, and before attempting any modification, D2 clicks on the method's metaphorical object presented to her in the scene, which instantly reveals in the artefact explorer the particular user story/ies that M1 is related to. D2 can now scroll through the revealed user stories, examining and inspecting each of the potentially affected functionalities before proceeding with any actual modification. In similar fashion to before, the user can read the description of those user stories in place without leaving the environment, and can select each to further reveal their own related code artefacts—allowing a controlled snowballing of traceability traversal if desired.

Scenario 3—Design Reasoning. This scenario presents a view that is expected to help a user decide if certain refactoring activities might be favoured, through the visual highlighting of abnormalities in implementation. A quality engineer *Q1* would like to identify and assess the contribution of Sprint *S1* to the system. Q1 selects the S1 tree node from the Development Artefact Explorer, which results in various buildings of the city to be illuminated in response. Q1 can now see a global perspective of S1's implementation and how it is distributed across her software structural landscape—hence better informing her decisions (see Figure 7.3 (top) and Figure 7.4 (b)). For instance, this view can reveal to her that S1's implementation is agglomerated in two particular modules, whereas, according to her knowledge of the functionalities being delivered by this sprint, she would instead have expected the implementation to be structured across four specific modules. This could alert her to undesired coupling that needs to be refactored. Another example can be demonstrated at the individual user story level. Suppose a specific user story requires database access and, by selecting it in the explorer, the engineer fails to see any illumination in the specialized DB module. This could indicate that the developer had potentially accessed the database directly instead of utilising the specialised DB module—hence, a refactoring would be needed to restore the quality and integrity of the codebase.

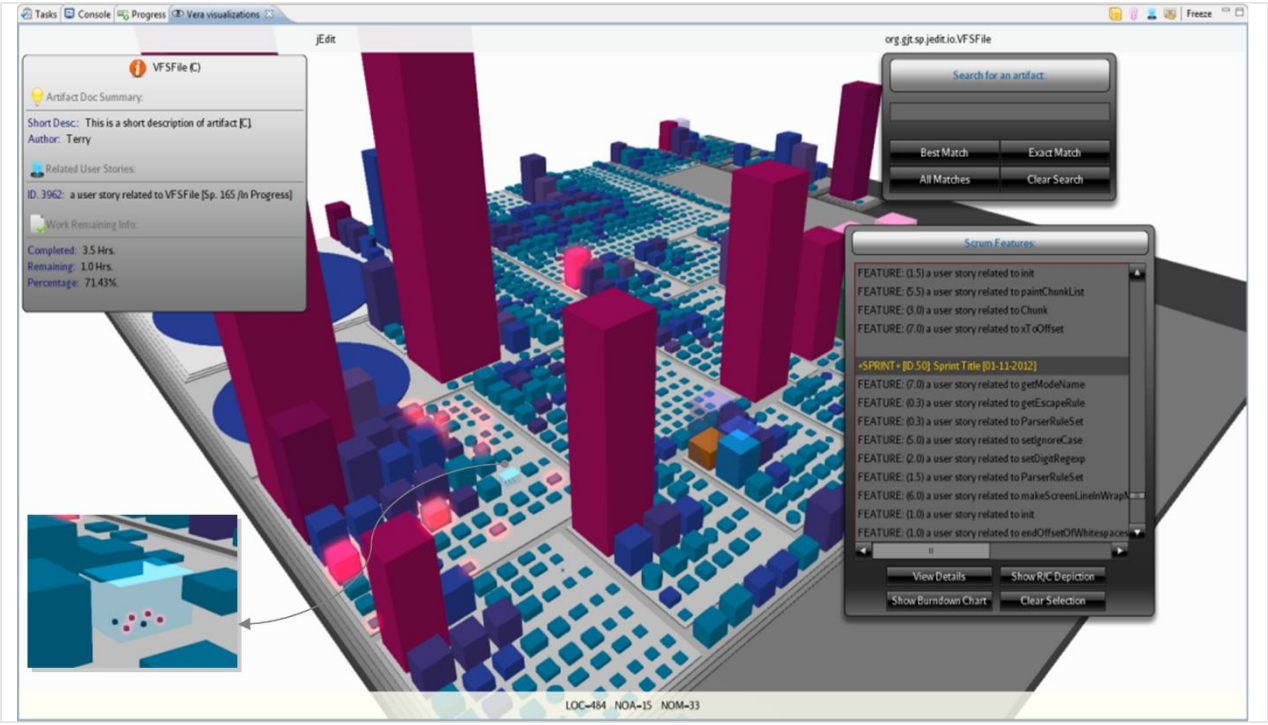
Such views can be generated for multiple Sprints at once, for multiple features (or user stories), or even an entire release.

Scenario 4—Monitoring of Structural Evolution, Process Implementation, and Development Progress. A manager G1 would like to inspect which parts of the system are being modified by his team while they are working on a given Sprint, S2. Following similar steps as in Scenario 3, G1 selects the S2 tree node in the Development Artefact Explorer. This triggers the visualisation into presenting a wide perspective view of the software city with those affected artefacts now all illuminated in red colour, allowing the manager to visually and readily identify the exact code artefacts that are being modified as a result of S2's implementation. It also gives a sense and a mental image of the new structural changes/evolution of the system's architecture, right in the context of prior existing components.

Scenario 5—Feature Location. This scenario addresses the well-known feature location problem, showing how the early concept underpinning this research makes its fulfilment straightforward. A developer D3 is tasked with enhancing a certain functionality F1 of a system. D3 needs to identify the specific code artefacts contributing to the implementation of F1. Using relevant keywords, D3 performs a search in the artefact explorer, which reveals two user stories. Together, they encompass the functionality he needs to enhance. D3 selects both user stories (multi-select) resulting in particular buildings lighting up in the city. D3 has now an illuminated view of where F1 is exactly implemented in the system (in what module(s) and in which classes), and can also inspect the textual details and description of each user story right in the context of the visualised system. He can point to any building to reveal its fully qualified name, and can choose to navigate directly to its source code. If individual methods rather than an entire class were linked to one of those user stories, then the building of the parent class would turn transparent, allowing the individual method blocks inside to shine through their class building. This way, D3 has now successfully identified all the code artefacts that he needs to study before adding the new enhancement to F1.

Scenario 6—Promoting Better Communication. This scenario demonstrates the concept's postulated potential in supporting team and stakeholder communication. A Scrum Master SM1 is chairing a Sprint Review event in which the development team (or teams in case of parallel Sprints), stakeholders, and the Product Owner are participating. A key goal of these events is to report on sprint progress and to highlight the product backlog items that were or were not completed. Using a large format screen, SM1 launches the Visualiser tool, presenting the team with a virtual city depicting the structure of their system in its latest status. She then places adjacently a screen capture of the system

from the last Sprint Review session, offering the participants a visual reference to inspect the latest structure against the earlier view (the former of which now shows some larger buildings and some other new buildings). She then selects the particular sprint tree node (or tree nodes in case of multiple sprints) in the Development Artefact Explorer, resulting in the highlight of all buildings (code artefacts) that were produced or impacted by this particular sprint—giving the whole team an overview of the scale and 'locality' of this sprint in the wider city structural landscape. SM1 is now equipped with an up-to-date visualisation of system artefacts, through which she can convey the latest status of the product and put her discussions into context; she can designate where this sprint has contributed and she can compare against an older view of the system.



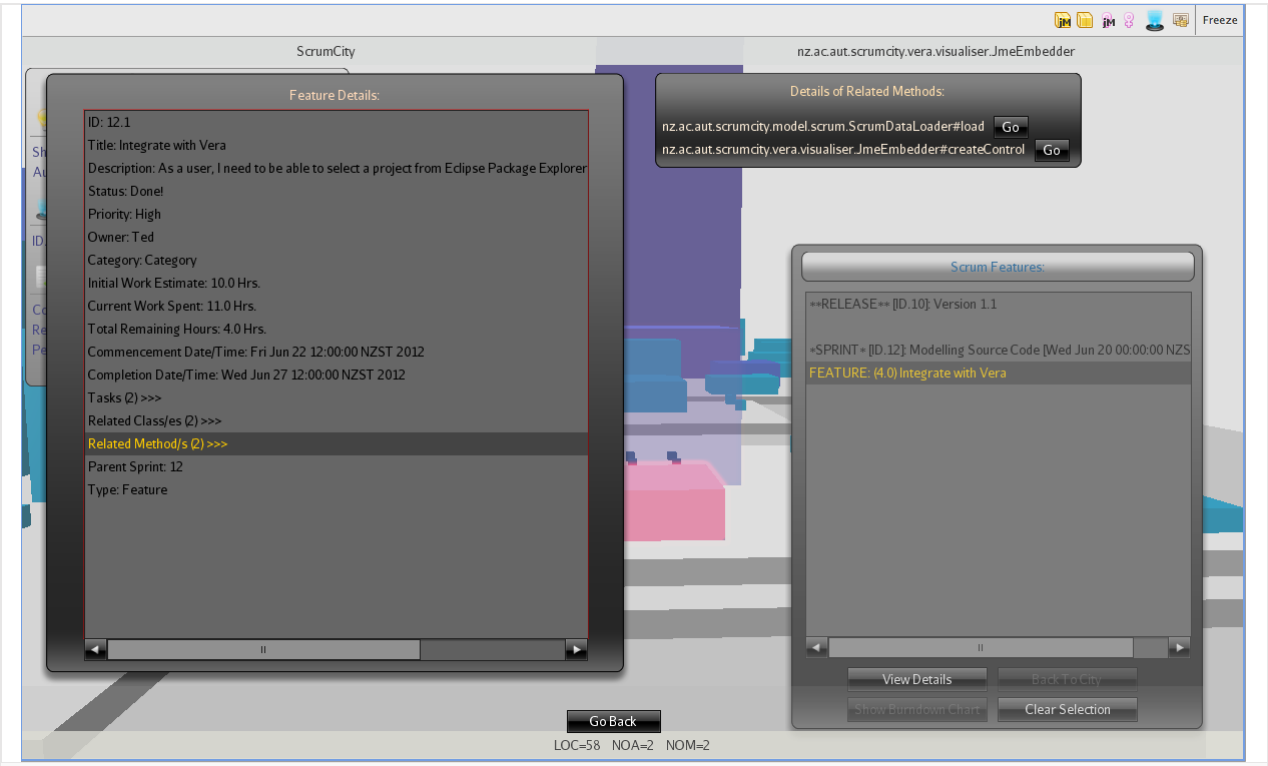


Figure 7.3: Top (previous page) and lower pictures show the potential capabilities and functionalities as demonstrated in the earlier prototype tool, ScrumCity. Adapted from (Alshakhouri 2013).

7.2.3 Discussion

The Cassandra Case Study is seen as an important opportunity in that it allowed for a sound and real-world verification of the capabilities and practical utilities of the proposed concept in its early form before further development. It serves to reveal the potential for the agile practitioner community in a testable form using real-world development data that experts can actively examine and relate to. The objective of this case study is to investigate if the visualisation technique would allow a user to discover new information about the dataset, arrive at any possible conclusions or decisions, and if it could indeed assist them if they were to undertake some common development tasks.

Analysis and Reflection:

The following paragraphs discuss key examples of the findings with the intent of examining and arguing for the above premises. They are largely inspired by the usage scenarios presented earlier, which are now possible to be tested with real-world data. The case study results in some interesting findings and demonstrates early promise in terms of delivering real benefits to practitioners. The findings of this laboratory investigation further come to form the basis behind a set of problems designed for conducting a preliminary usability evaluation, in the form of individual expert interview sessions.

Design Reasoning, Sprint Locality, and System Exploration. As was revealed earlier, the first prominent characteristic that is readily evident from Figure 7.2 is the locality of those 48 original feature requirements that are now distinctly visible across the system. The user gets immediate feedback on how these old features (*corresponding to version 1.0.0-beta1*) are now distributed across the latest version of the system. Such a view could potentially inform design and quality-related decisions and could be particularly valuable for Sprint Review sessions (see the next point). Managers or product owners are able to identify and relate to the particular system components that are impacted by this particular Sprint. They can further perceive—and relatively assess—the effort and the contribution size of that sprint with respect to the overall system—in that a building size indicates class size, and hence its potential complexity.

Feature Location (or Feature Traceability) and Mental Recall. Using the visualisation, we were able to readily identify the exact code artefacts that were implementing a given feature. For instance, the code artefacts implementing the basic 'Caching' functionality (labelled F2 in Figure 7.4(b)) are instantly revealed upon selecting that feature from the Development Artefact Explorer[101]—they are highlighted as seven pink-glowing buildings. The user has not just identified the specific code artefacts implementing the concerned feature, but they are now presented with an in-context visual representation of where those code artefacts are situated with respect to the system's overall structure (i.e., revealing locality of feature implementation). Moreover, prior research has demonstrated that the city-like metaphor feels familiar to the human mind, allowing the user to naturally relate to the resulting view, which stimulates the formation of mental images leading to better cognition and future mental recall of those particular entities and their locality within the system's architecture (Fourtassi, Rode, and Pisella 2017; Patterson et al. 2014).

In-context Inspection, Pre-emptive Change Impact Analysis, and Dependency Identification. If a developer needed to debug or add an enhancement to the Caching functionality, she would know exactly what class needs to be inspected before carrying out her changes. Further, she could conveniently proceed to inspect those classes on the spot by simply selecting a class' building and choosing to navigate to its source code file, all within the IDE platform. Further, if the developer decides that she needs to modify the *ConcurrentLinkedHashCache* class (which is one of those revealed seven buildings), she can first check if this change could impact other functionalities by simply selecting this class' building in the visualisation, which would then reveal two features being selected in the

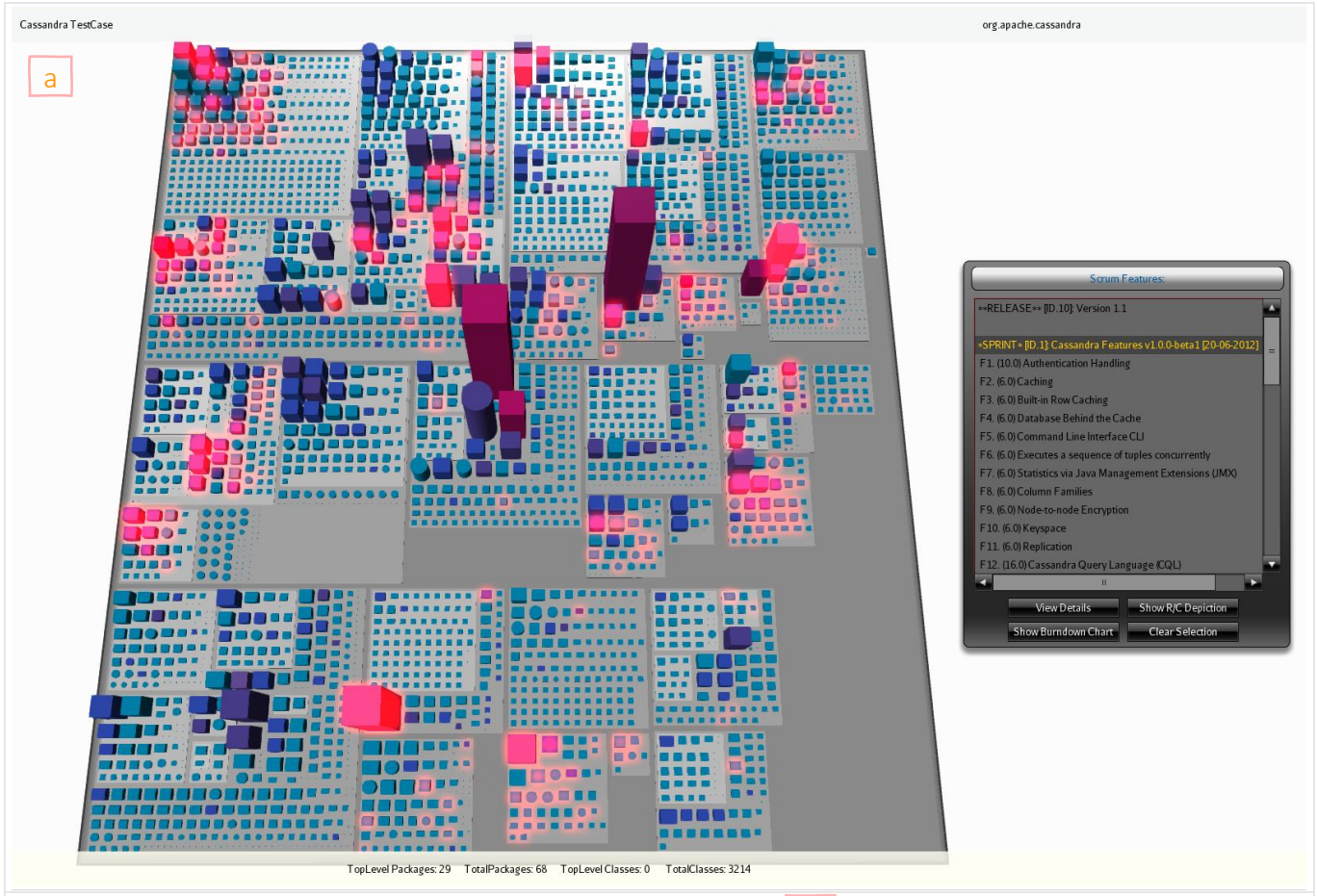
[101] Named Feature Explorer in the prototype

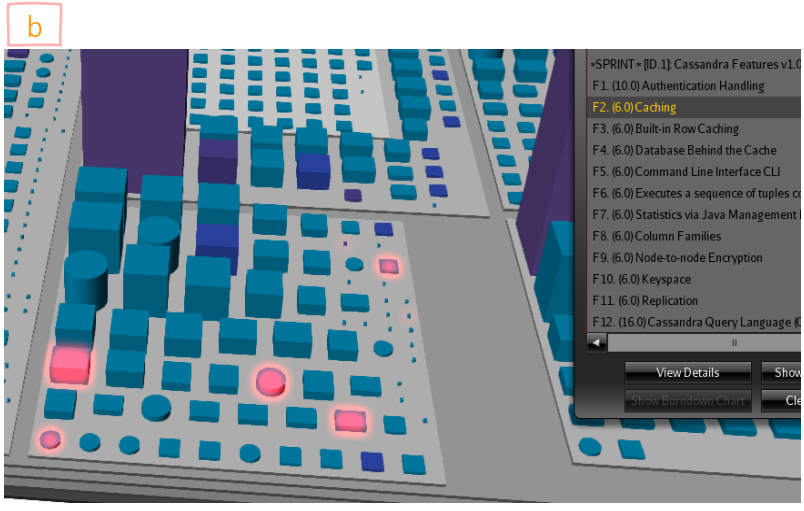
feature explorer[102] (see Figure 7.4(c)), also note the darker pink building representing the now selected class). In addition to the caching functionality that was already known, the developer can now see that this particular class is also involved in F4 (Database Behind the cache) implementation and thus care must be taken to ensure modifications do not negatively impact this second feature.

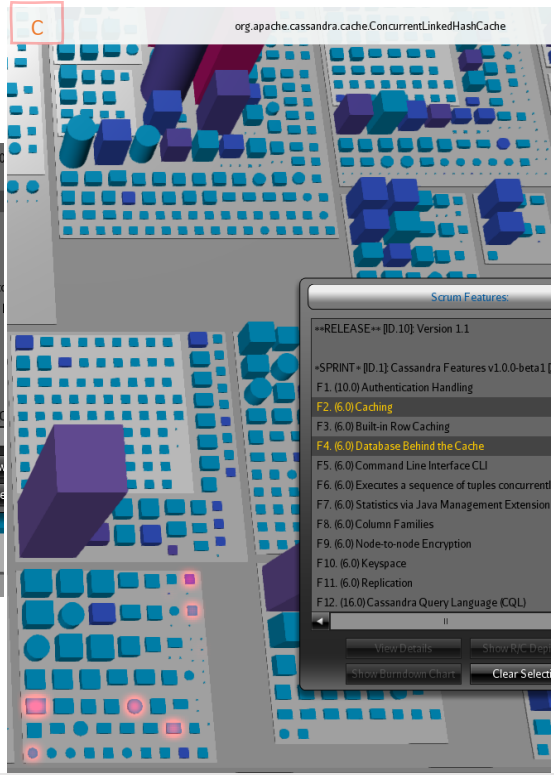
Exposing Complexity, and Effort Estimation. In a re-examination of Figure 7.4(b), a team leader or an analyst could recognise that the implementation of the 'Caching' feature is isolated in a single small package with a handful of small-sized classes, leading consequently to the conclusion that it is of low complexity and should hence require relatively low effort to enhance or debug. In comparison, other features such as F16 (Column Expiration) shown in Figure 7.4(f), as well as F42 (Ec2 Multi Region Snitch) (not shown) are seen to be very simple functionalities that are each realised by a single and independent small class that does not impact any other class or feature. In a similar mechanism, an analyst investigating the 'Marshalling' functionality (F25) can observe the considerable complexity of this feature, which is implemented by a relatively large number of classes as shown in Figure 7.4(e)—which consequently indicates a potentially higher maintenance effort. An interesting finding is also observed in feature 9 (Node-to-Node Encryption) where implementation is realised across three modules, possibly an indication of a potential interdependency or coupling (see Figure 7.4(d)).

Closing. Thanks to the earlier developed prototype, the above findings serve to demonstrate some of the practical utilities of the visualisation technique underpinning this work in a real-world setting, albeit in its preliminary form. As remarked earlier, ScrumCity did support method-level traceability links but this feature could not be tested on the Cassandra dataset as it only contained file-level mappings. It is worthy to emphasise again that, as this research has revealed, such tracelinks are hardly maintained in a disciplined manner by today's practicing community—let alone, at such lower granularity levels. Yet, the value of those tracelinks is well-established in literature, which begs the question of why it is not embraced by developers. As, will be shortly revealed, the feedback from industry users seems to indicate strongly that the problem is largely due to present feasibility and practicality of such a practice. As has been discussed in earlier chapters, a key objective of this research is to enable and promote such practice by providing a practical solution that developers could begin to consider and adopt.

[102] selecting a class would automatically highlight all related classes and all related features


Cassandra TestCase
org.apache.cassandra
a
Scrum Features:
RELEASE [ID.10]: Version 1.1
SPRINT [ID.1]: Cassandra Features v1.0.0-beta1 [20-06-2012]
F1. (10.0) Authentication Handling
F2. (6.0) Caching
F3. (6.0) Built-in Row Caching
F4. (6.0) Database Behind the Cache
F5. (6.0) Command Line Interface CLI
F6. (6.0) Executes a sequence of tuples concurrently
F7. (6.0) Statistics via Java Management Extensions (JMX)
F8. (6.0) Column Families
F9. (6.0) Node-to-node Encryption
F10. (6.0) Keyspace
F11. (6.0) Replication
F12. (16.0) Cassandra Query Language (CQL)
View Details
Show R/C Depiction
Show Burndown Chart
Clear Selection
TopLevel Packages: 29 TotalPackages: 68 TopLevel Classes: 0 TotalClasses: 3214


b
F1. (10.0) Authentication Handling
F2. (6.0) Caching
F3. (6.0) Built-in Row Caching
F4. (6.0) Database Behind the Cache
F5. (6.0) Command Line Interface CLI
F8. (6.0) Column Families
F9. (6.0) Node-to-node Encryption
F10. (6.0) Keyspace
F11. (6.0) Replication
View Details


c
org.apache.cassandra.cache.ConcurrentLinkedHashCache
Scrum Features:
RELEASE [ID.10]: Version 1.1
F1. (10.0) Authentication Handling
F2. (6.0) Caching
F3. (6.0) Built-in Row Caching
F4. (6.0) Database Behind the Cache
F5. (6.0) Command Line Interface CLI
F8. (6.0) Column Families
F9. (6.0) Node-to-node Encryption
F10. (6.0) Keyspace
F11. (6.0) Replication
F12. (16.0) Cassandra Query Language (CQL)

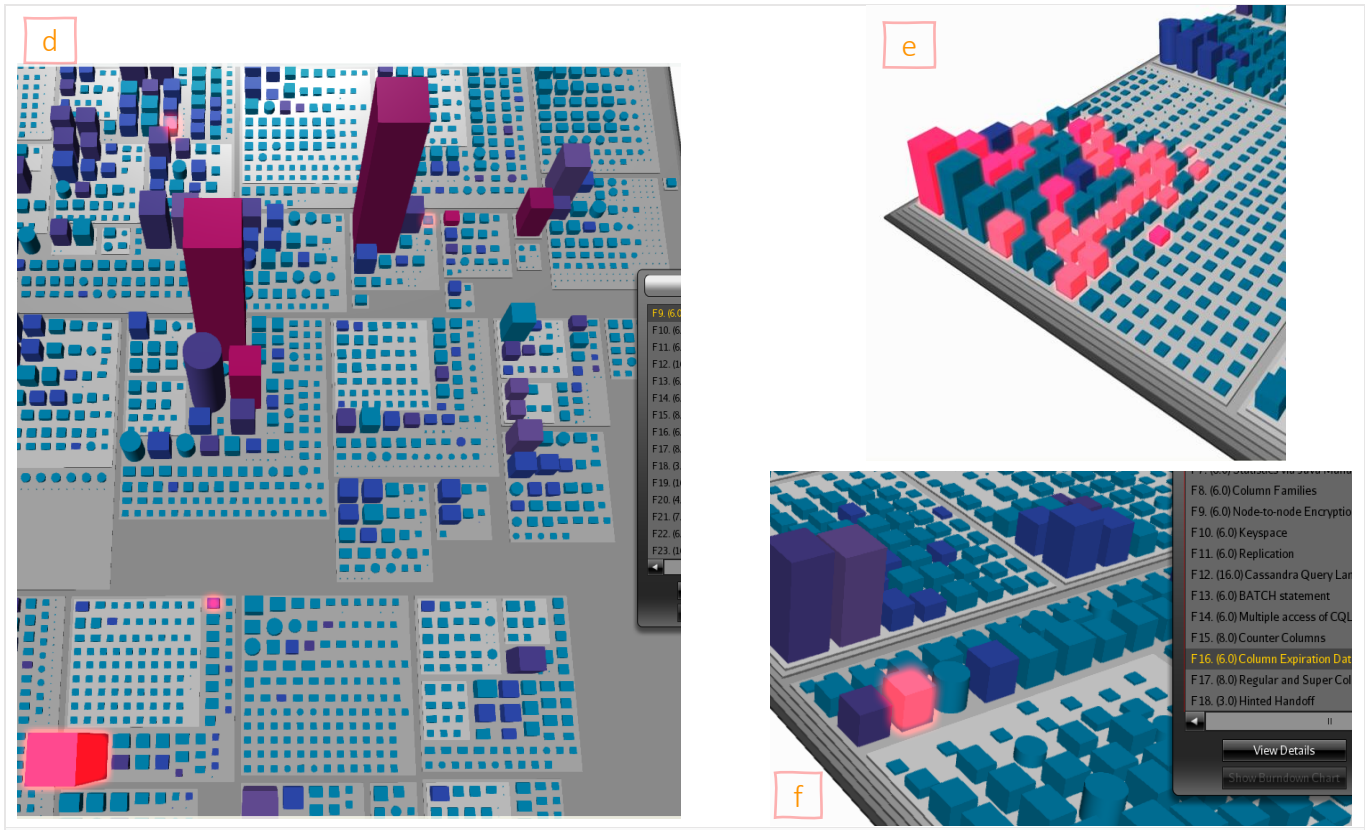


Figure 7.4: Left to right from top: **a)** Visualised Cassandra source code, with the original 48 requirement features mapped to their code implementations, exposing their implementation locality as seen across version 2.2.0 of the system, **b)** selecting the 'Caching' feature in the explorer reveals the code artefacts implementing it, **c)** selecting the `ConcurrentLinkedHashCache` class in the visualisation reveals two related features in the explorer, **d)** dependency and coupling can be identified in the implementation of feature 9 (Node-to-node Encryption), **e)** complexity (and consequently maintenance effort) is easily identifiable for feature 25, which is implemented by 29 classes, and for feature 16 that is implemented by a single class (**f**). Adapted from (Alshakhouri, Buchan, and MacDonell 2018).

7.3 Second Mini-Case Study

This case study was conducted as part of Phase Three evaluation activities (see section 8.7 in next chapter). Its purpose was to verify and test the functionalities of the research tool, AgileInsight, at the end of the development cycle, and before conducting the demonstration interviews with industry experts.

7.3.1 Dataset Research and Selection

Part of this research involved investigation carried out to look for suitable datasets from the extant literature. A comparable situation to that faced in phase one had to be navigated during phase three as well in order to provide a suitable real-world dataset. However, the amount of work expended this time was much larger, and was quite involved. Academic datasets where the three conditions required (original requirements in small actionable units, product source code, and traceability links) are met, were still scarce in the literature. A number of works were examined in an attempt to locate a suitable dataset. It was not desirable to reuse the Cassandra dataset that was the subject of the study in phase one, as it did not have method level tracelinks. It also had a relatively small set of features—48 only. Fortunately, two data showcase papers on the specific subject of requirements traceability datasets were published close to the time of this stage of the study. The datasets featured in those papers were examined and assessed, and the most appropriate one was selected. The selected dataset is named iTrust, and the next section presents more detail on its assessment, and the process of preparing it for the study.

While the effort of the dataset research activity is considered to be valuable to other researchers, it probably constitutes less importance to most readers. Hence, interested readers are kindly referred to Appendix IV to learn more about this part of the research.

However, a point of interest to be reported here is that all datasets encountered were exclusively confined to the Java language[103]. While this could be a natural result due to the language's popularity, the latest indicators from development communities suggest that other languages are quickly taking over in popular use. Representing other languages in research datasets will certainly be of importance and interest to researchers in the near future. It would certainly be of particular interest to this work as well, as it would have allowed us to demonstrate and evaluate the traceability and the visualisation

[103] The only uncertainty was around the list provided by (Zogaan et al. 2017), as they did not include the language datapoint in their classification. But through the naming and prior experience, at least the majority could be confirmed as Java systems.

mechanisms of AgileInsight outside the Java language—which has strongly dominated software visualisation research, and apparently so for the late research in traceability[104]. Another remark to be highlighted, is that the scarcity of the method-level tracelinks in datasets—and the fact that researchers had to pay developers to manually reconstruct this knowledge—appears to suggest that it is impractical and infeasible for developers to declare tracelinks at this lower level of granularity using currently existing tools. This is an area where AgileInsight (through its IDE-integrated active collection mechanism) could possibly work towards enabling such functionality.

7.3.2 iTrust—Selected Dataset, Preparation and Loading

iTrust is an open source Java-based web application maintained by the Realsearch Research Group at North Carolina State University. The codebase consists of Java and JSP files, along with JavaScript, HTML, and CSS for the front-end side.

For the purpose of this evaluation, two research datasets of iTrust are used as follows:

- **iTrust Dataset A:** this dataset is obtained from the CoEST[105] repository (see Appendix IV).
- **iTrust Dataset B:** this dataset is made available by (Hammoudi et al. 2021)[106].

Each dataset consisted of requirements defined as use cases, and tracelinks connecting each use case to its source code implementation. Dataset A included the source code of iTrust v10, while Dataset B only pointed to a public repository containing the source code[107]. As discussed earlier, a dataset with tracelinks at different levels of granularity is desired for the evaluation. Dataset A provides tracelinks at file-level only, while dataset B provides the tracelinks at method-level only. To best demonstrate and evaluate the work, it was decided to merge both datasets. Before explaining how the merging was accomplished, it is helpful to quickly examine an example entry from each dataset, which is provided in Table 7.2.

Table 7.2: Example entries from dataset A and dataset B. Note the requirement (source) difference between the two, along with the difference in mapped targets. (Not all target artefacts are listed for Dataset B)

Dataset A Example Entry	**Requirement (source):** UC3S2 **Requirement text:** If answer to security question is correct, allow user to change their password. An email notification is sent.

[104] The Dronology dataset appears to utilise both Java and Python languages, and it would make an interesting opportunity when the dataset becomes available with source code.
[105] http://coest.org/
[106] https://doi.org/10.5281/zenodo.4453526
[107] The paper pointed to the repository: https://sourceforge.net/projects/itrust, and while authors did not indicate the exact version they used, best effort at determining it seemed to indicate they used v21.0.

	Code Artefact (Mapped targets):
	`ResetPasswordAction`
	`AuthDAO`
	`EmailUtil`

Dataset B Example Entry	Requirement (source): UC3
	Requirement text: Not provided.
	Code Artefact (Mapped targets):
	`AccessDAO.setSessionTimeoutMins()`
	`ResetPasswordAction.checkRole()`
	`ResetPasswordAction.getSecurityQuestion()`
	`AuthDAO.getUserRole()`
	`AuthDAO.resetPassword()`

Analysis. The iTrust dataset in general has about 40 functional requirements that are defined as use cases. Each use case is further broken down into main sub-flows (labelled as S) and alternative sub-flows (labelled as E). As Table 7.2 might have already revealed, the way these 'source' use cases are mapped to the 'target' artefacts is slightly different among the two datasets. The following brief analysis seeks to explain this along with a few other key differences that are of interest:

Dataset A:

- The mapping to source artefacts is provided at the main sub-flow, the alternative sub-flow, and sometimes at the main use case level (when it has no sub-flows).
- The naming of source artefacts is indicative of the mapping level provided, for example: UC3 indicates main use case level mapping, UC3S1 indicates sub-flow mapping, and UC3E1 would indicate alternative sub-flow mapping.
- In total, dataset A has **131** sources—all three levels combined
- It also has a total of **534** tracelinks to Java and JSP source files (targets). Some were, however, found to be duplicate, and after removing those the total came to **399** tracelinks.

Dataset B:

- The mapping to source artefacts is provided at the main use case level only. This means the source artefact will always have a naming in this format UC3, UC5, and so on, but not UC3S1 or UC3E1.
- Has a total of **34** sources only—that means 34 use cases out of the original 40. This is reflecting the fact that some use cases have been dropped from development.
- Has a total of **166838** tracelink entries to Java methods (targets). No mappings to JSP files are provided.
- The large number of tracelink entries is due to the authors' unique mapping which is discussed in Appendix IV, where each single method has a mapping entry to each single use case. A conditional value determines then if that entry is a trace, a no trace, or undefined.

- After an automated extraction, the truthy tracelinks came out to only 307 mappings.
- Considering its recency, dataset B unsurprisingly has tracelinks to additional new classes that may not necessarily exist in dataset A. (Note the `AccessDao` class in the example given in Table 7.2)

Merging the Two Datasets. Since the sources in dataset A could appear at any of three levels (main use case, main sub-flow, or alternative sub-flow), while the sources in dataset B are only provided at the main use case (UC) level, an approach had to be deliberated in order to find a proper way to merge the two datasets. In particular, the sources in the two datasets must be mapped to each other so their targets can then be combined. Three approaches were considered in this regard:

- Option 1: map each source from dataset A to all corresponding sources in dataset B. This is the most correct approach to use as it would generalise all those lower level sources into their parent UC. Unfortunately, dataset B does not provide the actual text of the top level use cases (none of the provided links to the website had worked). This option is thus not feasible.
- Option 2: map each source from dataset B to all corresponding sources in dataset A. This would result in spreading the top level UC sources across the lower level sub-flows—the opposite of option 1.
- Option 3: implement a selective mapping, as follows:
    1. map top level UC sources from dataset B to top level UC sources in dataset A, when such items exist in A—few do.
    2. if not, map UC sources from B to all main sub-flow sources (those labelled with 'S') in dataset A (e.g., UC3S1, UC3S2)—the majority of cases will fall in this category.
    3. Leave alternative sub-flow sources in dataset A as is—that is, do not map UC sources from dataset B to these alternative sub-flows. In other words, method-level mappings will not be represented for alternative sub-flows.

Decision. Option 3 was eventually selected and adopted for the merging process. The justification behind this was its realistic applicability, and the fact that it achieves the goal behind the purpose of the merging. The goal of the merging was not to produce accurate traceability reflection as much as demonstrating the multi-level tracelink capability of AgileInsight and its underlying approach. Reflecting method-level mappings in the main sub-flows is sufficient to accomplish this objective. Nonetheless, this does introduce a level of inaccuracy as not all method-level tracelinks from dataset B's top-level UC might necessarily apply to each lower-level sub-flow under that UC in dataset B. This is an acknowledged inaccuracy, however it was tolerated given that it does not impact the objective of the evaluation nor the scenario tasks involved. Another acknowledged consequence is the fact that by combining the file-level and the method-level tracelinks for a given source, it necessarily creates

undesired redundancy. That is, if a given UC is mapped to individual methods in a particular file, it is no longer of value—and rather defeats the purpose—to retain the file-level mapping. Nonetheless, this point was also tolerated given the same justifications above. While in this particular case, the file-level tracelinks are redundant, they still serve the purpose of demonstrating the multi-level traceability. More importantly, it simplifies the inclusion of Jsp targets, which are only provided in Dataset A. Additionally, it was noticed in a good number of cases that dataset B appeared to contain missing tracelinks to some Java target files[108].

Merging Outcome. After writing a simple algorithm to automatically implement the approach in Option 3, the net result of total tracelinks came to be **1274** (file-level + method-level tracelinks).

Merging Examples. In order to better demonstrate the merging approach adopted, two examples are given below with brief commentary.

Example 1:

Dataset A Mapping	Dataset B Mappings
{ UC25 => /auth/surveyResults } { UC25 => ViewSurveyResultAction } { UC25 => HospitalsDAO } { UC25 => SurveyResultDAO } { UC25E1 => /auth/surveyResults }	{ UC25 => ViewSurveyResultAction.getSurveyResultsForHospital()} { UC25 => ViewSurveyResultAction.getSurveyResultsForZip() } { UC25 => HospitalsDAO.getAllHospitals() } { UC25 => SurveyResultDAO.getSurveyResultsForHospital() } { UC25 => SurveyResultDAO.getSurveyResultsForZip() }
Our Combined Mapping after Merge	
{ UC25 => /auth/surveyResults } { UC25 => ViewSurveyResultAction } { UC25 => HospitalsDAO } { UC25 => SurveyResultDAO } { UC25E1 => /auth/surveyResults } { UC25 => ViewSurveyResultAction.getSurveyResultsForHospital() } { UC25 => ViewSurveyResultAction.getSurveyResultsForZip() } { UC25 => HospitalsDAO.getAllHospitals() } { UC25 => SurveyResultDAO.getSurveyResultsForHospital() } { UC25 => SurveyResultDAO.getSurveyResultsForZip() }	

*Note that the file-level mappings in dataset A point to either Java files (effectively classes) or to Jsp files—the ones where full path is given refer to Jsp files. Method-level tracelinks are not added to alternative sub-flow sources (highlighted in yellow).

Explanation. In this case above, the original tracelinks in A are primarily available in use case level mappings (i.e., UC25), with only one alternative sub-flow level (UC25E1). As per the merging criteria, the method-level tracelinks were mapped to the main use case level only. However, the majority of sources in A are provided at main sub-flow level (i.e., UC?S?), and occasionally at alternative sub-flow

[108] Admittedly, in some of those cases, this could be a simple result of future refactoring that was carried out.

level (i.e., UC?E?). In those cases, the merging algorithm will only add the method-level mappings to the main sub-flow sources. Example 2 below illustrates this process.

Example 2:

Dataset A Mapping	Dataset B Mappings
{ UC21S1 => /auth/hcp-er/emergencyReport} { UC21S1 => EmergencyReportAction} { UC21S1 => PatientDAO} { UC21S1 => AllergyDAO} { UC21E1 => /util/getUser} { UC21E1 => GetUserNameAction} { UC21E1 => AuthDAO } { UC21E2 => /util/getUser } { UC21E2 => GetUserNameAction } { UC21E2 => AuthDAO }	{ UC21 => EmailUtil.sendEmail() } { UC21 => EmergencyReportAction.getAllergies() } { UC21 => EmergencyReportAction.getBloodType() } { UC21 => EmergencyReportAction.getCurrentPrescriptions()} { UC21 => EmergencyReportAction.getImmunizations() } { UC21 => EmergencyReportAction.getPatientAge() } { UC21 => EmergencyReportAction.getPatientGender() } { UC21 => EmergencyReportAction.getPatientName() } { UC21 => GetUserNameAction.getUserName() } { UC21 => ViewMyRecordsAction.getReportRequests() } { UC21 => AllergyDAO.getAllergies() } { UC21 => AuthDAO.getUserName() } { UC21 => PatientDAO.getCurrentPrescriptions() } { UC21 => PatientDAO.getDiagnoses() } { UC21 => PatientDAO.getImmunizationProcedures() } { UC21 => PatientDAO.getPatient() } { UC21 => PersonnelDAO.getAllPersonnel() } { UC21 => ReportRequestDAO.addReportRequest() } { UC21 => ReportRequestDAO.getAllReportRequestsForPatient()}
Our Combined Mapping after Merge	
{UC21S1 => /auth/hcp-er/emergencyReport} {UC21S1 => EmergencyReportAction} {UC21S1 => PatientDAO} {UC21S1 => AllergyDAO} {UC21S1 => ReportRequestDAO.addReportRequest()} {UC21S1 => ReportRequestDAO.getAllReportRequestsForPatient()} {UC21S1 => EmailUtil.sendEmail()} {UC21S1 => EmergencyReportAction.getAllergies()} {UC21S1 => EmergencyReportAction.getBloodType()} {UC21S1 => EmergencyReportAction.getCurrentPrescriptions()} {UC21S1 => EmergencyReportAction.getImmunizations()} {UC21S1 => EmergencyReportAction.getPatientAge()} {UC21S1 => EmergencyReportAction.getPatientGender()} {UC21S1 => EmergencyReportAction.getPatientName()} {UC21S1 => GetUserNameAction.getUserName()} {UC21S1 => ViewMyRecordsAction.getReportRequests()} {UC21S1 => AllergyDAO.getAllergies()} {UC21S1 => AuthDAO.getUserName()} {UC21S1 => PatientDAO.getCurrentPrescriptions()} {UC21S1 => PatientDAO.getDiagnoses()} {UC21S1 => PatientDAO.getImmunizationProcedures()} {UC21S1 => PatientDAO.getPatient()} {UC21S1 => PersonnelDAO.getAllPersonnel()} {UC21E1 => /util/getUser} {UC21E1 => GetUserNameAction} {UC21E1 => AuthDAO} {UC21E2 => /util/getUser} {UC21E2 => GetUserNameAction} {UC21E2 => AuthDAO}	

*Note that the file-level mappings in dataset A point to either Java files (effectively classes) or to Jsp files—the ones where full path is given refer to Jsp files. Method-level tracelinks are not added to alternative sub-flow sources (highlighted in yellow).

Loading the Resulting Dataset. As explained in earlier chapters, AgileInsight does not use intermediary files or local databases, as all operations are performed in a live manner. This means design items (or sources, as they are known in traceability research terminology) are accessed and retrieved on demand directly from the agile dashboard of the user. Similarly, the code items (or targets, in traceability terminology) are accessed directly from the user's checked out local repository. All tagging operations are performed on the local repository and then pushed automatically to the remote instance. Thus, in order to prepare the dataset for the evaluation, the following had to be performed:

- A Trello board was created to host the iTrust design items
- A Github repository was created to host the source code[109]
- An algorithm was written to implement the merging mechanism introduced earlier and output the result of combined tracelinks.
- A small program was written to automatically load the design items into the Trello board, and then automatically trigger the three-legged tagging operation corresponding to each tracelink found for that design item.

Instead of executing the loading process all at once, it was decided to have it carried out in a controlled manner. This would also allow for some tracelink entries to be done manually with the experts during the running of the evaluation sessions. The loading program was thus modified to take the design item identifiers—that is, the use case IDs, as a comma separated user input from a text box. The process then only performs the loading for those particular use cases entered. As an unintended side benefit, this process can be now adapted easily to load any dataset in the future—regardless of the agile dashboard to load the data to.

Demos of Loading. A brief video demo of the automated loading process is available on the following link for the reader's reference. It could be of help in further illustrating the process[110].

- Video Demo: https://blaiski.github.io/agileInsight.page/#dataset-loader

Closing. With the merged iTrust dataset now fully loaded on Trello and Github, the case study can be carried out. This also sets up the necessary preparation for the evaluation sessions with the experts that is covered in next chapter (section 8.7).

[109] iTrust v21.0 was used here in order to match with the version that was apparently used in dataset B. The names of very few classes had to be restored though to their original naming.
[110] Two alternative video captures (in normal speed) can be accessed here [1]: https://www.dropbox.com/s/8zez6o8y8w682sp/UC10E1%20%26%20UC23E1.mov?dl=0, and here [2]: https://www.dropbox.com/s/5rxddikvae6gzqr/UC26E1.mov?dl=0. Note that the text files that appear in the videos are prints outs for verification purposes only—they are not part of the loading process.

7.3.3 iTrust—A very Brief Case Study

This section presents a mini-case study using the iTrust dataset that is focused on illustrating two specific aspects of AgileInsight—its visualisation capabilities, and its traceability application features. The visualisation aims to showcase the tool's core characteristics, such as exposing code structure, while the traceability features serve to showcase the application uses made possible thanks to synchronising the code with its original design intents.

The environment where the case study was conducted on appears in below table.

Table 7.3: Environment Specification of Phase Three Evaluation (inc. second mini-case study)	
Machine	MacBook Pro (Retina, 15-inch, Mid 2015). Processor: 2.5 GHz, Quad-Core Intel Core i7. Memory: 16 GB, 1600 MHz DDR3. Graphics: AMD Radeon R9 M370X 2 GB. Running macOS Monterey.
Vscode	Version: 1.62.3, Date: 2021-11-17, Electron: 13.5.2, Chrome: 91.0.4472.164, Node.js: 14.16.0, V8: 9.1.269.39-electron.0, OS: Darwin x64 21.2.0

Dynamic Mapping View 1: Linear Capped Classes with Normalised Methods

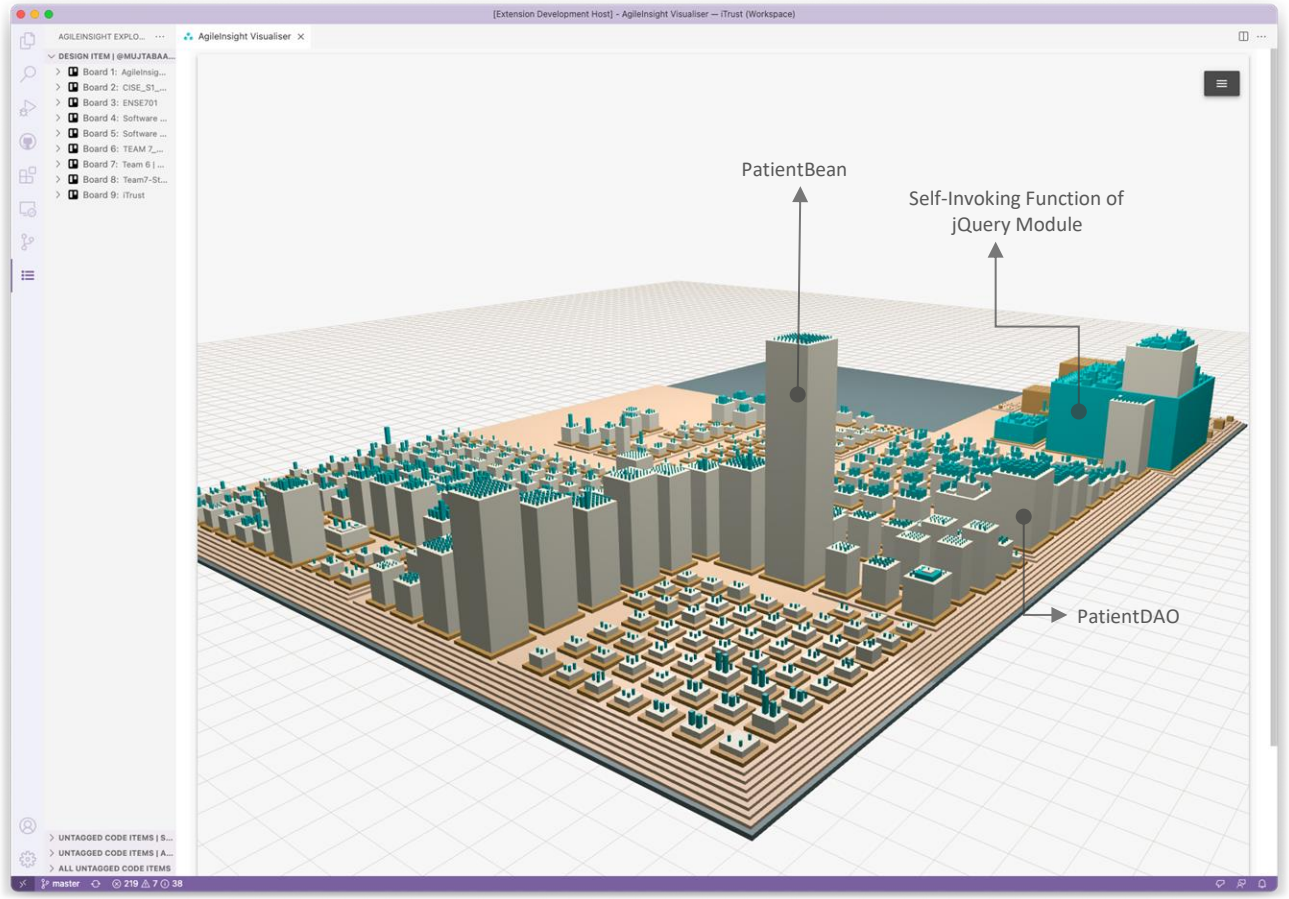


Figure 7.5: iTrust codebase as Visualised by AgileInsight using linear capped mapping for classes and normalised mapping for methods.

The above figure showcases the iTrust dataset as visualised by AgileInsight. In this view, the primary containers (classes in this case) are mapped using a linear function (see section 6.9.2), but with an enforced cap limit to keep buildings from growing disproportionally tall. The secondary containers (methods and functions, in this case) are mapped using a normalised function. In the visualisation, two main modules are distinctively identifiable, the main iTrust package containing the core Java system (front side), and the Webroot module containing the Jsp and other frontend files (CSS/JavaScript), which is shown at the back of the visualisation scene.

Dynamic Mapping View 2: Linear Capped Classes with Linear Methods

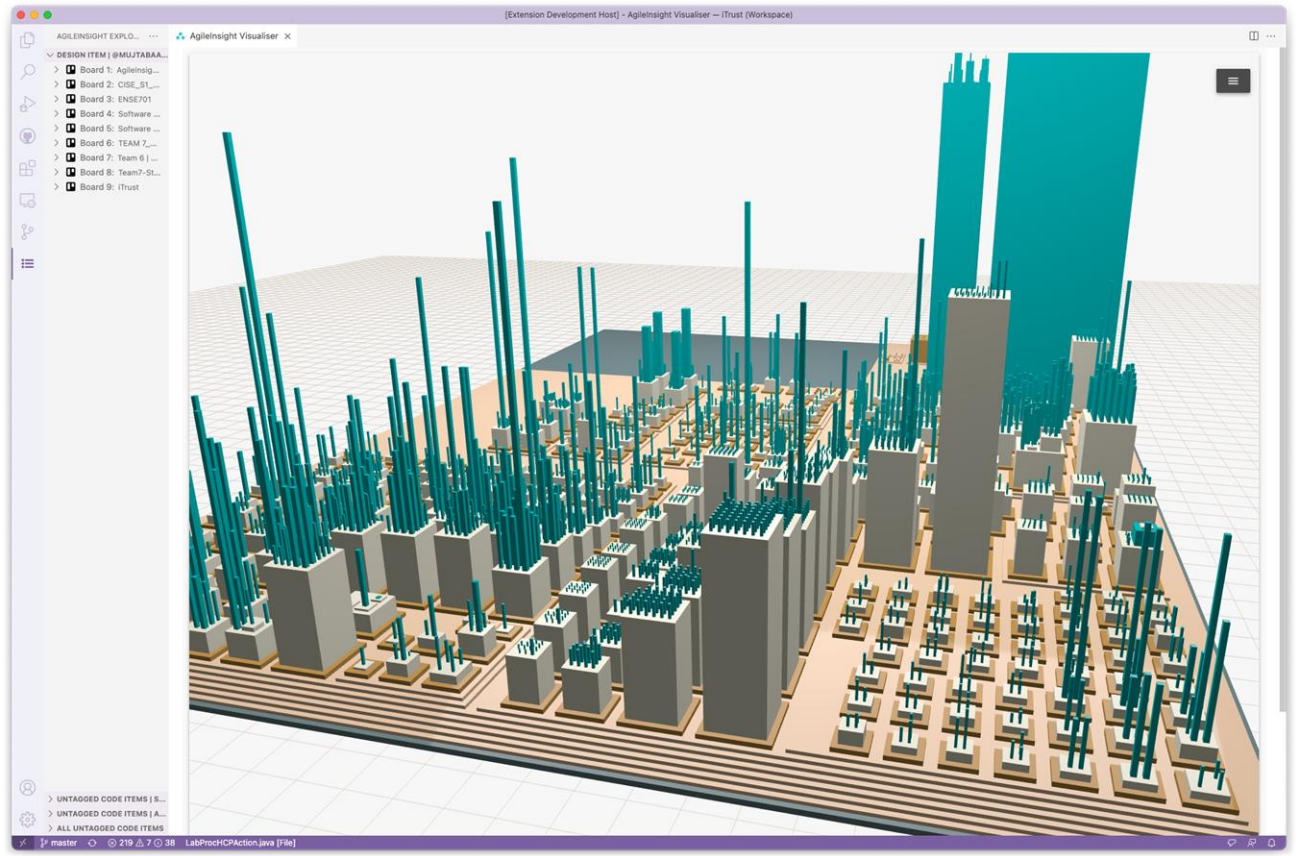


Figure 7.6: iTrust codebase as Visualised by AgileInsight using linear capped mapping for classes and linear mapping for methods.

The above figure shows the same iTrust dataset, but now the secondary containers are mapped using a linear function. The scene has now become cluttered, but the complexity of methods and functions is faithfully represented now. Functions in the JavaScript modules (see the Webroot module) convey a distinctively complex implementation. The largest two buildings at the back are jQuery and FaceBox JavaScript libraries (from right to left, respectively), and are only included in the visualisation for illustrative purposes.

Dynamic Mapping View 3: Normalised Mapping

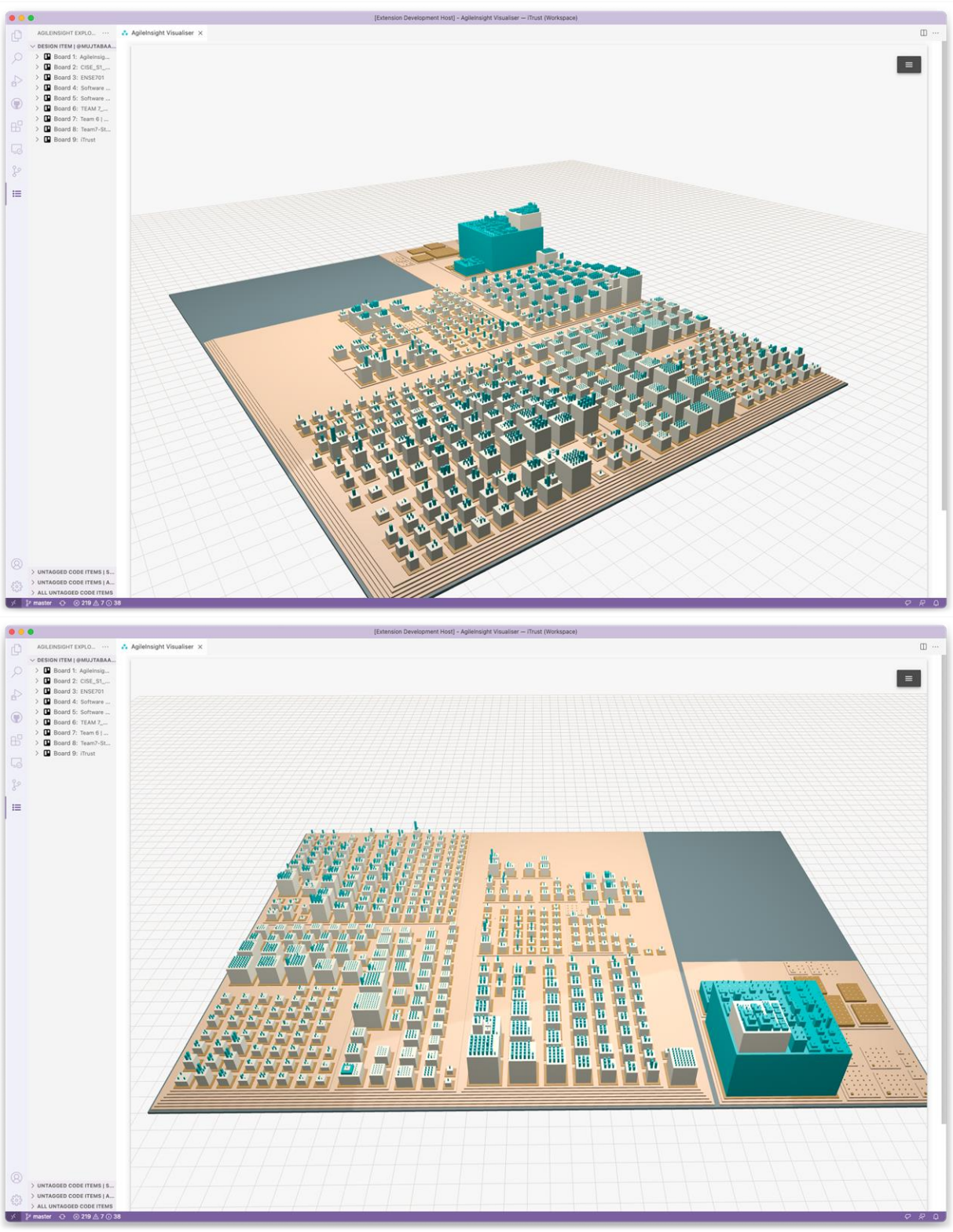


Figure 7.7: iTrust codebase as visualised by AgileInsight using normalised mapping for primary and secondary containers.

The above figures showcase the same iTrust system, but now using a normalised mapping function for both the primary containers and secondary containers. The scene now appears simpler with a more harmonised look. Buildings still bear a reasonable variance in height reflecting their varying complexity,

and the scene is more accessible now (due to less occlusion). The Webroot module is also clearly visible on the right side of second figure. Overall, the scene appears to expose the overall code structure of the iTrust system. Various modules and buildings are annotated for convenience. The Webroot module features Jsp, JavaScript and CSS files.

Similar Structures

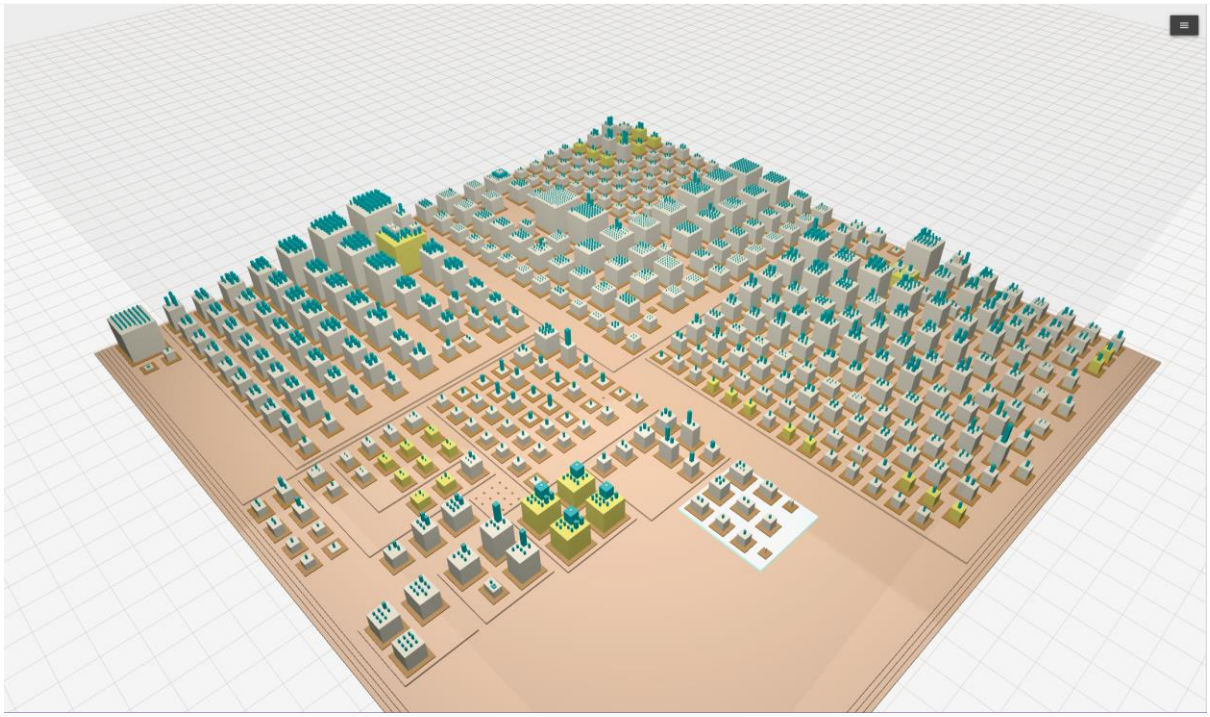


Figure 7.8: A view of iTrust codebase showing similar structures highlighted in yellow color.

The above figure illustrates how the visualisation exposes a number of identical-looking buildings. Upon inspection of those buildings, it is revealed that they indeed bear similar code implementations and structures. This could offer insights on potential refactoring and better redesign of the codebase. This view was demonstrated and discussed with experts during the evaluation sessions.

Accessible Traceability

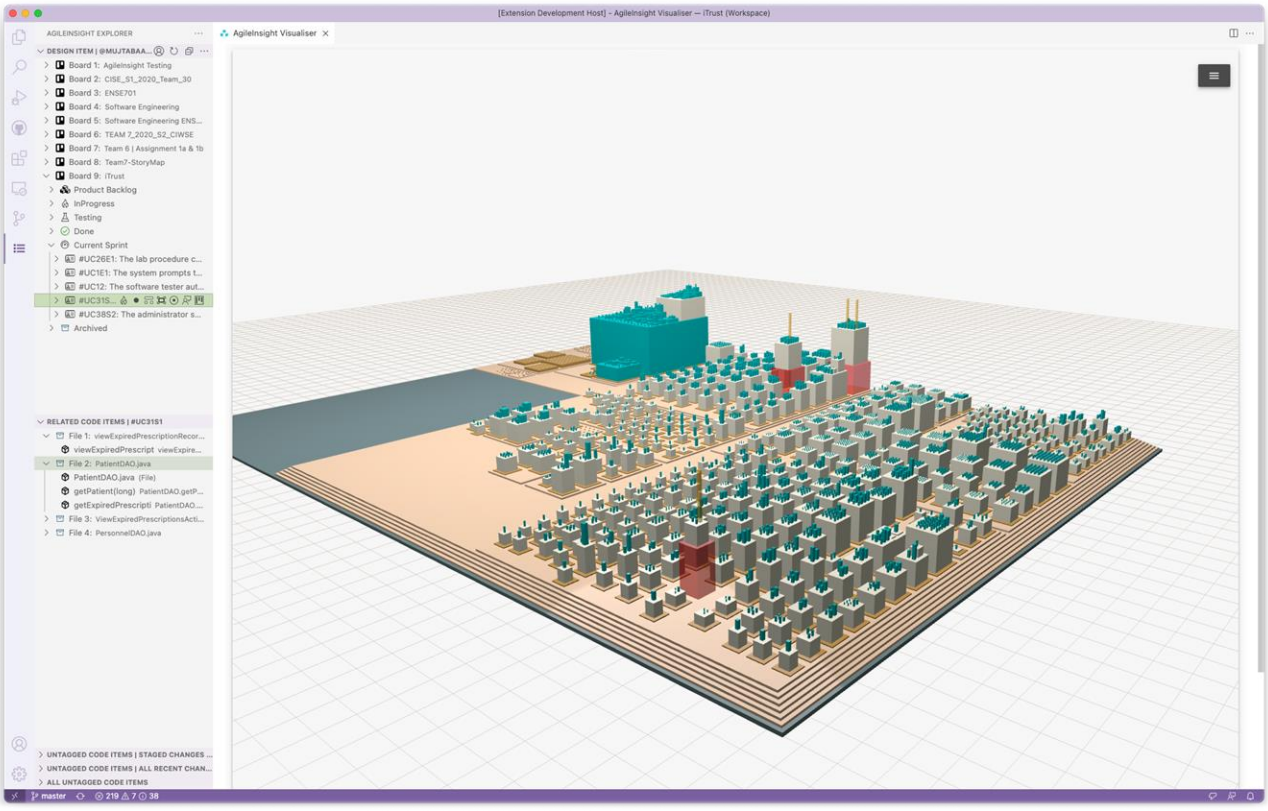


Figure 7.9: A view of visualised iTrust codebase showing artefacts being highlighted and animated reflecting implementation locality of a selected design item.

The above figure illustrates partial design items of iTrust displayed in the Dashboard view on the left side (pulled from its Trello board). A design item (UC31S1) has been selected to reveal its related code items in the lower viewlet. The scene distinctively reveals those code items across the scene, offering a sense of feature locality. In real life, the coloured items are animated with graceful 'popup' movements to further aid with spotting the items. The red colours indicate file-level tracelinks whereas the yellow colours indicate inner-level tracelinks (i.e., class-level, method-level, and so on).

A Complex Feature—UC15S3

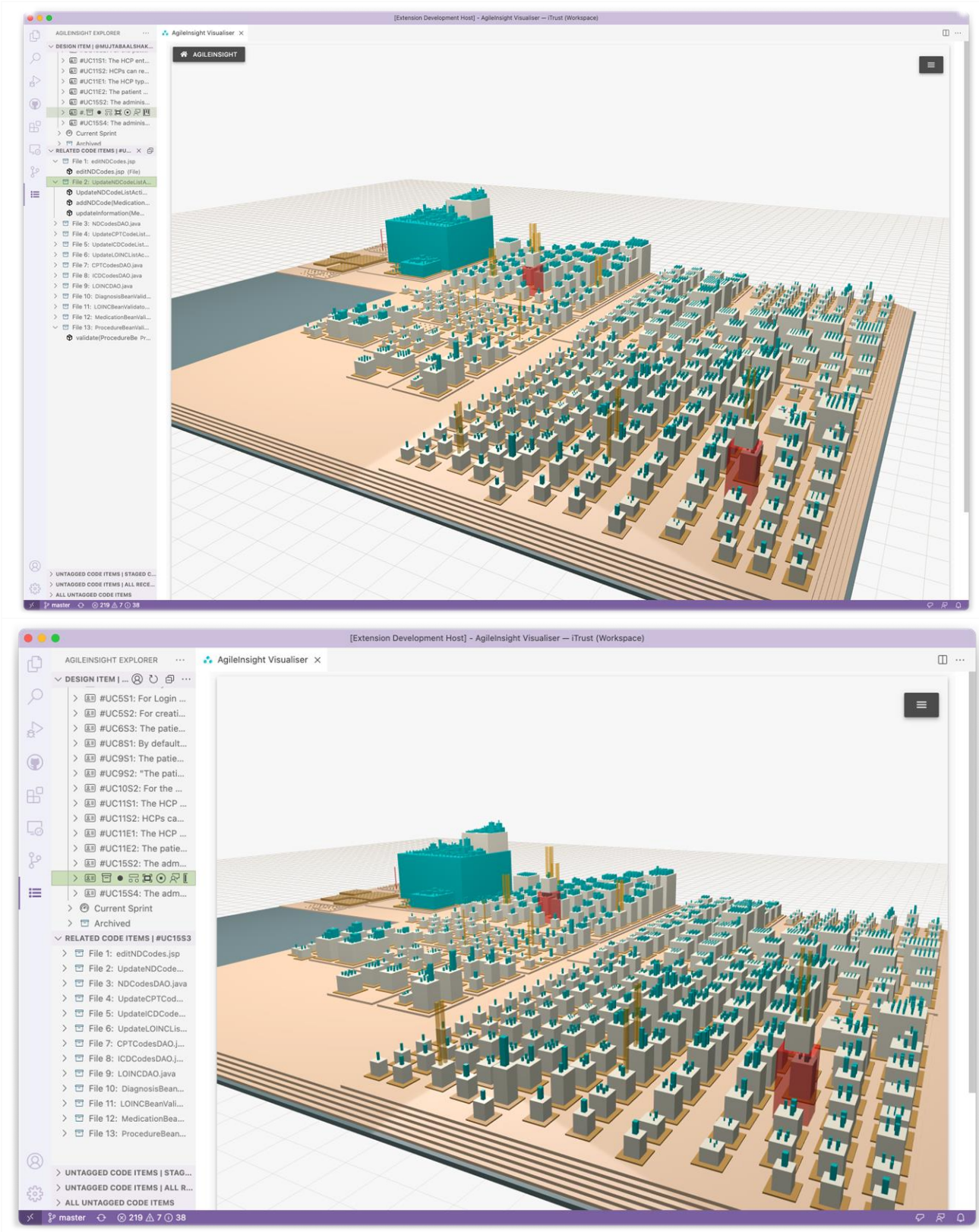


Figure 7.10: Two views of iTrust as visualised by AgileInsight demonstrating contextual traceability in action for a complex feature.

The above two figures offer a similar traceability mechanism, but this time a more complex feature is showcased. UC15S3 is related to an exceptionally large number of code items in a total of 13 files as

the lower tree list viewlet displays. The scene reveals the distribution of these related code items across multiple modules of the codebase. At first look, it can be readily realised that only three file-level relations are involved—the red-coloured objects, one of which is a Jsp file appearing at the back. The rest are all method relations (yellow coloured objects).

Bi-Directional Traceability

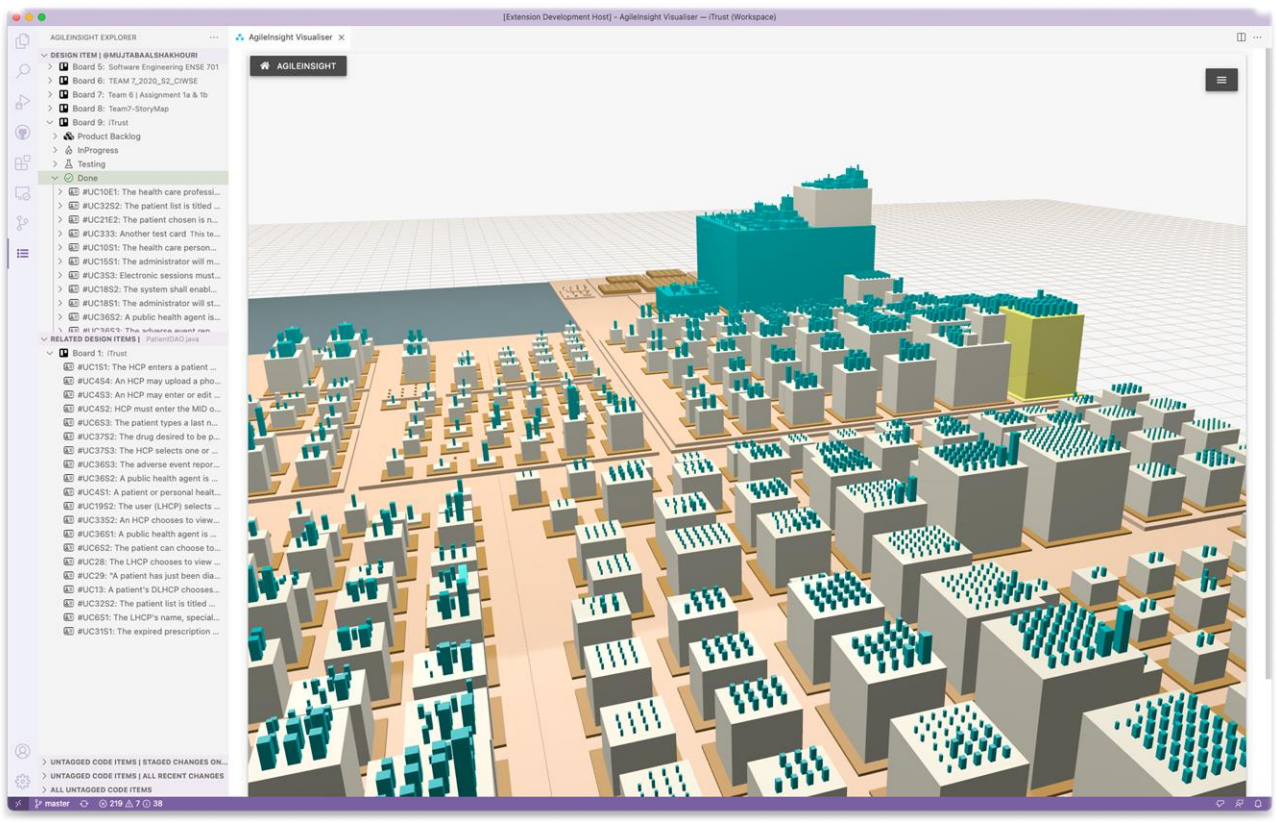


Figure 7.11: Bi-directional traceability in action showing design items revealed in side viewlet (lower left) after command invoked for a building of interest (yellow coloured).

The above figure serves to showcase one of the bi-directional functionalities offered by AgileInsight. A user has selected a particular building within the scene (yellow coloured) and chose to reveal the design items related to it. On the left side, the Related Design Items viewlet displays an exceptionally large number of related design items (20 in total). Given that this is a real-world dataset, this view could potentially alert quality engineers to address an emerging god class.

An interactive Working Ground

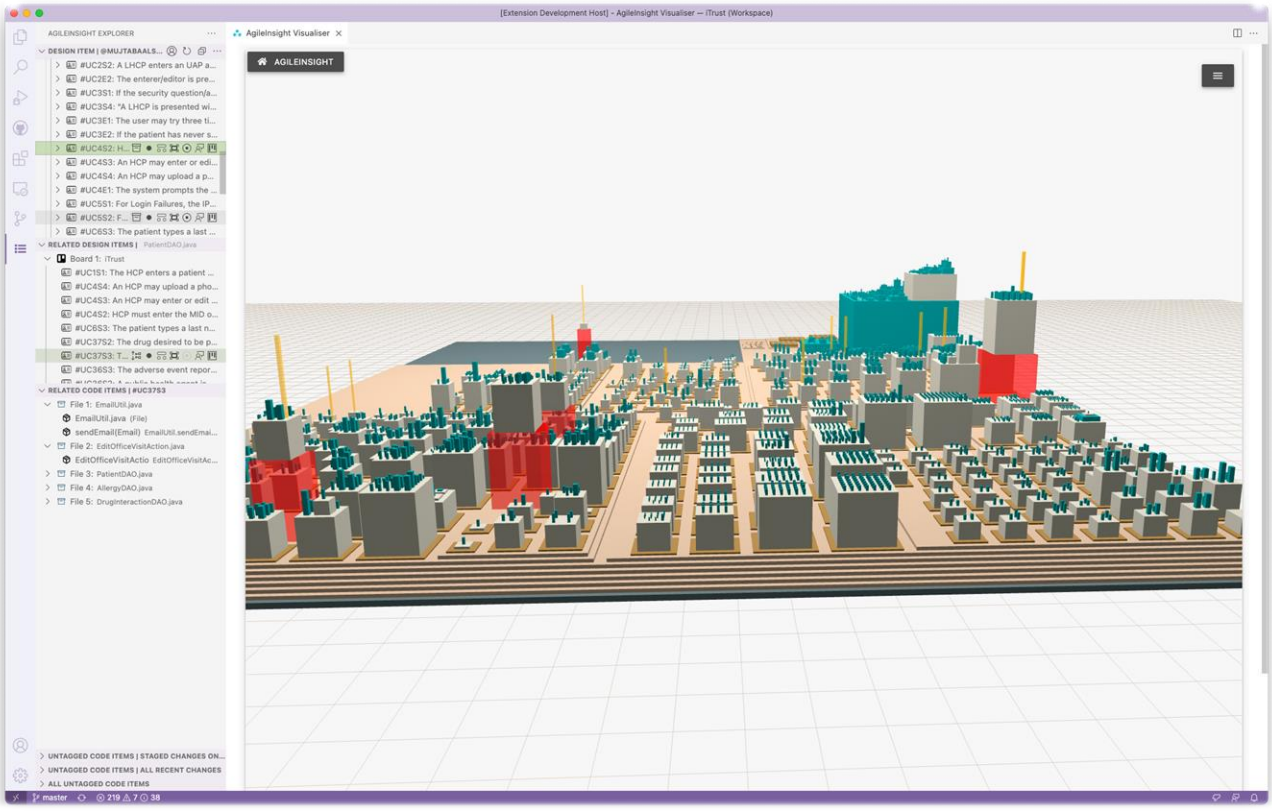


Figure 7.12: AgileInsight potential to offer a working ground (or sandpit) for developers exploring and learning about their codebase.

In the above figure, the user employs AgileInsight to study the iTrust codebase. They have selected two features to reveal their location across the visualised codebase. They are able to access all the relevant design information right within their IDE. They engage in a series of traceability traversals between the design items and their related code items. They can access the traceability information as a simple tree list, as well as a visual representation mapped to the codebase. They can instantly navigate to the source code of any item to inspect it.

7.4 Multi-Language Showcase

This section presents a visual showcase of 15 real-world open-source systems, implemented using different programming languages. The goal of the showcase is twofold. The first goal is to demonstrate the potential of the visualisation in exposing the structure and complexity of source code, allowing users to possibly gain insights about their code. This potential could then be extended when the source code is further synchronised with the original requirements—something that the research tool seeks to facilitate through its introduced tagging mechanism. The second goal is to test and demonstrate the agnostic nature of AgileInsight, and its ability to parse and visualise source code irrespective of the underlying programming language—thanks to the LSP feature of Vscode. As detailed in Chapter 6, AgileInsight was designed to work in an agnostic manner. During development, the tool was tested with Java, CSharp, JavaScript, Python, and TypeScript. In writing this section, new languages were attempted 'out of the box', and the visualisation worked gracefully with almost no modification being needed[111]. Nonetheless, due to current inconsistencies (and some imperfections) in the LSP implementations of some languages, some servers take longer than others in provisioning the document symbols[112]. More importantly, while AgileInsight was able to work with all these languages, its current implementation certainly requires improvements and refinements before it could be put to public use. As the benchmarks provided at the end of this section reveal, its parsing capability is clearly impacted by the individual LSPs that Vscode needs to communicate with, as well as their number—parsing a workspace containing a single language appears to be considerably faster than a multi-language workspace.

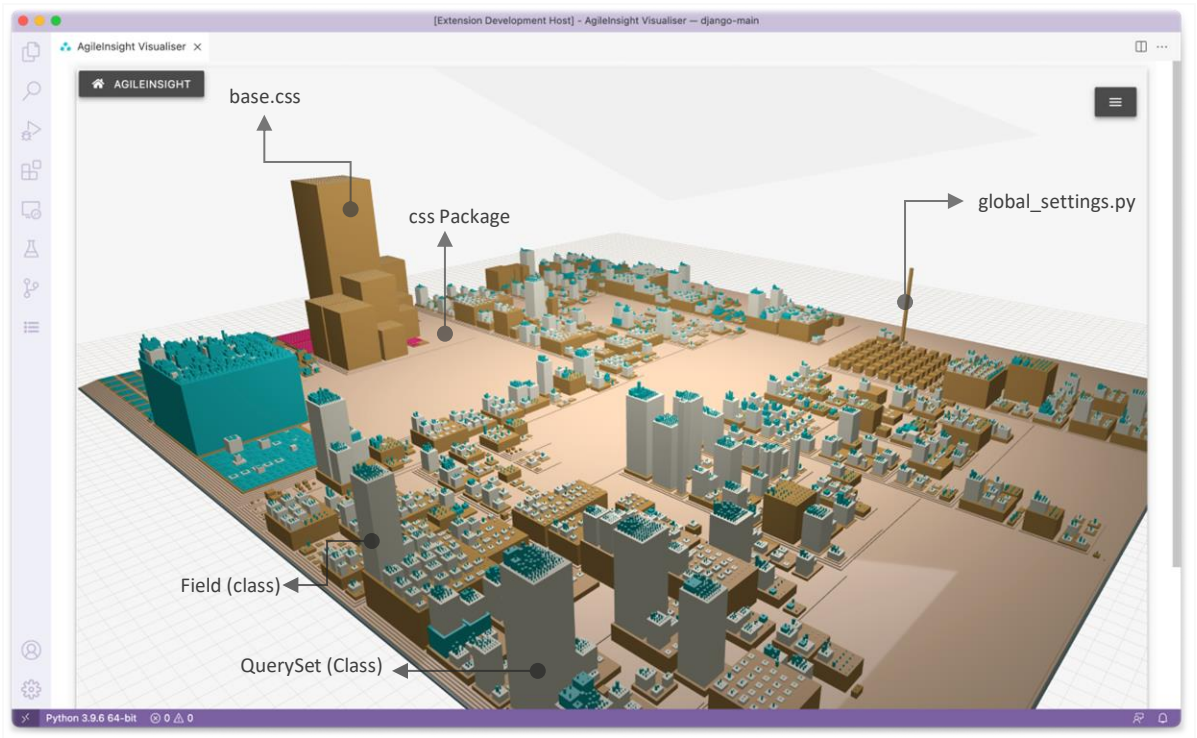
The multi-language showcases that follow focus specifically on the visualisation aspect, with very brief commentary provided. The visual scenes presented reveal the visualisation's ability to expose the code structure in a series of languages that were attempted. As discussed earlier, the software visualisation literature appears to have particularly focused on Java systems, and to the best of our knowledge, AgileInsight appears to be the first language-agnostic visualisation tool. Many of the systems presented below belong to languages that have rarely—if any—been visualised or presented before in literature, such as Python, JavaScript, Go, and Swift. A table offering benchmarks is presented at the end of this section.

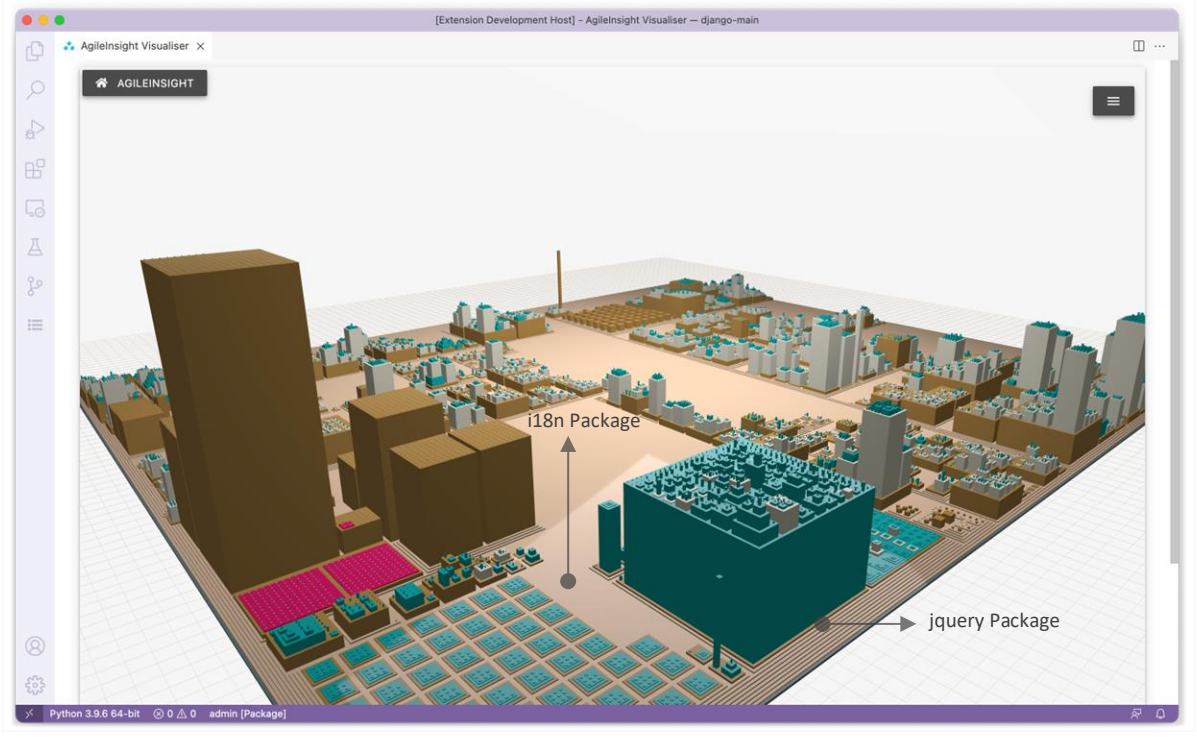
[111] The only modification performed was adding a conditional statement to treat 'struct' objects as primary containers.

[112] It was found that time calibration of the request-and-wait part of the parsing operation could improve the parsing speed for some languages (see section 6.5.1).

1- Django—Python

The following figures present the visualisation of the Django open-source library[113], represented by its public Github main branch as of 10th April, 2022. It consists of a total of 889 files, spanning the following languages: Python, JavaScript, HTML, and CSS. The first two figures use capped linear mapping for primary containers, and normalised mapping for secondary containers. An all-linear mapping view appears next in the two figures presented afterwards. The visualisation offers a range of exposed code structures as denoted by the annotations.





[113] https://github.com/django/django

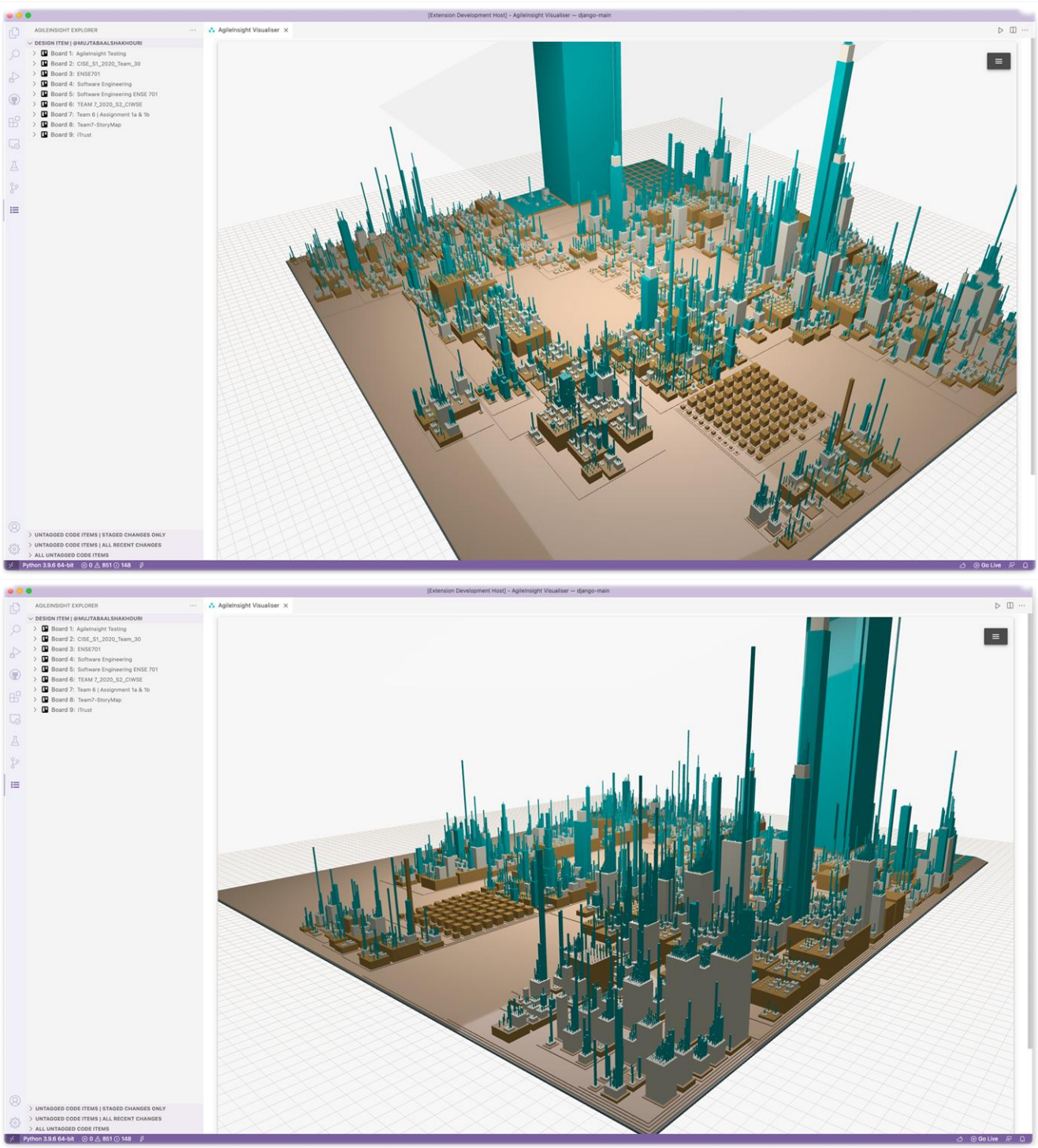


Figure 7.13: Django's library visualised by AgileInsight as of 10th April, 2022, using capped linear mapping for primary containers, and normalised mapping for secondary containers (top two figures). Lower two figures show the library visualised using linear mapping.

2- Keras—Python

The following figures shows Python's deep learning API, Keras, represented by its GitHub master branch[114] as of 10th April, 2022. It consists of 661 Python files. White buildings are classes, bluish-green components are methods or functions (when not inside a class), and brown elements are files.

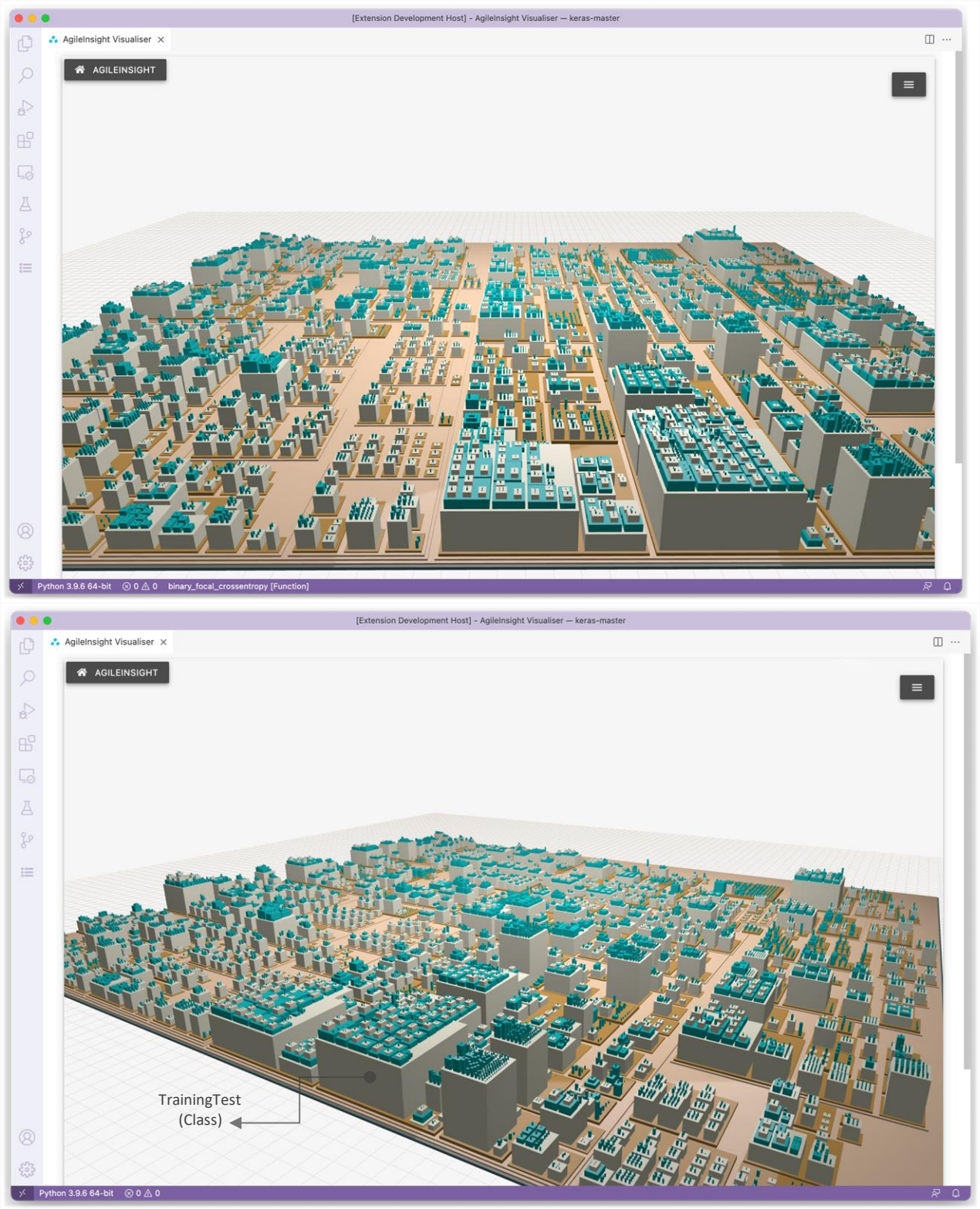


Figure 7.14: Keras' API as of 10th April, 2022, visualised using default normalised dynamic mapping.

[114] https://github.com/keras-team/keras

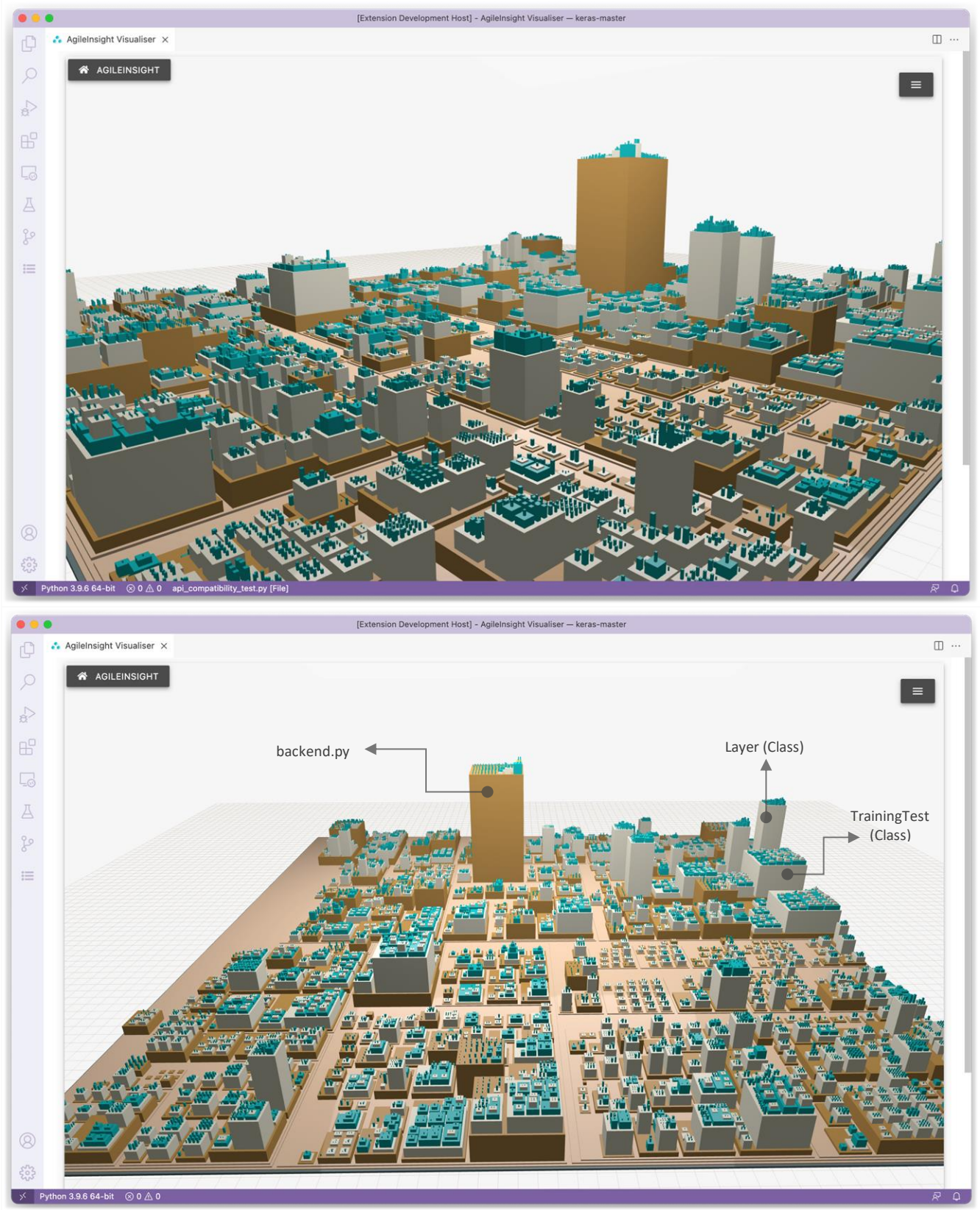


Figure 7.15: Keras' API as of 10th April, 2022, visualised using capped linear mapping for primary containers, and normalised mapping for secondary containers. Here, files and class height is a closer reflection of the number of children contained by the objects (direct children only). See section 6.9.2 for details on mapping functions used.

3- Facebook SDK for iOS—Swift

Below is a visualisation of Facebook's API for iOS represented by its main Github branch[115] as of 10th April, 2022. It consists of a total of 779 files, spanning Swift, Objective-C, and JavaScript source code. The pink buildings represent namespaces, while the burgundy building denotes structs.

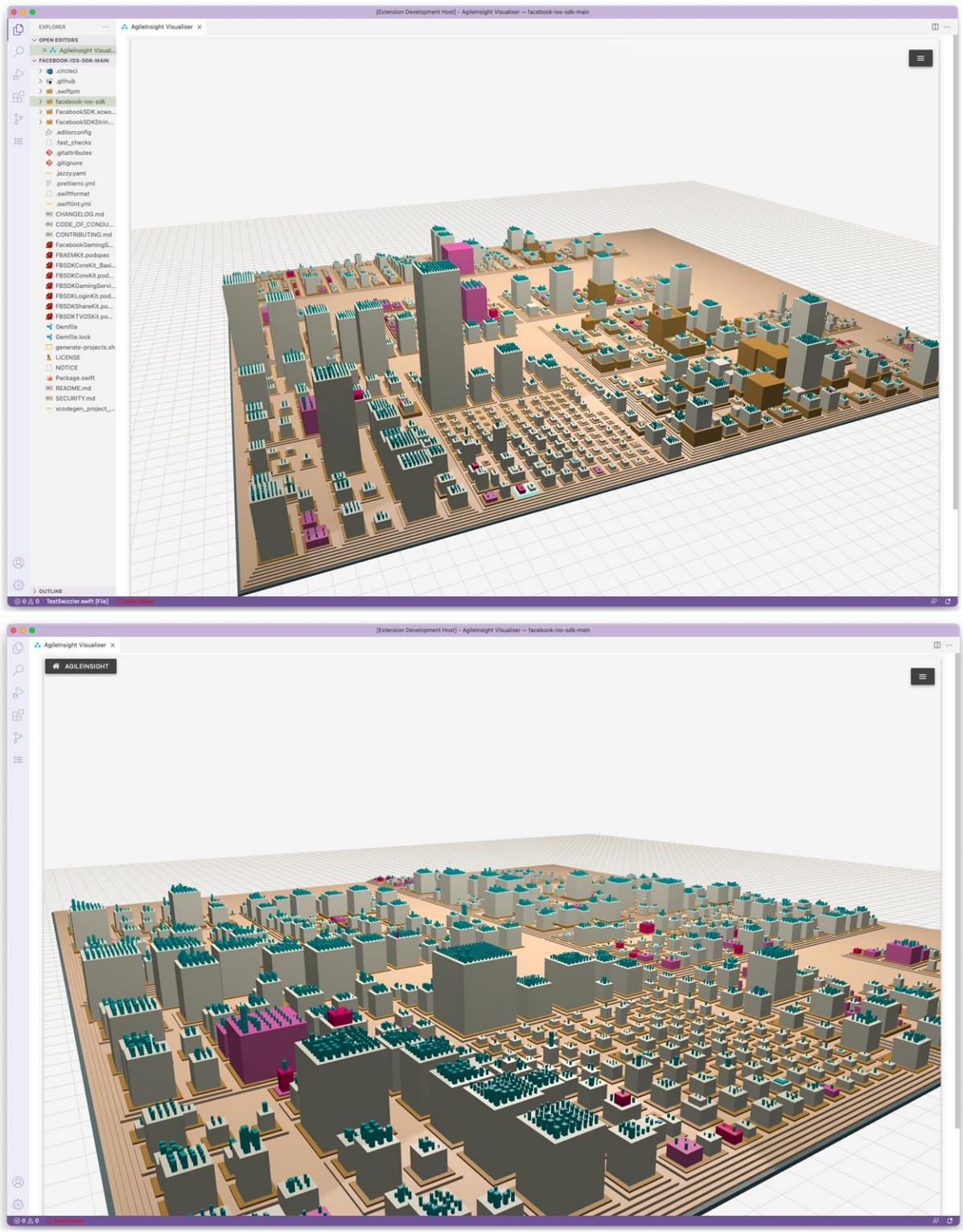


Figure 7.16: A visualisation of Facebook's iOS API as of 11th April, 2022. Top figure uses capped linear for primary containers mapping, while the lower shows the visualisation in default normalised mapping.

[115] https://github.com/facebook/facebook-ios-sdk

4- ReactiveX—Swift

The following is a visualisation of ReactiveX API for Swift, represented by its main GitHub branch[116], as of 10th April, 2022. It consists of 996 files of Swift and Objective-C source code. The library's structure appears to feature relatively low complexity files—no tower buildings even in capped linear mode.

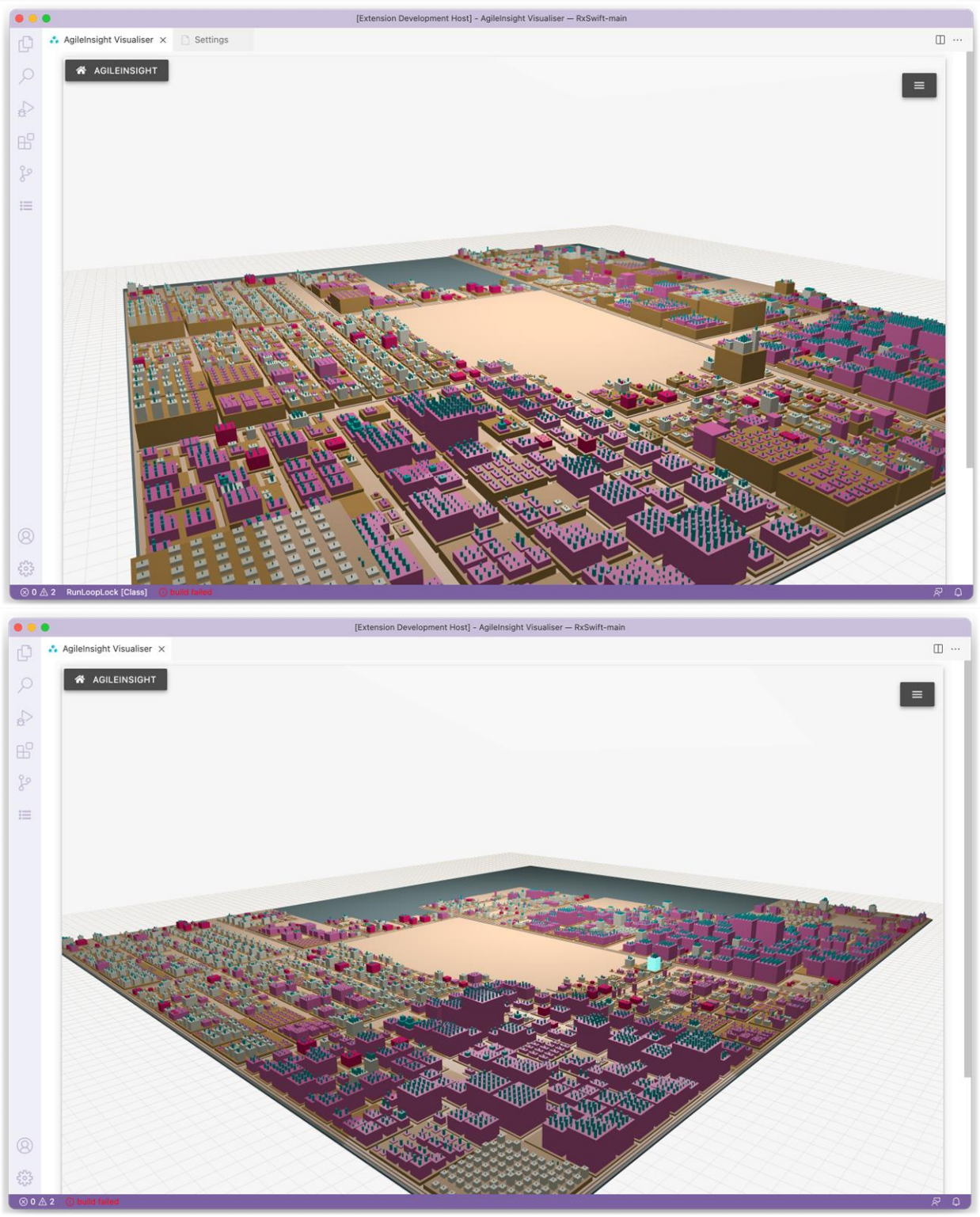


Figure 7.17: A visualisation of ReactiveX Swift API as of 10th April, 2022. Top view uses capped linear mapping for primary containers while the lower uses the default normalised mapping.

[116] https://github.com/ReactiveX/RxSwift

5- CodeEdit—Swift

The figures below show the visualisation of CodeEdit's Swift source code. It features a small codebase of 164 Swift source files, represented by its GitHub main branch[117] as of 10th April, 2022. Due to the simple structure of the codebase, the normalised mapping in this case (lower view) appears to produce little difference compared to the capped linear mapping (top view)—an expected outcome of the function.

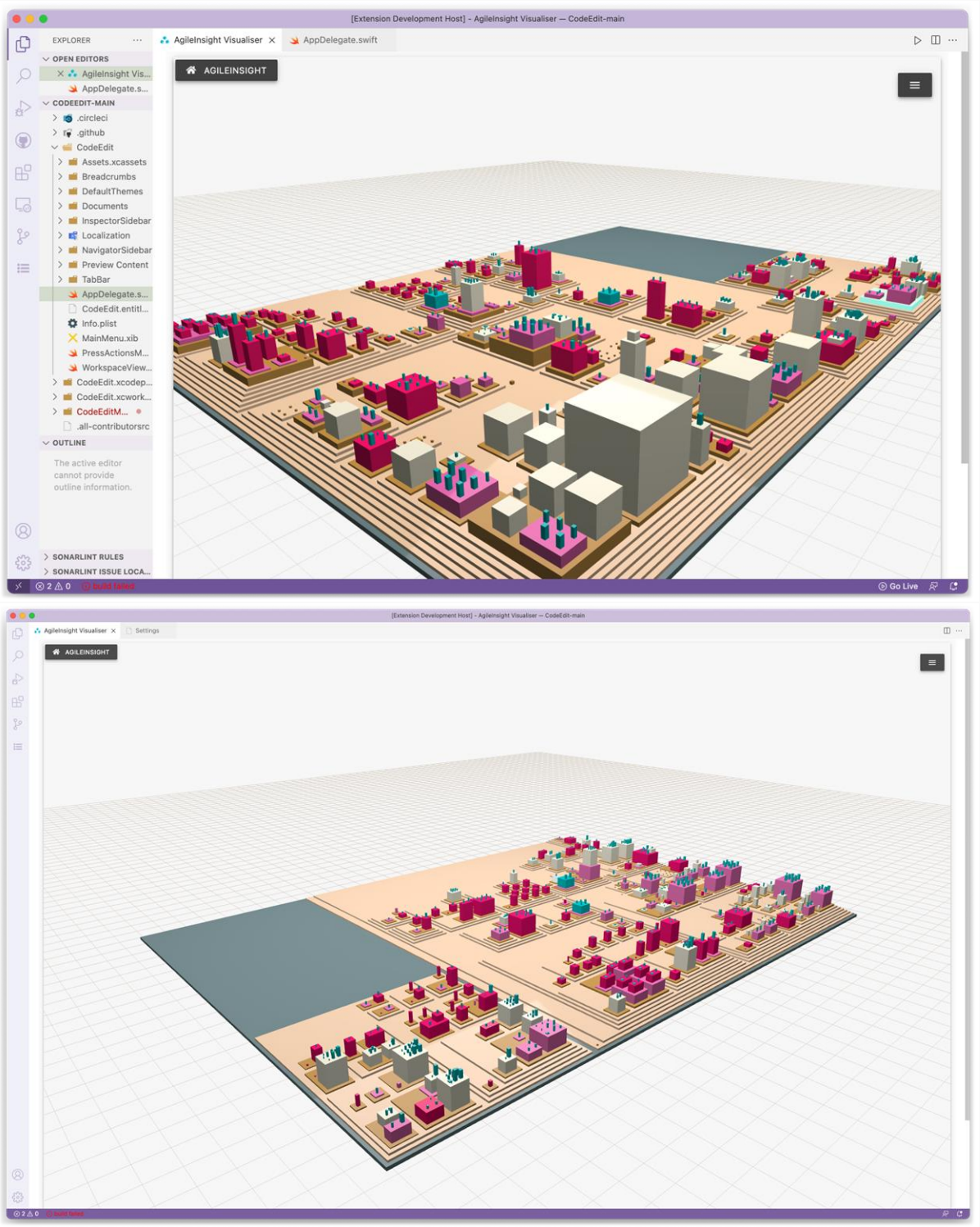


Figure 7.18: A visualisation of CodeEdit's source code as of 10th April, 2022 showing capped linear mapping for primary containers in top view, and default normalised mapping in lower figure.

[117] https://github.com/CodeEditApp/CodeEdit

6- ASP.NET Boilerplate—Csharp

Below is a visualisation of the ASP.NET Boilerplate API represented by its Github main branch[118] as of 10th April, 2022. It consists of a total of 1401 files spread across Csharp, JavaScript, Typescript, and CSS source code. Except for its Resources package (highlighted in the lower view), the codebase appears to harbour very few files of exceptional complexity, and rather appears to be dominated by a large number of small and encapsulated code elements. The peculiar module in the middle represents the Resources package, displaying a few especially complex functions.

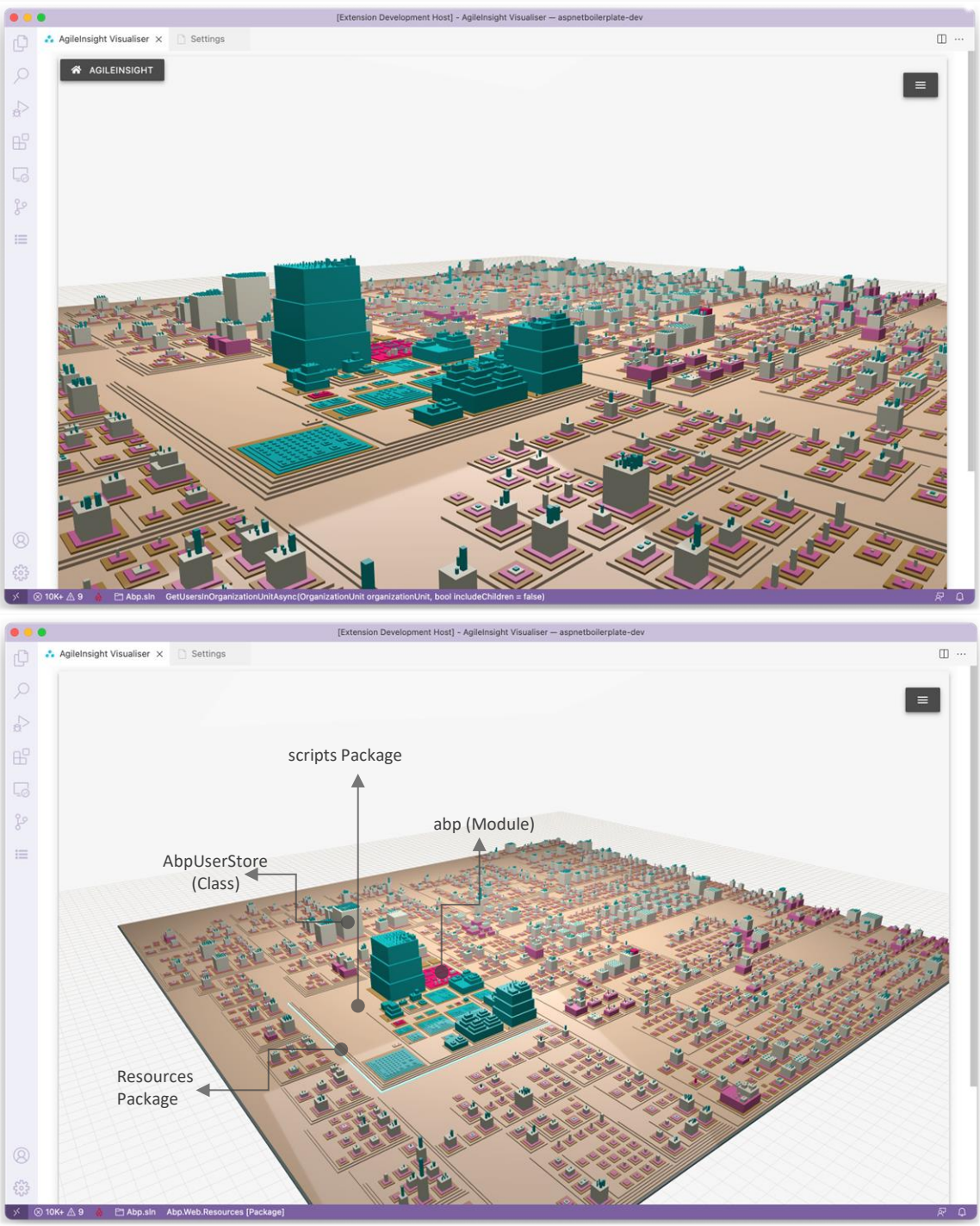
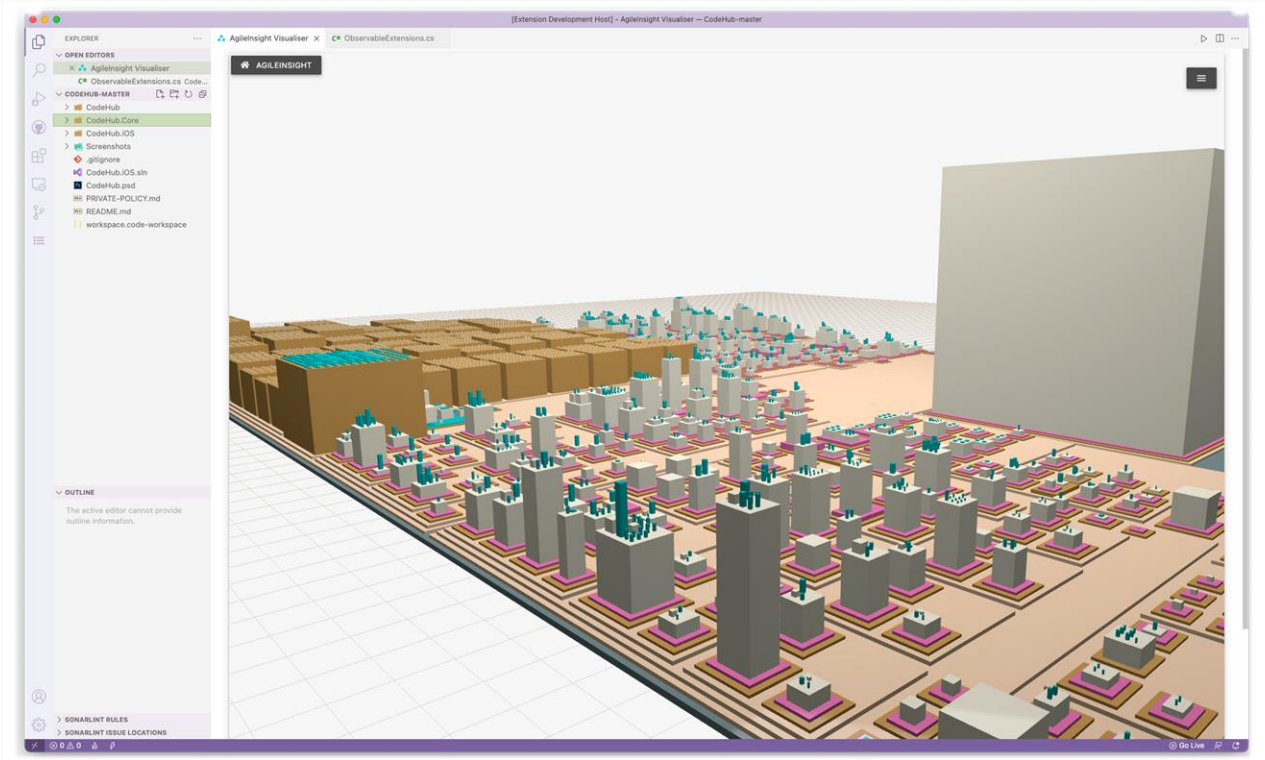


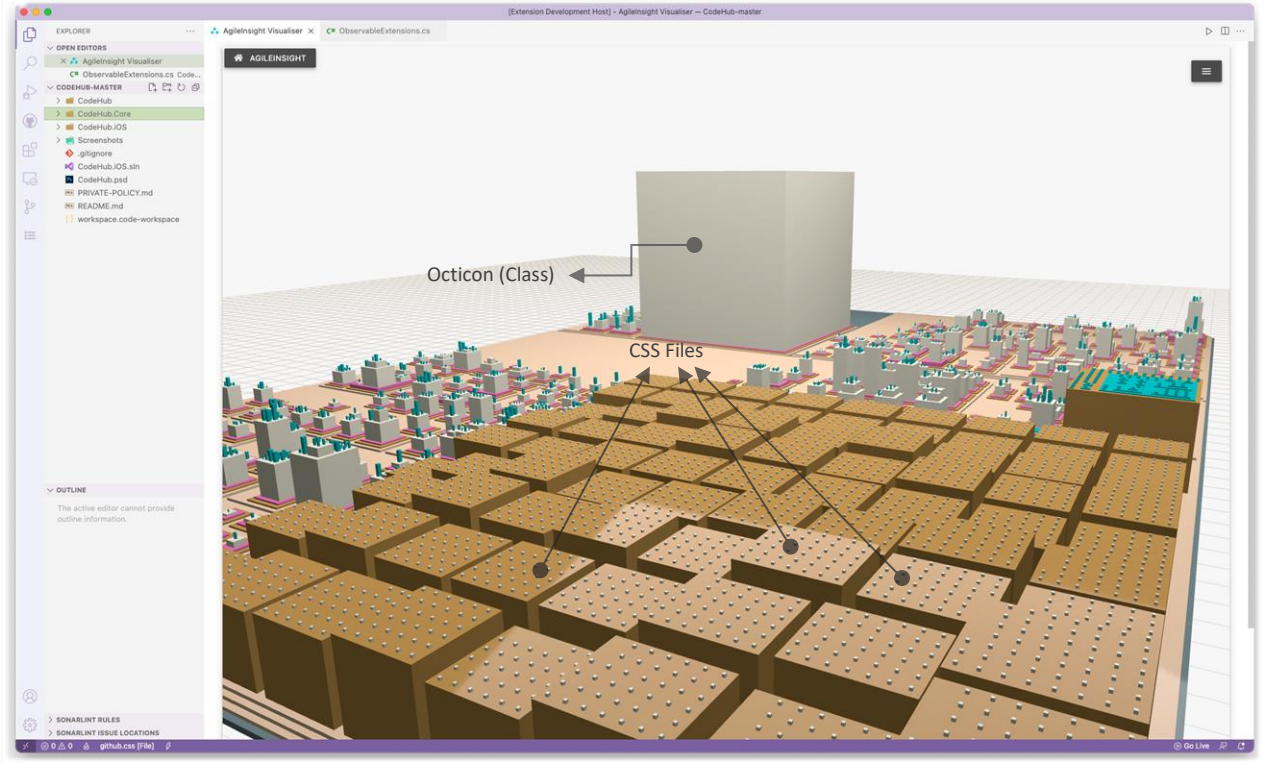
Figure 7.19: ASP.NET Boilerplate API visualised by AgileInsight as of 10th April, 2022. Linear capped mapping is used in top view and default normalised mapping in lower view.

[118] https://github.com/aspnetboilerplate/aspnetboilerplate

7- CodeHub—Csharp

The figures below show a visualisation of the CodeHub application for iOS devices as represented by its Github master branch[119] as of 10th April, 2022. It consists of a total of 377 files implemented in Csharp, JavaScript, and CSS source code. The scene appears to be dominated by an exceptionally complex god object (its `Octicon` class), and a good number of CSS files (the brown cubes) that are also non-trivial.





[119] https://github.com/CodeHubApp/CodeHub

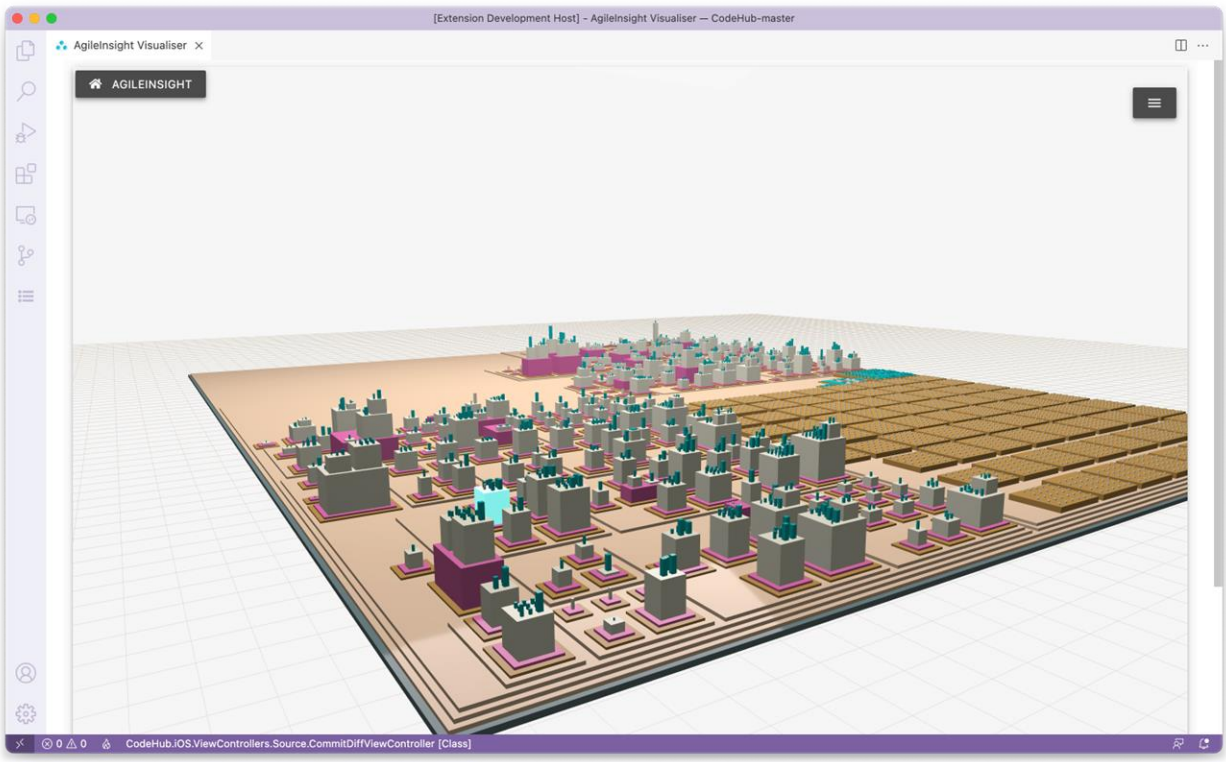
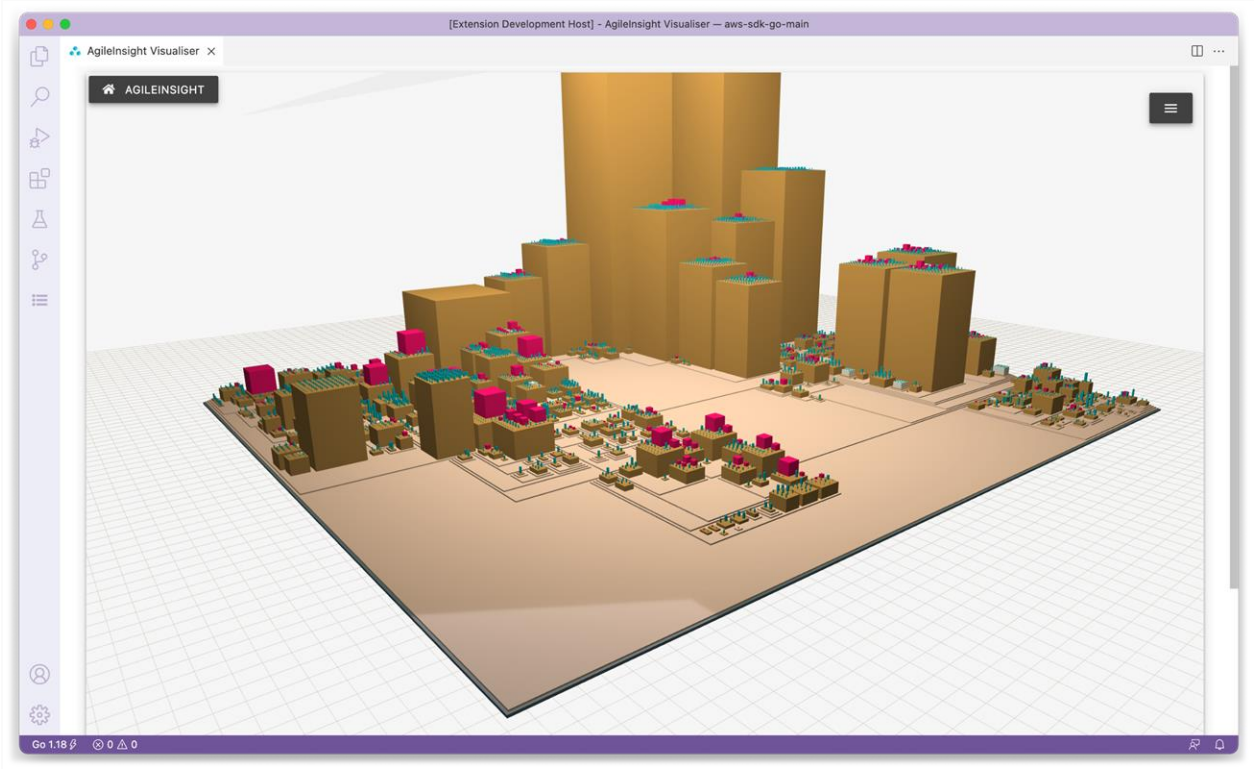


Figure 7.20: CodeHub master branch visualised by AgileInsight as of 10th April, 2022. Linear capped mapping is used in top two views, while lower one shows a default normalised mapping.

8- AWS SDK for Go (Without Service Module)—Go

Below is a partial visualisation of Amazon's AWS API for the Go language—the 'Services' module is not included (see the benchmarks section for details). It represents the API's Github main branch[120] as of 11th April, 2022, and consists of 409 Go source files. The codebase appears initially to be dominated by a good number of complex files. However, that complexity is only a reflection of the large number of direct descendants composing those files (functions and structs), which appears to be different from most object-oriented languages where a file typically contains a single or limited number of classes. Nonetheless, those functions are clearly far from complex as denoted by their relatively small sizes, which the normalised view (third figure) makes clear. Another feature of the code base is the limited number of classes (white buildings).



[120] https://github.com/aws/aws-sdk-go

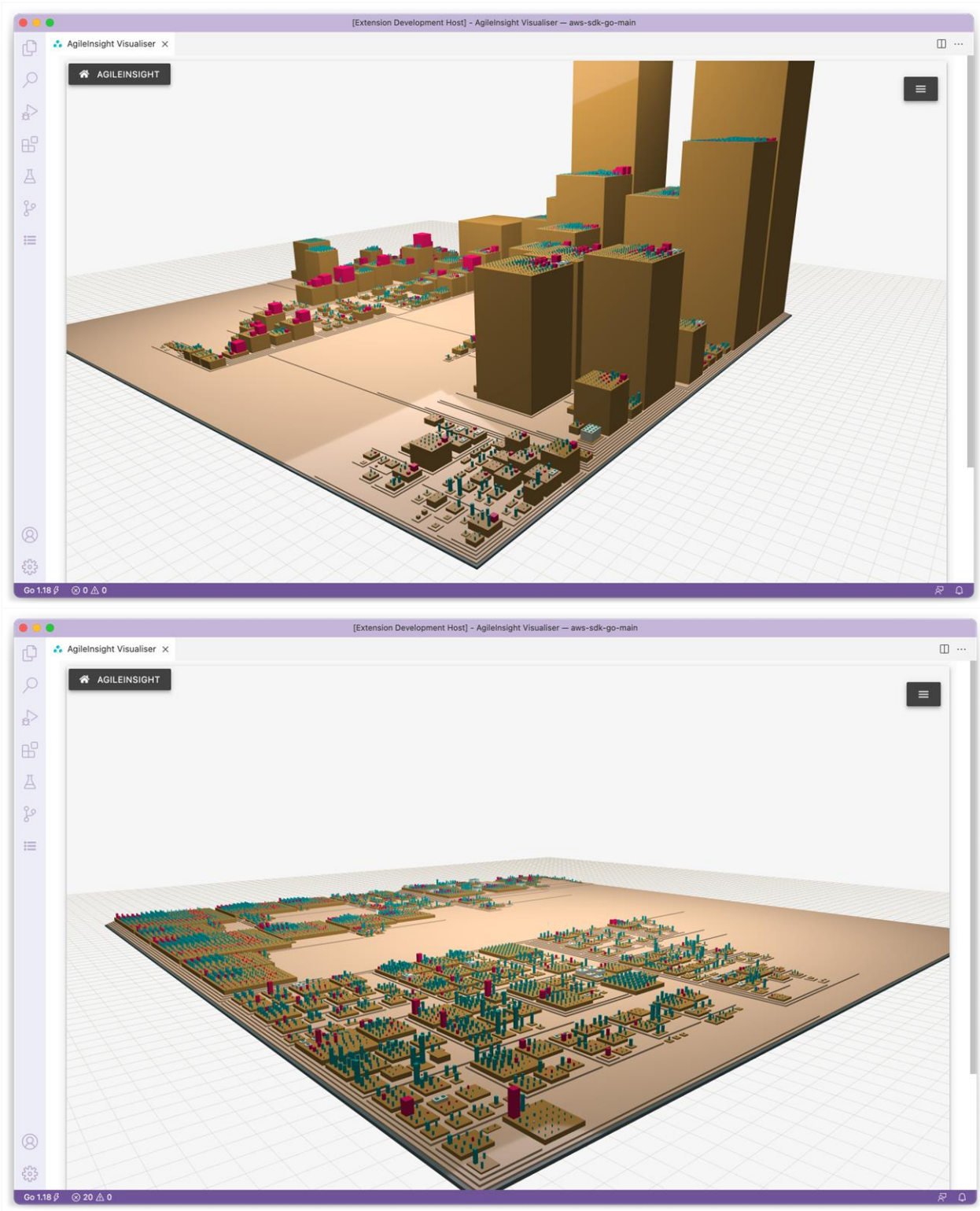
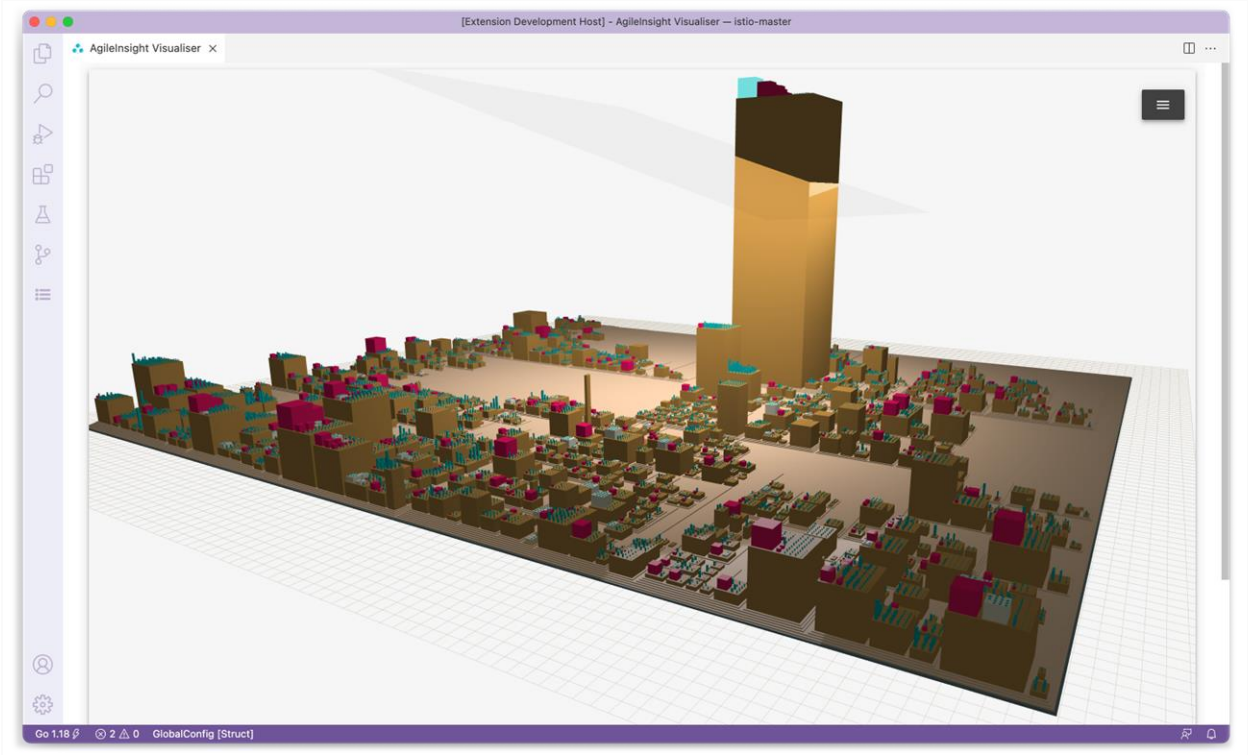
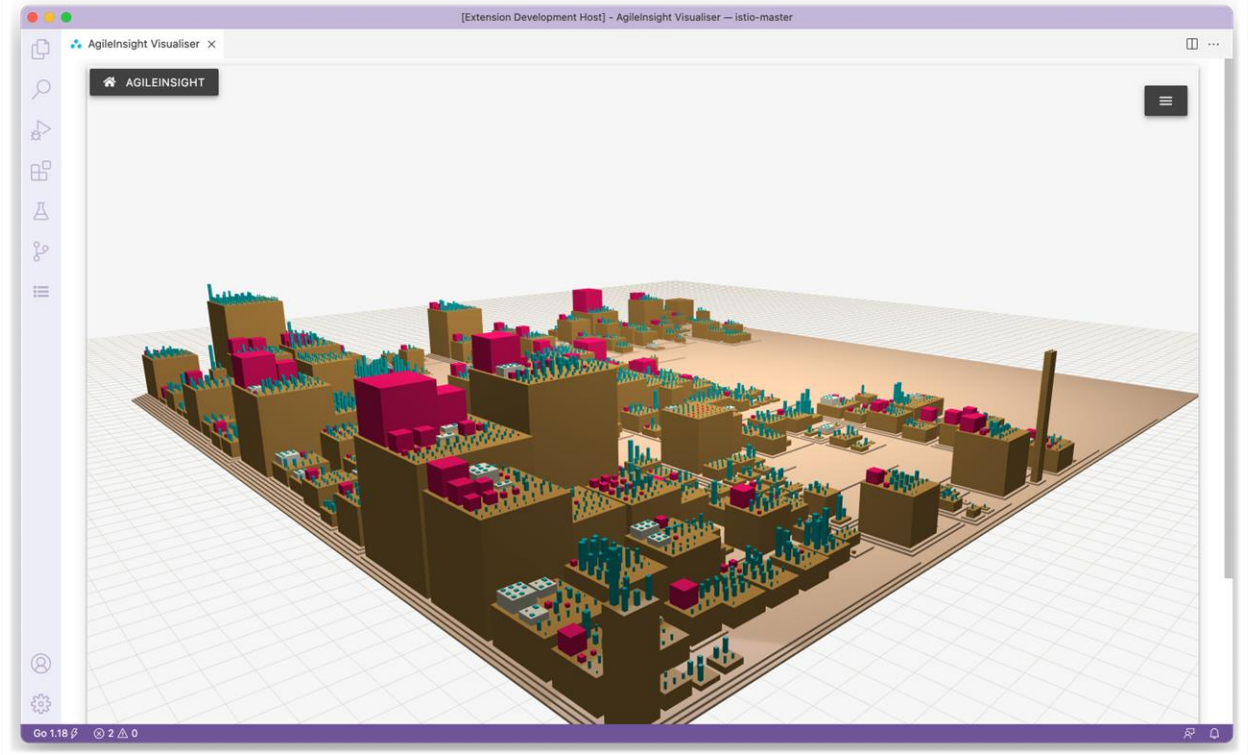


Figure 7.21: Partial visualisation of AWS's API for the Go language (excludes the Service Module) as of 11th April, 2022. Capped linear mapping is used in first two views, while default normalised mapping is used in the third.

9- ISTIO—Go

Below is a visualisation of the microservices platform, ISTIO, represented by its master Github branch[121] as of 11th April, 2022. It consists of a total of 1391 files of Go language source code. A single file (`values_types.pb.go`) appears to contain an exceptionally large number of functions, and a good number of non-simple structs. Upon inspection, it later appeared to be a central management of configuration-related values and procedures.





[121] https://github.com/istio/istio

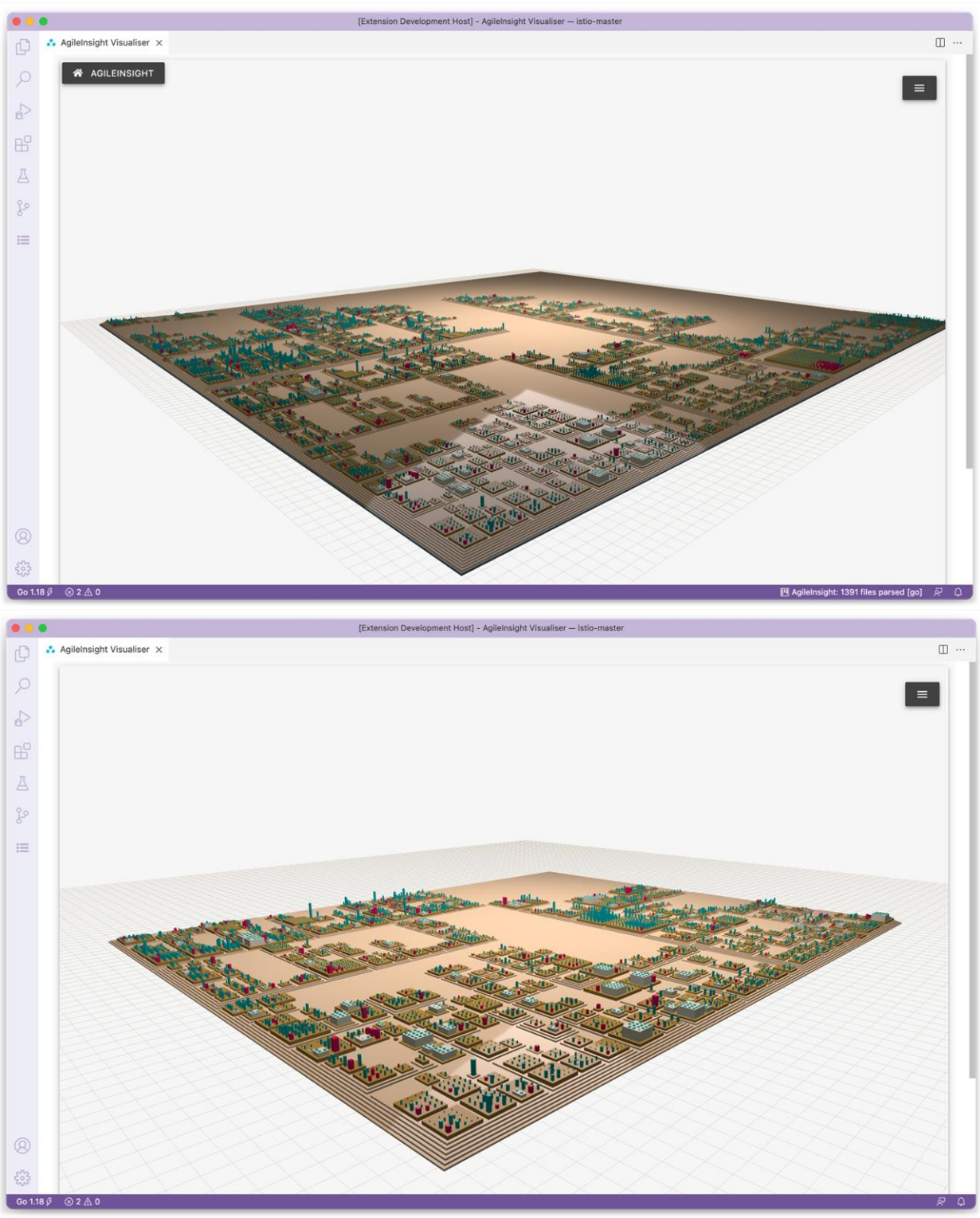
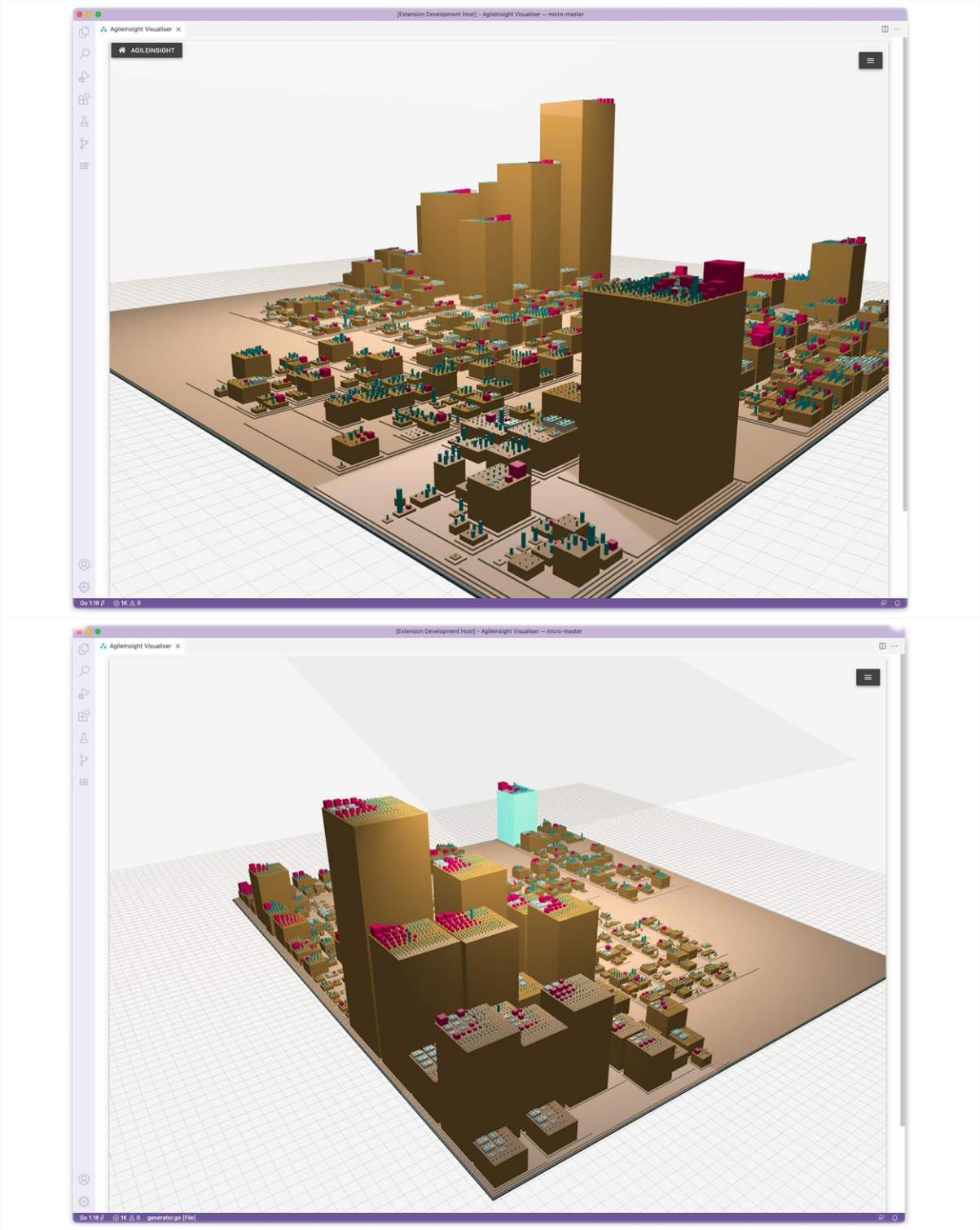


Figure 7.22: A visualisation of ISTIO's master branch as of 11th April, 2022. The first two views use capped linear mapping and show an overall, then a focused view (showing the `test` package), respectively. The lower two views use default normalised mapping, and similarly show an overall then a focused view (this time showing the `pkg` package), respectively.

10- Micro—Go

Below is a visualisation of Go's Micro API represented by its main Github branch[122] as of 11th April, 2022. It consists of a total of 586 Go source code files. The various visualisations of the Go systems presented so far appear to reveal its unique programming style (compared to the other languages)—few structs in each file accompanied by a large number of functions. Classes (with their inner methods) are clearly less used in Go.



[122] https://github.com/micro/micro

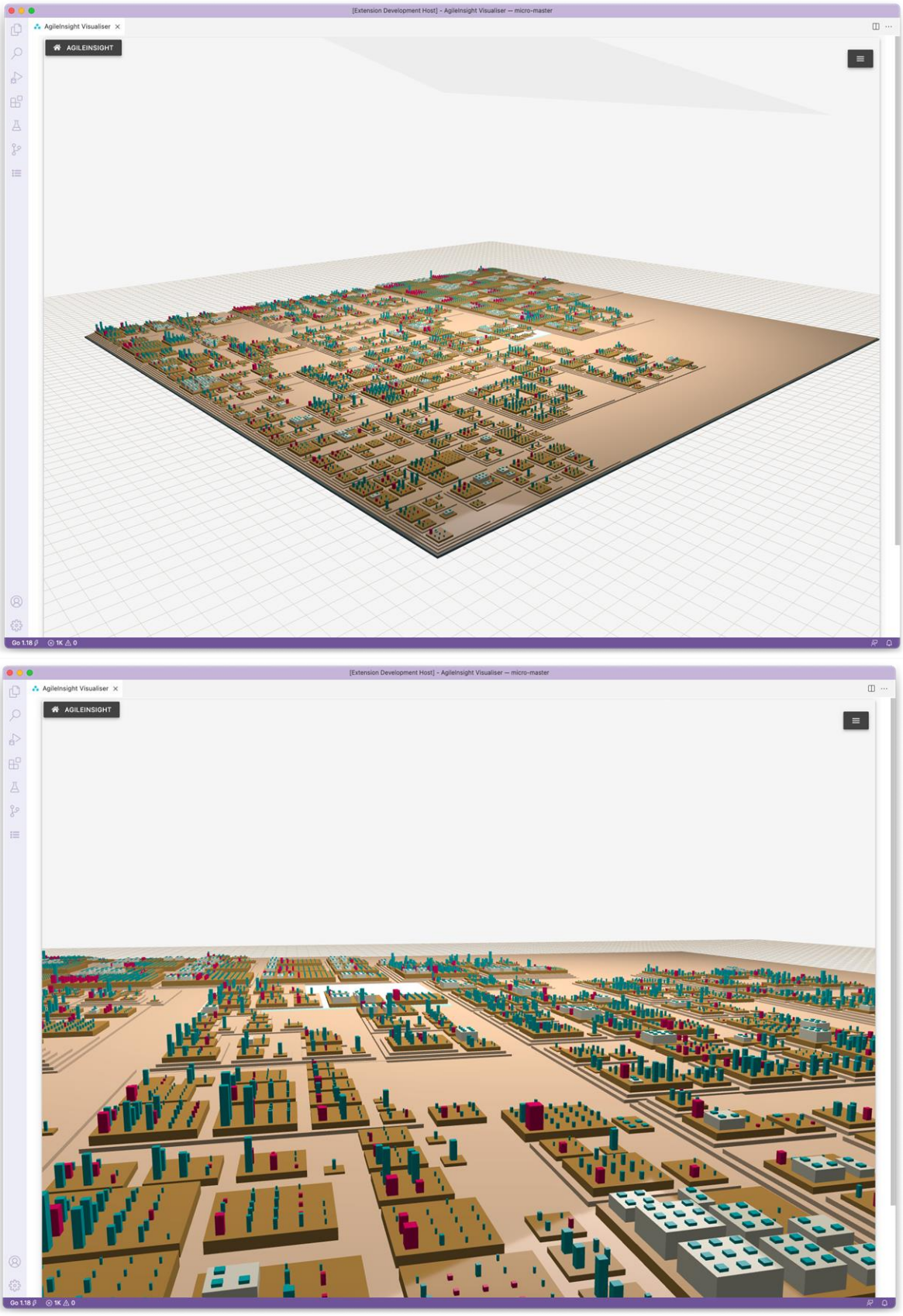
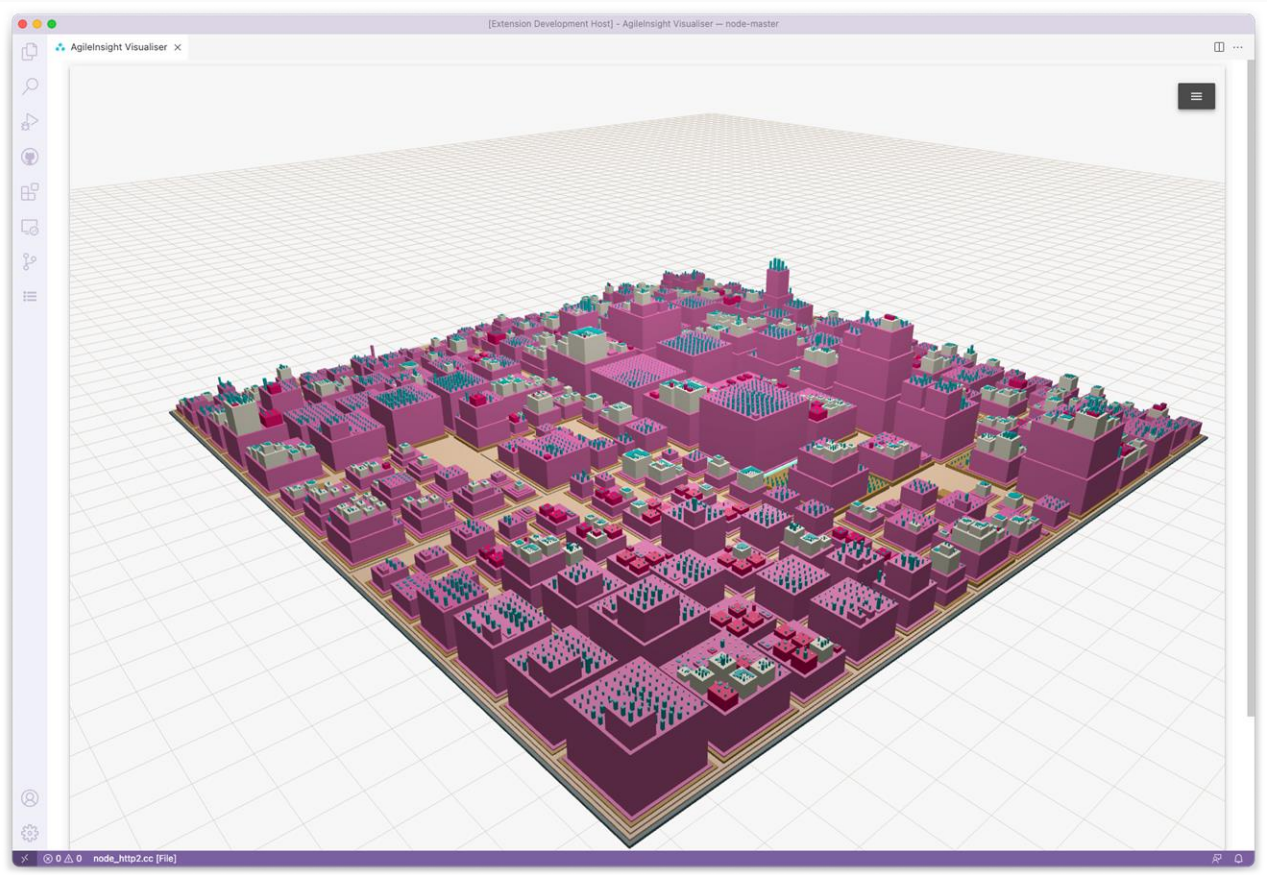


Figure 7.23: A visualisation of Go's Micro API as of 11th April, 2022. Capped linear mapping is used in first two views, while default normalised mapping is used in the latter two.

11- Node.js (C++ Backend)—C++

The figures below show a visualisation of the Node.js backend engine, which is written in C++. The visualisation depicts the src folder only of the Node.js Github master branch[123] as of 10th April, 2022. It consists of a total of 291 header (`.h`) and implementation (`.cc`) files. Namespaces (pink-coloured) dominate the C++ codebase, but it also features a large number of classes and structs (burgundy-coloured).



[123] https://github.com/nodejs/node/src

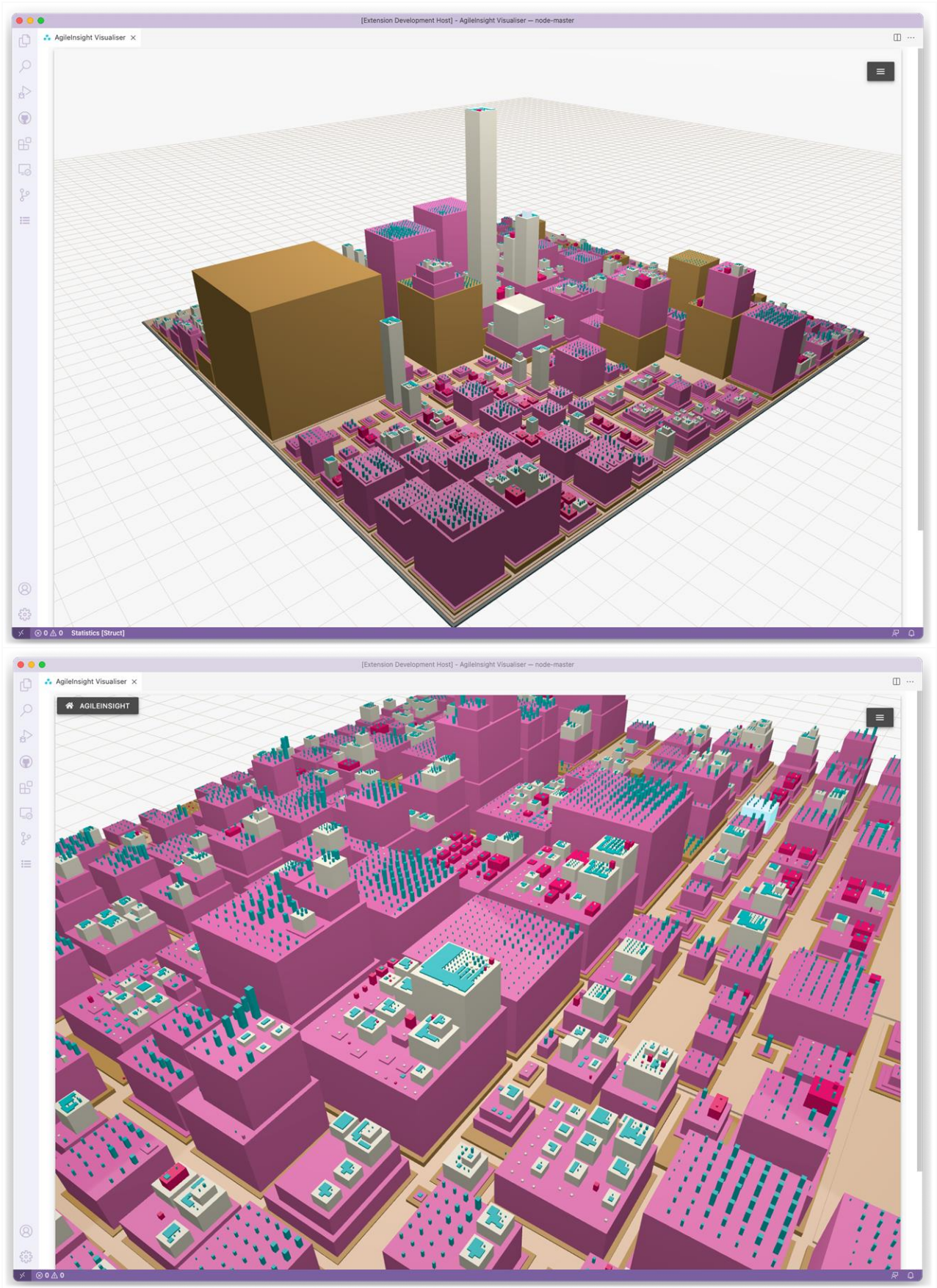
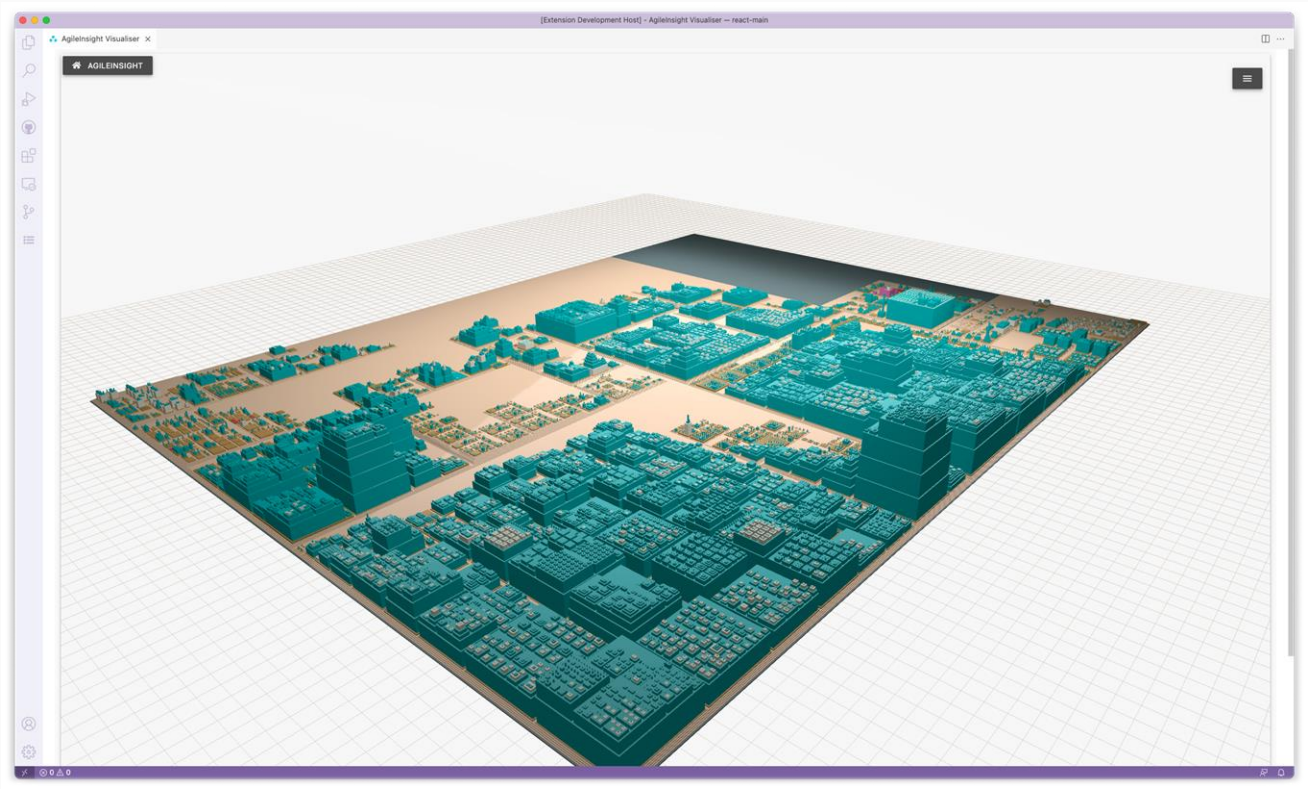


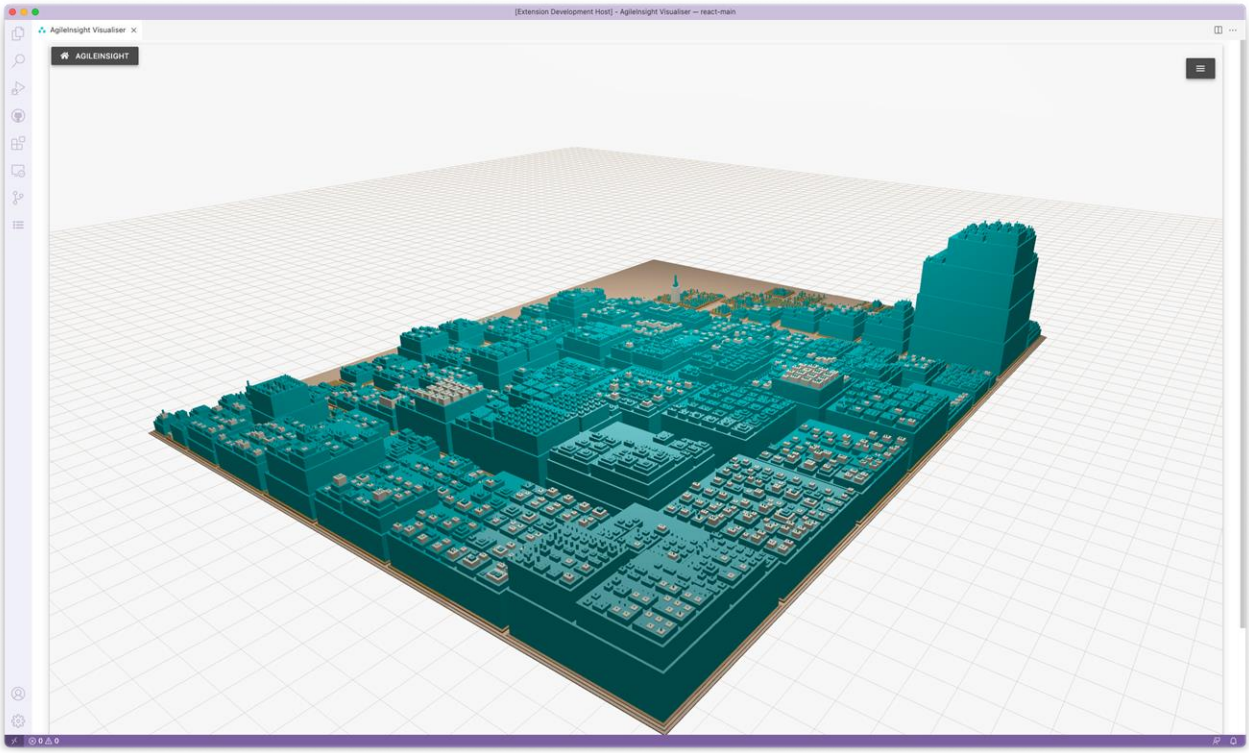
Figure 7.24: A visualisation of Node.js backend C++ engine (`src` package of master branch) as of 10th April, 2022. First view uses default normalised mapping, while the second uses capped linear mapping for primary containers. The third view displays irregularly-shaped methods, which is to be investigated in future.

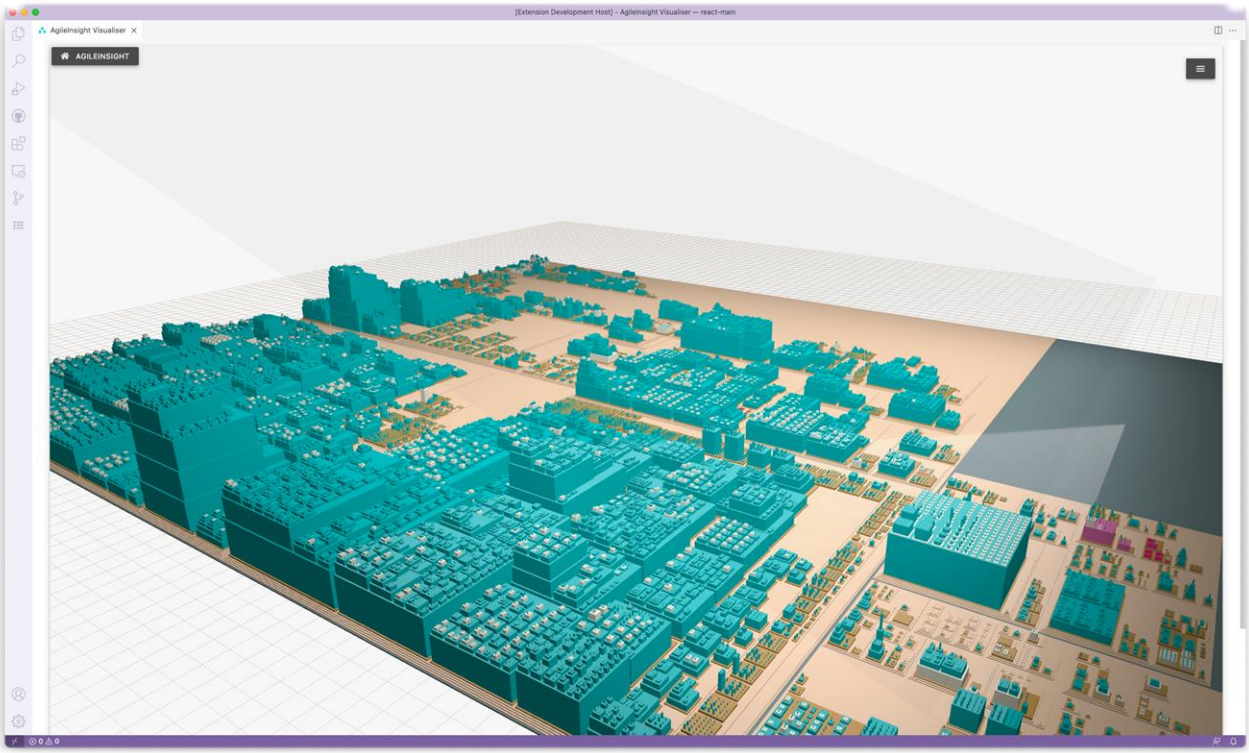
12- React—JavaScript

Below are four figures showcasing a visualisation of Facebook's React API, represented by its main Github branch[124] as of 10th April, 2022. It consists of a total of 1874 files, spanning JavaScript, CSS, HTML, Typescript, and C++ source code. The functional-based programming style is clearly visible, with top level and nested functions dominating the visualisation. Many files appear to adopt the 'Immediately-invoked Function Expression' model, or IIFE, that is common in JavaScript—as evident by those large buildings with numerous functions nested under a single outer function. The normalised mapping has barely any effect on this codebase, as demonstrated by the lower two figures.



[124] https://github.com/facebook/react


[Extension Development Host] - AgileInsight Visualiser — react-main
AgileInsight Visualiser
AGILEINSIGHT


[Extension Development Host] - AgileInsight Visualiser — react-main
AgileInsight Visualiser
AGILEINSIGHT

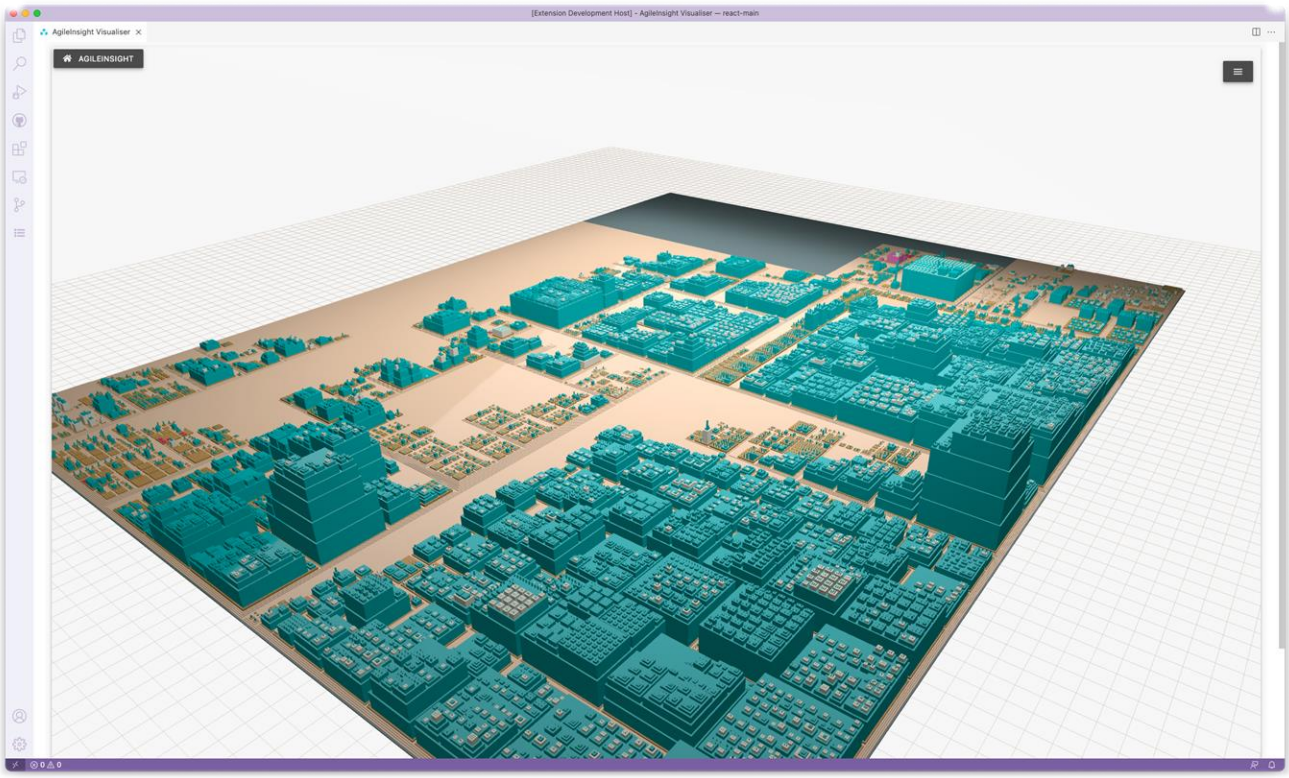


Figure 7.25: A visualisation of Facebook's React API as represented by its main branch as of 10th April, 2022. Capped linear mapping is used in first two views, showing an overall, then a focused view, respectively. Lower two figures use default normalised mapping (with little effect here), and similarly show an overall view then a focused one.

13- AngularJs—JavaScript

Below is a visualisation of AngularJs API represented by its master Github branch[125] as of 10th April, 2022. It consists of a total of 849 JavaScript source code files, and features a large number of 'locale' files (the identical looking buildings).

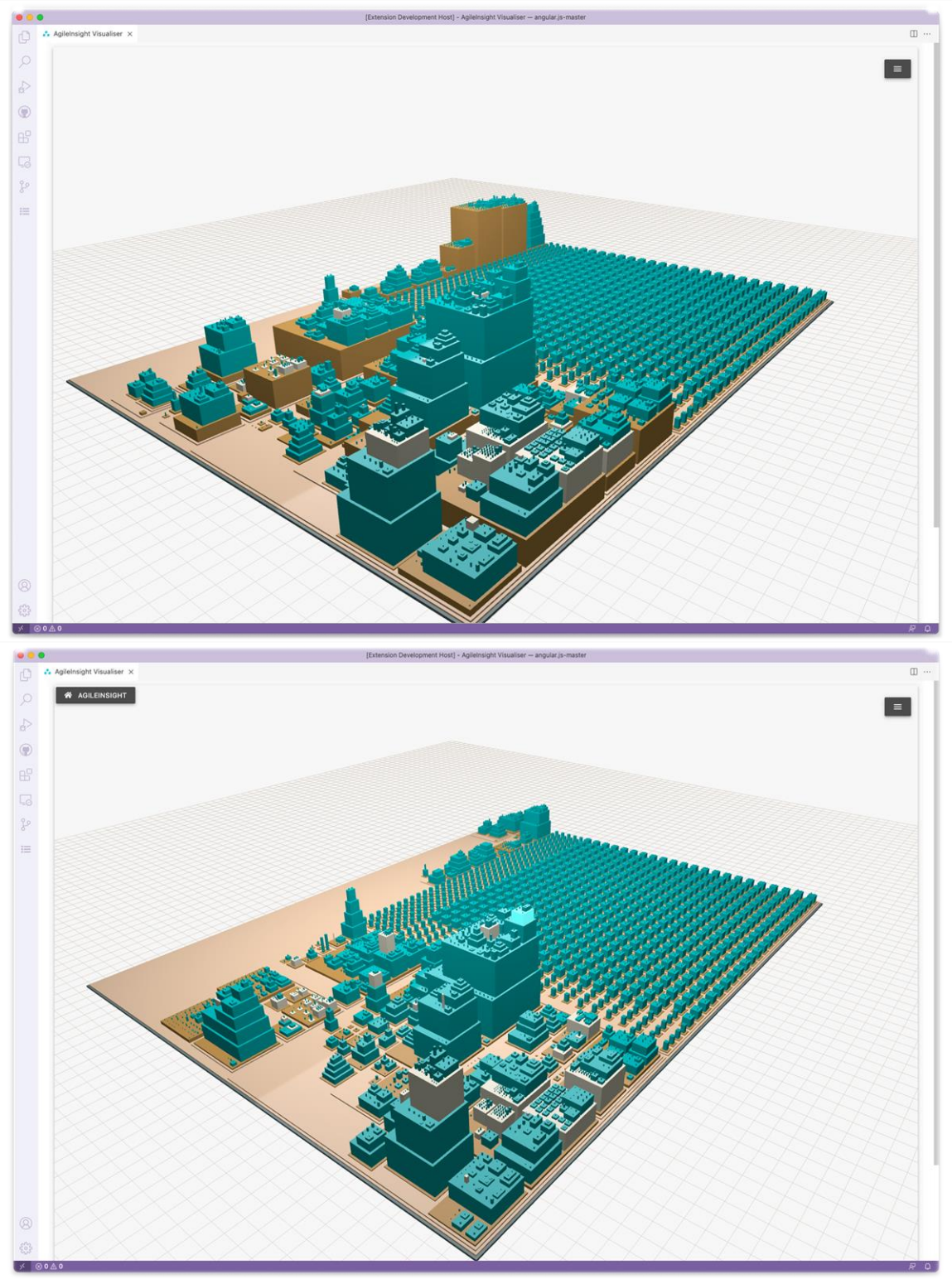
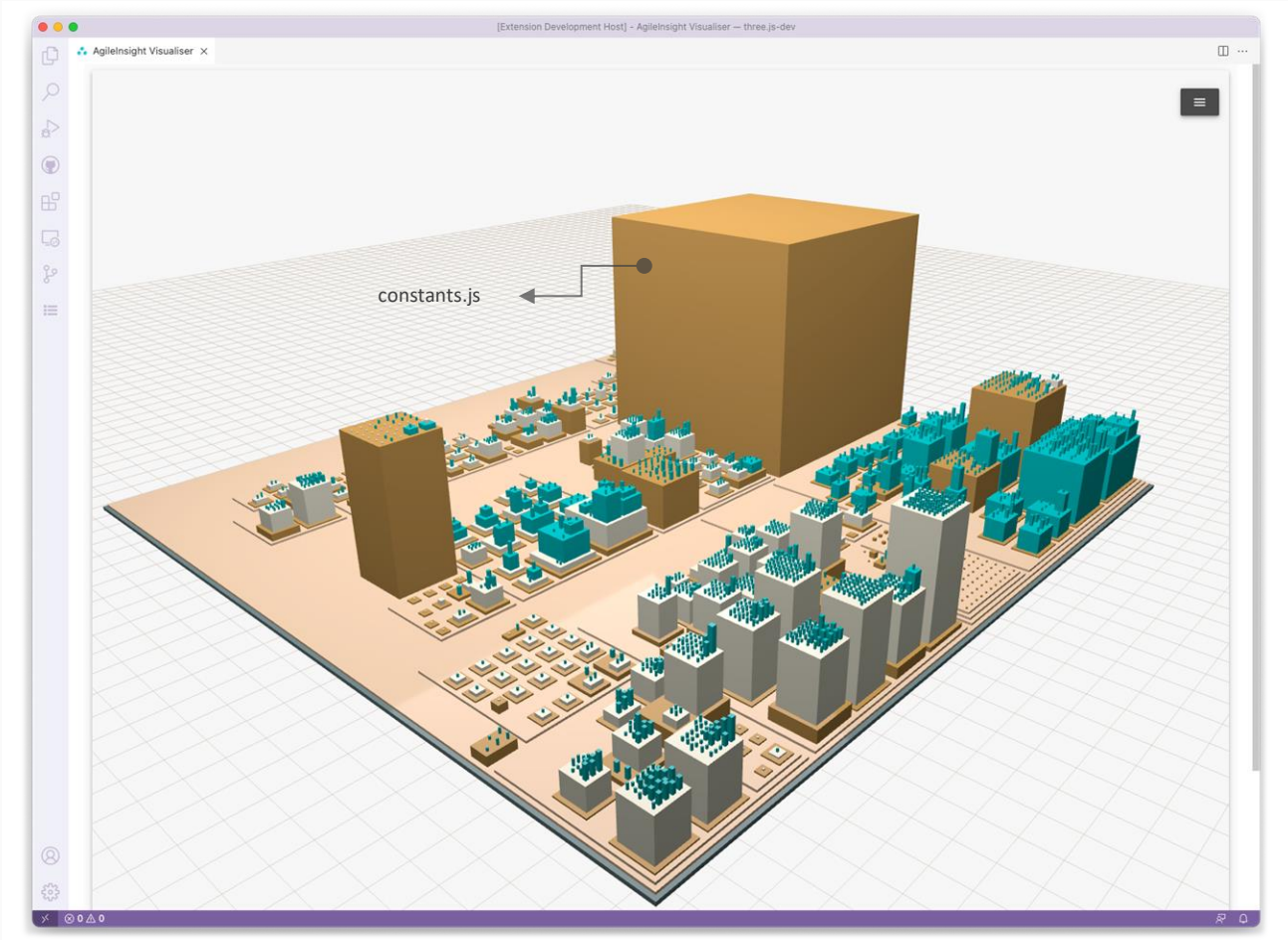


Figure 7.26: A visualisation of AngularJs library represented by its master branch as of 10th April, 2022. Top view uses capped linear mapping for primary containers, while lower view uses default normalised mapping.

[125] https://github.com/angular/angular.js

14- Three.js—JavaScript

Below is a visualisation of the three.js 3D graphics library represented by its Github[126] development branch as of 10th April, 2022. It consists of a total of 363 JavaScript source code files. When using the capped liner mapping, a towering building emerges representing its `constants.js` file—which contains a large number of direct children. The building is relegated to a more realistic size when using the normalised mapping technique (see annotation in second figure)—which better reflects the actual complexity in this case.



[126] https://github.com/mrdoob/three.js/

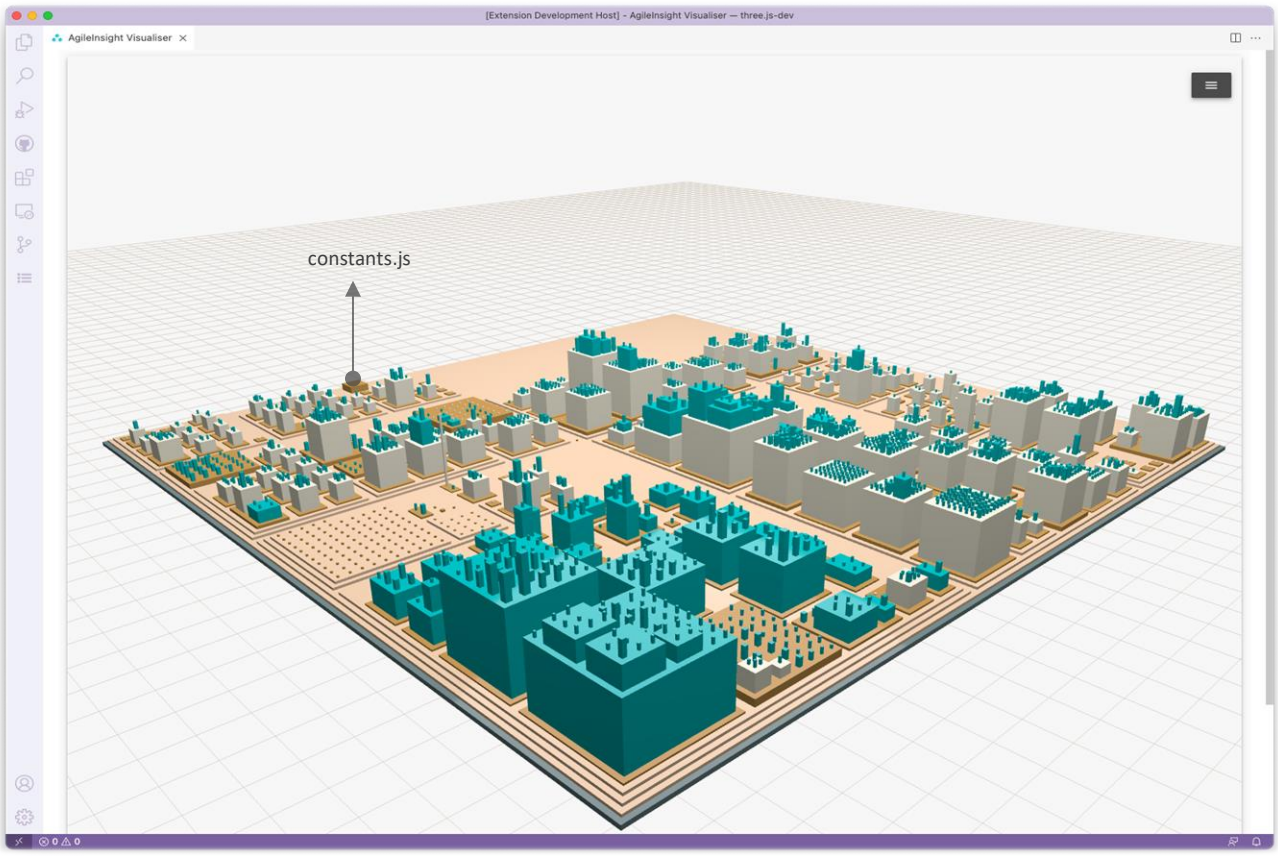
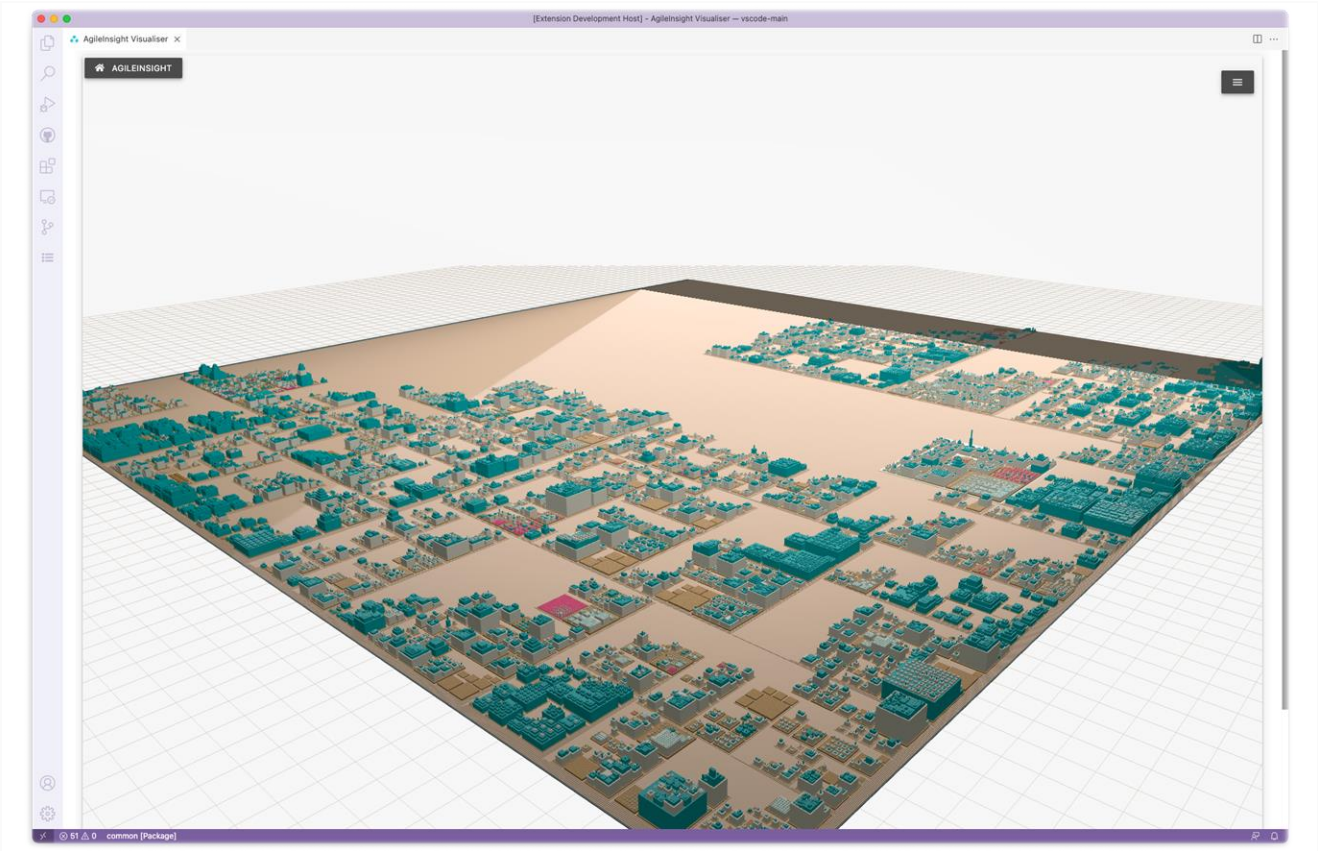


Figure 7.27: A visualisation of three.js 3D graphics library (used by AgileInsight to generate visualisation scenes) representing its development branch as of 10th April, 2022. Capped linear mapping is used in first view, while default normalised mapping is used in the second.

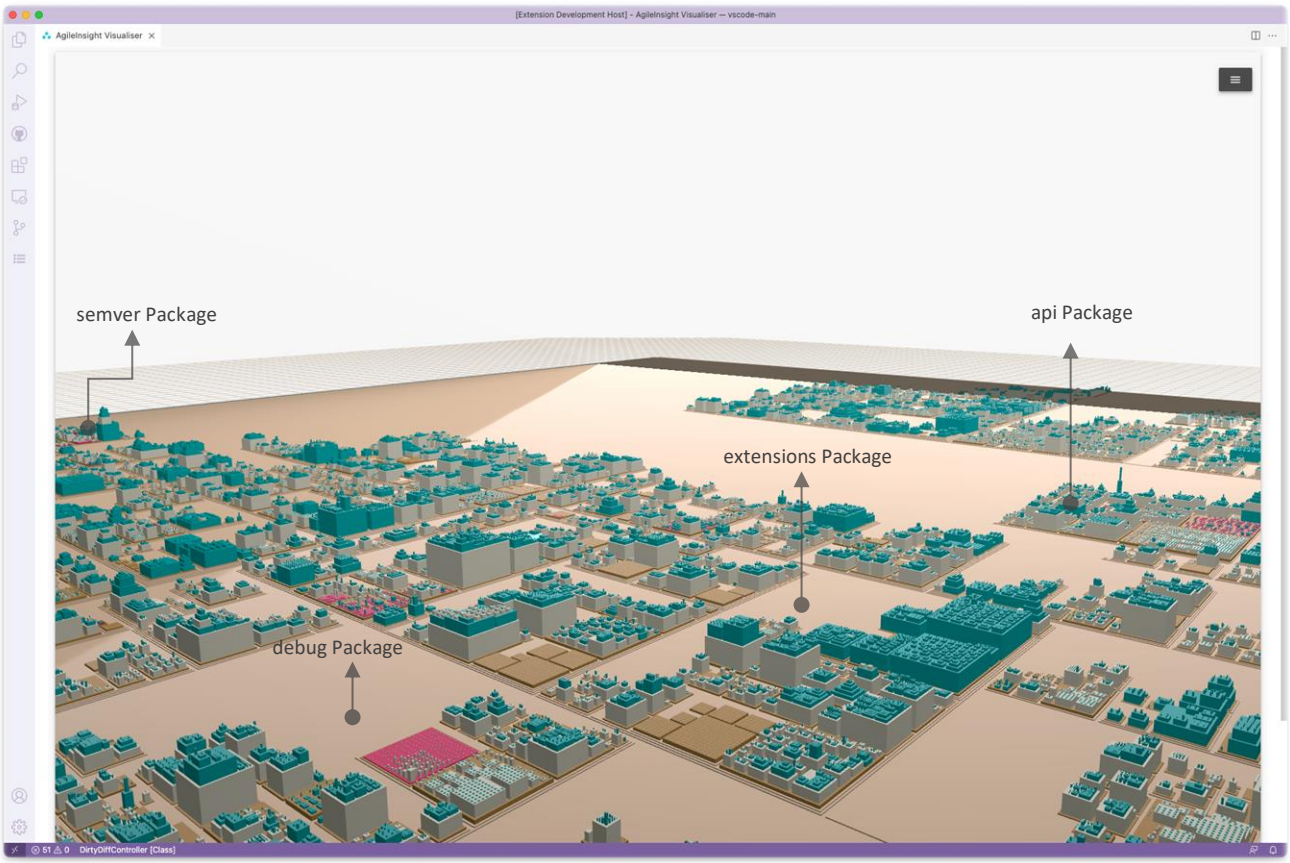
15- Vscode—JavaScript

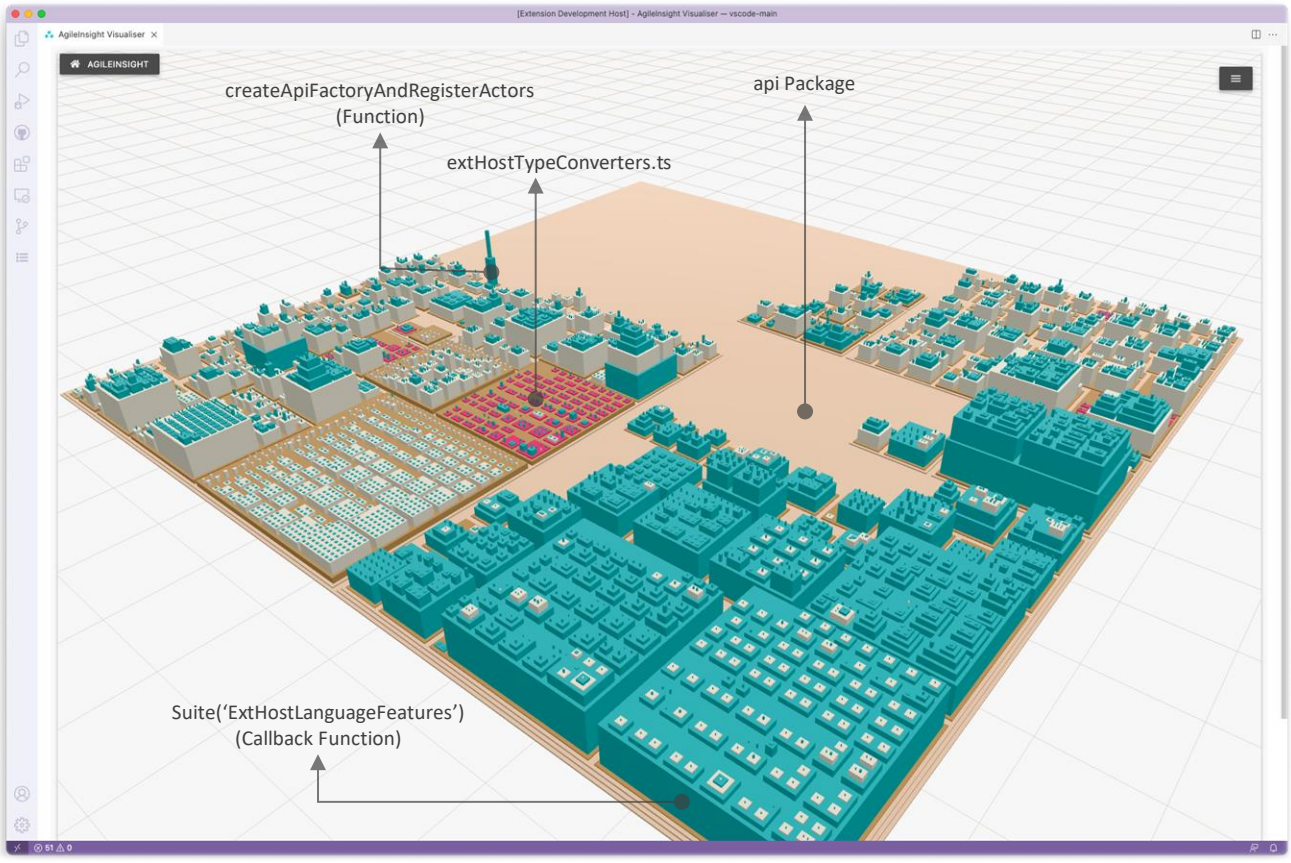
Below is a visualisation of Vscode's main Github branch[127] as of 27th April, 2022. It consists of a total of 3280 files, spanning Typescript, JavaScript, HTML, and CSS source code. The codebase is large and it was surprising to be able to generate the visualisation on a relatively low-resourced laptop machine[128]. In a real situation, users are not expected to visualise an entire master branch—at least not for daily uses. A focused view showing a selected module appears in the third figure.



[127] https://github.com/microsoft/vscode

[128] Navigation was hardly practical though.


semver Package
api Package
extensions Package
debug Package


createApiFactoryAndRegisterActors
(Function)
extHostTypeConverters.ts
api Package
Suite('ExtHostLanguageFeatures')
(Callback Function)

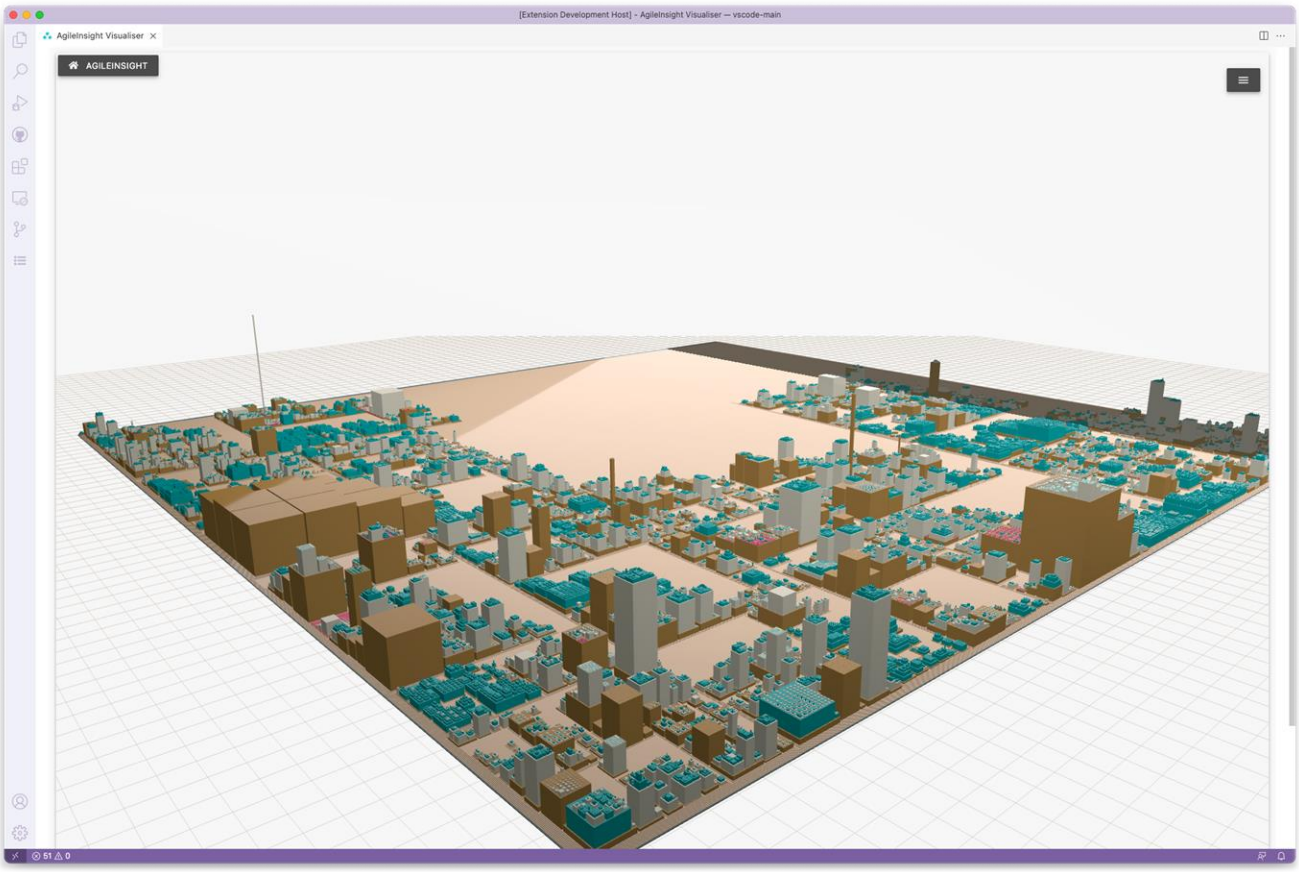


Figure 7.28: A visualisation of Vscode's main Github branch as of 27th April, 2022. Default normalised mapping is used in the first three figures, where an overall view, a closer overall view, and a selected focused view are shown, respectively. The fourth figure shows an overall visualisation of Vscode using capped linear mapping.

7.4.1 AgileInsight Benchmarks

Application/System	Languages	Total Files	File Parsing	Building Scene Node Primary Container—Capped Linear	Rendering Scene Primary Container—Capped Linear	Building Scene Node Normalised	Rendering Scene Normalised
Gin	Go	81	7s 468ms	180.559 ms	406.324 ms	160.607 ms	326.420 ms
AWS SDK for Go (Without Service)§	Go	409	1:09.754 (m:ss.mmm)	1.028s	680.335 ms	832.696 ms	769.529 ms
ISTIO	Go	1391	3:11.633 (m:ss.mmm)	1.172s	1218.615 ms	1.387s	1860.937 ms
Micro	Go	586	50.752s	959.111 ms	997.992 ms	1.077s	1248.363 ms
Facebook SDK for iOS	Swift, Objective-C, JavaScript	779	4:11.074 (m:ss.mmm) ————— 1:09.951 (m:ss.mmm) for 582 Swift only files	783.777 ms	973.541 ms	674.086 ms	1165.557 ms
CodeEdit	Swift	164	6.343s	103.381 ms	286.093 ms	72.271 ms	296.640 ms
RxSwift	Swift, Objective-C	996	4:91.29 (m:ss.mmm)	1.044s	1741.358 ms	854.425 ms	1522.190 ms
ASP.NET Boilerplate	Csharp, JavaScript, Typescript, CSS,	1401	1:51.425 (m:ss.mmm)	989.552 ms	1177.363 ms	935.747 ms	1092.880 ms
CodeHub	Csharp, JavaScript, CSS,	377	30.2s	574.347 ms	830.412 ms	474.566 ms	844.7958 ms
Django	Python, JavaScript, Html, CSS	889	1:02.744 (m:ss.mmm)	1.211s	1436.452 ms	1.376s	1660.082 ms
Keras	Python	661	1:15.816 (m:ss.mmm)	1.163s	1436.972 ms	1.191s	1557.525 ms
Node.js C++ Backend	C++	291	20:07.132† (m:ss.mmm) ————— 1:33.830 (for 136 files)	759.036 ms	903.129 ms	780.823 ms	1111.255 ms
Vscode	Typescript, JavaScript, html, CSS	3280	22:16.939‡ (m:ss.mmm)	8.967s	7306.753 ms	13.980s	32083.054 ms
React	JavaScript, CSS, HTML, Typescript, C++	1874	34:26.614† (m:ss.mmm)	3.071s	3123.739 ms	2.824s	3224.227 ms
AngularJs	JavaScript	849	36.34s	626.431 ms	669.864 ms	599.744 ms	829.927 ms
three.js	JavaScript	363	0:13.25 (m:ss.mmm)	265.392 ms	428.864 ms	262.554 ms	458.083 ms
iTrust	Java, CSS, JavaScript, HTML	582	23.620s	389.884 ms	666.052 ms	410.679 ms	621.383 ms

† C++ LSP server took considerably long to provision DocumentSymbols.
§ When Service module was included, total parsed files came to 1926 with a processing time of 4:18.543 (m:ss.mmm). Unfortunately, the Service module causes AgileInsight to crash during rendering of the scene. Investigation will be carried out in future. (Vscode had a total of 3280 files and did not cause AgileInsight to crash).
‡ In general, it was noticed that the parsing process would take considerably longer when it needed to process multiple languages. Improvement of the algorithm is certainly needed in future. However, the process is heavily reliant on the performance of Vscode language services module, as well as the individual implementation of each language's LSP.

Chapter 8
EVALUATION

8.1 Introduction

One argument that was laid in motivating this work was the lack of rigorous empirical evaluation of the software visualisation tools produced in academia, and how despite there being no shortage of such tools, barely any has made it to the industry or the working environment of the practicing user. This is in spite of the established benefits and advantages brought by these tools, as evidenced in the limited but prominent empirical results that have been generated. Researchers have been highlighting this issue for over a decade and recent important studies have focused on investigating the existing empirical work in the hope of identifying the key weakness points and presenting lessons and guidance for future researchers. For example, the systematic review of Seriai et al., and more recently that of Merino et al., both concluded that controlled experiments are very limited in software visualisation research and that the majority of studies fall in the category of case studies, simulations, and lab experiments (L. Merino et al. 2018; Seriai et al. 2014). Field experiments are virtually non-extant. Müller et al. argued that individual controlled experiments are not sufficient to establish general statements about the advantages of software visualisation—particularly those in 3D—due to the large number of influencing factors that must be controlled over a series of experimentations (Muller et al. 2015).

We argue that, in addition to the relative absence of field studies or limited rigorous controlled experiments, another vital aspect could also be behind the inability of software visualisation technology to reach the table and working environment of the developer: the lack of inclusion of the user and their environment in the design of such tools. If such tools are to be adopted by the practicing community, then they must be designed to fit and integrate unhampered into their working environment and into their technological infrastructure. Rigorous testing of isolated or hard to access tools is not sufficient to attract users to embrace these tools. The tool must demonstrate its usefulness in facilitating real tasks and activities in the daily workflow of the user—i.e., its tight relevancy. But it must also demonstrate its ability to transparently fit into the existing technology stack of the user—introducing a new isolated tool will hardly attract adoption, especially when the underlying technology is completely new to the user. User entry barriers need to be eliminated—or at least significantly lowered.

Thus, while conducting comprehensive empirical validation was not attained in the course of this work, the research has nonetheless addressed an important lacking aspect in the field. The intended end user has rarely been involved in the actual design process when it comes to existing software visualisation tools. While a few important works have been conducted to validate tools in controlled experiments (Baum et al. 2017; Fittkau, Krause, and Hasselbring 2015; Wettel, Lanza, and Robbes 2011), the design and development process of these tools are typically carried out independently in academic laboratories, and the user is only involved in the evaluation process *after* the tool has already been

produced. In their extensive 2018 systematic review, Merino et al. appear to point to this problem, when they labelled the disconnection between software visualisations and the real needs of the developers as the main threat to the field (L. Merino et al. 2018).

As elaborated in Chapter 3 this research follows the Hevner et al. methodology of design science research. A key aspect of Hevner's design science is that the end user and the application environment play a central role in the research right from the start and throughout the process. In our work, we have selected this methodology in order to address the lack of this particular aspect in present software visualisation research. The end user has been involved in this work from its inception, and has played an influential role in shaping the concept underpinning the work itself. Once the concept was solidified enough, collaboration with the end user continued to inform and guide the development and refinement of the product. The evaluation incorporated three phases of interview-based collaboration sessions where expert developers observed interactive demonstrations and provided reviews and feedback. Likert-scaled surveys were also used to add a quantitative measure to the evaluation. Results from each phase formed valuable input to guide and inform the next cycle of the work. The evaluation activities have in fact been very constructive and instrumental in facilitating this work and in building the tool. So, aside from providing tool validation, this aspect in itself is seen as an important and a distinctive advantage. Nonetheless, the feedback in each of the three phases shows promising interest from users and a dominating positive theme. Overall, the results form a strong motivation for undertaking more rigorous forms of empirical evaluation efforts—as well as continued development of AgileInsight—in the near future.

Remark. The central theme of this research is Applied Software Visualisation—a choice that intentionally focuses on integrating technology into the user application domain and environment, which appears to have only started to gain attention in recent years, as indicated at the beginning of this thesis. This is different from developing visualisations that address specific application domains such as software evolution, runtime traceability, exposing the static structure, or scaffolding comprehension (all of which have been well-addressed by prominent works). It is about putting the visualisation into practice and bringing it right into the middle of the existing user environment to support their daily tasks and activities—or as some might call it, to 'operationalise the technology'. Thus, while the evaluation effort in this work is largely influenced and driven by its software visualisation background, the *application* of requirements traceability—as will come to be noticed—is nonetheless a strongly present aspect of its features and functionalities. In fact, few of its features can stand evaluation independently and irrespective of the visualisation. However, despite this overlapping multi-disciplinary nature, the research's empirical background is primarily informed by empirical works from

the software visualisation field. Insights from the traceability domain have also been sought from the few relevant works that featured aspects of empiricism such as (Cleland-Huang and Vierhauser 2018; Rahimi, Goss, and Cleland-Huang 2017; Guo et al. 2016; Hayes, Li, and Rahimi 2014).

The rest of this chapter is dedicated to presenting and discussing the evaluation activities that permeated this work. Section 8.2 first presents the overall design of the evaluation. Section 8.3 then discusses the design of problem tasks and scenarios. Section 8.4 briefly covers the planning and setup of the interview sessions. Sections 8.5, 8.6 and 8.7 then present the details of each evaluation phase. Section 8.8 concludes with a summary of insights and directions for future work.

8.2 Overall Design

In Hevner's methodology, design research is empowered by three continuous cycles that feed each other in the course of the development. The Relevance cycle ensures that the application domain and the user environment keep guiding the Design cycle, while the Rigor cycle is constantly consulted to ensure the design is well-grounded and informed by the relevant literature. The research must also result in contributing back to the literature with the lessons learned and expertise gained—which the body of this thesis is hoped to fulfil.

The design of this research is shaped by three phases of evaluation activities that kept the design constantly informed by inputs from the end user, and in strong touch with the target environment and its application domain. Each of these phases are briefly discussed next, mainly to highlight the purpose and the general design of each. A diagram is presented afterwards in Figure 8.1 to provide a high-level picture of the overall design of the evaluation activity. Details of each phase are presented individually in the later sections.

Phase 1—Informing the Research Concept

This phase was carried out at a very early stage of the research. Its primary purpose was to better understand the user environment, and ensure the research concept as well as the technological infrastructure of the tool to be designed are strongly related to the environment of the user. In other words, this phase serves to establish the relevancy cycle. It played a significant role in shaping the final concept of the research.

Phase 2—Informing the Tool Design

After the results of phase one were studied and analysed, and after further review and study of existing literature, a prototype design was put in place. Expert users were then called in again to review this preliminary design before it was implemented. Valuable feedback and insights were collected at this stage that proved to be instrumental in further refining the design.

Phase 3—Preliminary Validation of AgileInsight

The results of phase two were analysed and studied to further strengthen the relevancy of the tool's design. The underlying concept has also benefited from this input leading to important refinements. Just as in phase one, expert users provided valuable tips and insights on how the tool could best help developers in the real world. With this information, the design was improved one more time and then

turned into a functioning tool. Expert developers were called in for the third time to now observe a demonstration of the tool and provide their feedback and comments. The result of this stage have been illuminating, and will drive yet future development of the tool to a stage where it can be made available for use by the practicing community.

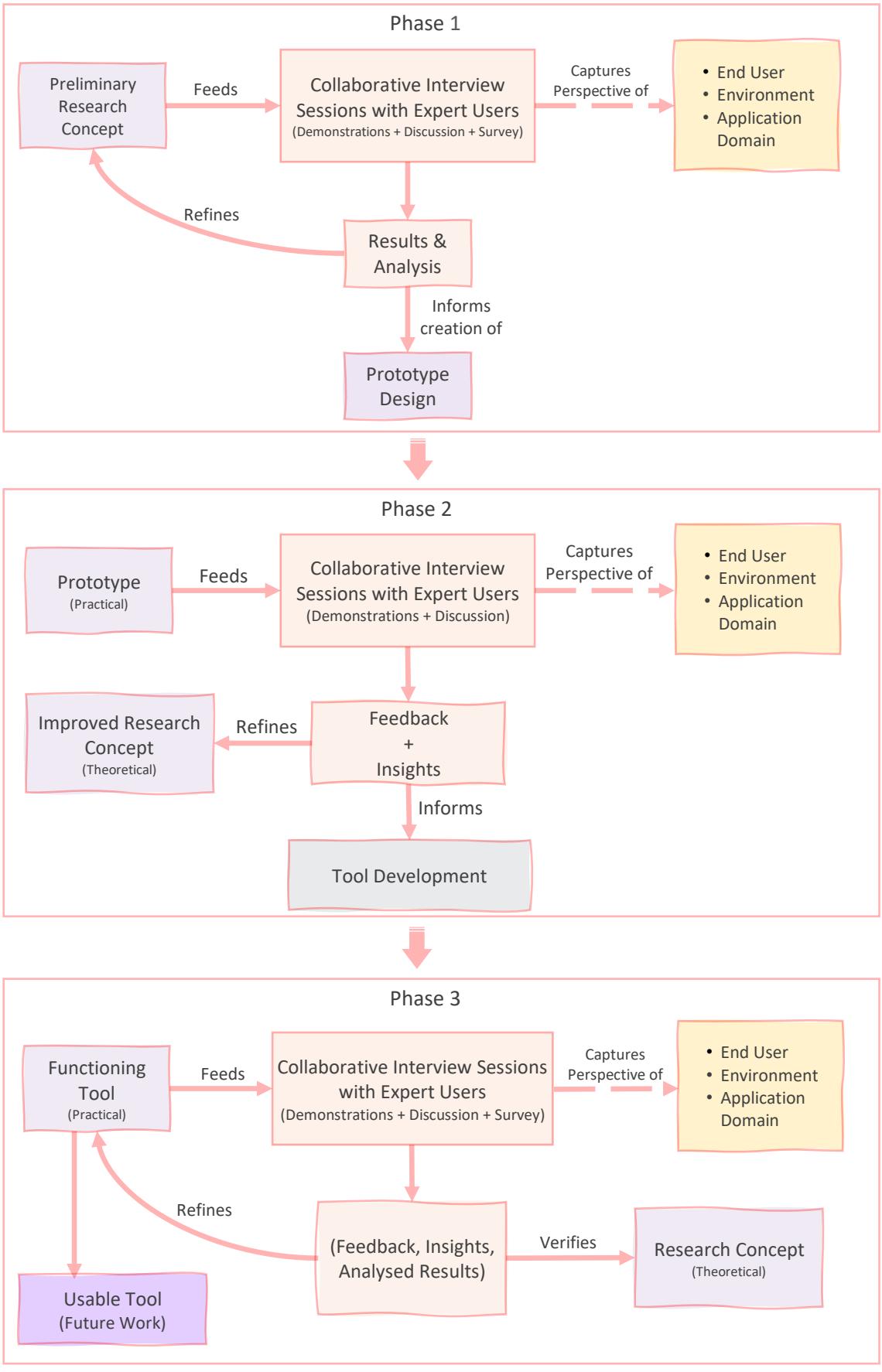


Figure 8.1: Overall Design of the three phase evaluations adopted in this work.

8.3 Designing Problem Tasks and Scenarios

Designing suitable user tasks is a crucial aspect in any evaluation effort that involves the end user. In this work, the tasks and their usage scenarios were important in the first phase in order to elicit user needs, to understand the difficulties and problems where the tool and its visualisation could provide help, and to understand their expectations of such a tool. In the second phase, the tasks were focused around the prototype's proposed features in order to cross check the general sanity of the design and elicit further improvements. As for the last phase, the tasks needed to verify the usability of the implemented features—some of those features were directly identified from the earlier phases, while others were newly proposed. They also needed to identify other potential benefits not examined earlier with the users, as well as collect suggestions and insights for improvements.

The design of tasks and general lessons for user evaluation of visualisation tools has received good attention since the early stages of the field (Knight and Munro 1999; Maletic, Leigh, and Marcus 2001; Sensalire, Ogao, and Telea 2009). This work has particularly made use of some recent literature in this regard, in order to guide the design of the tasks. Seriai et al. in their systematic mapping study focusing on software visualisation give some attention to the user tasks featured on the examined papers and note that they were primarily task-specific in nature, with only about one-quarter representing tasks of an exploratory nature (Seriai et al. 2014). They also found that the majority of those tasks were addressing the maintenance process. Brehmer and Munzner produced an extensive study in 2013—in the wider information visualisation domain—to classify user tasks in visualisation tools. They highlight an important finding in that study—that visualisation tasks are rarely isolated, but rather a series of interdependent subtasks are normally required to achieve a full objective (Brehmer and Munzner 2013). Their work builds on the prominent model introduced by Munzner in 2009, which was revisited by Meyer et al. in 2012 (Munzner 2009; Meyer, Sedlmair, and Munzner 2012) in which they systematically address the issue of evaluating visualisation tools, providing guidance to identify and validate threats at four layers of the design. Most recently, the systematic review of Merino et al. (2018) provides a comprehensive picture to the nature, scope, and category of user tasks that appeared across the corpus of their reviewed papers. The breadth of their classification was helpful in recognising potential tasks, but most importantly, it helped us to avoid areas that were over-represented in the extant software visualisation research. An important finding of their study, however, is that many of the tasks featured in these works were not aligned with pre-specified goals of the evaluation, and that some evaluations used such tasks to arrive at generalised conclusions. They hence recommend to carry out surveys to inform and collect the user requirements before a tool is built—an approach that is adopted in this work.

The tasks for the three stages were developed iteratively, in line with lessons learned from the above studies. For example, an objective was first determined and then a few user tasks were specified to fulfil that objective. In the later phases, many of the tasks were either identified from the first phase, or backed by the user feedback during that phase. Those tasks were then weaved into scenarios inspired from users' daily workflows, to make them fit in naturally and feel familiar to the users. In summary, the tasks and the scenarios are ultimately addressing a real problem experienced by the developers, or to improve their ability to handle certain tasks—and this is informed by the feedback and ideas collected from them.

Lastly, the work planned originally to execute a conventional user evaluation where the user performed the tasks themselves. However, during the pilot sessions, it was discovered that such an approach would be infeasible to attain in this work. It was realised that users need adequate prior time to gain sufficient familiarity of any non-trivial tool and its features, before they could be expected to execute work-relevant tasks. Unfortunately, given the three phases required for this work, the overall amount of time required to allow each user to obtain sufficient background sounded impractical. More detrimental though, the users had limited availability and a session is normally scheduled for 1 to 1.5 hours maximum. With such short timeframes, it is hardly possible and rather unwise to ask users to perform tasks themselves. Instead, a more appropriate approach was found to have the users act as individual observers in interactive demonstrative sessions conducted by the candidate. As is revealed in the details of the upcoming sections, the user gets to interact with the demonstrator by asking questions and expressing their objections or agreements during the execution of each scenario and its tasks.

8.4 Planning Interviews

There are slight variations to the setup and running of the expert interview sessions for each of the three phases. However, they by and large share a common theme and approach. This section presents the common design and setup detail shared by the three phases, while the variations are left for the specific sections of each phase.

Participants' Background: The number of participants in each phase ranged from 6 to 9, each with industrial experience in software development that ranged from 1 to 16 years. No gender preference was sought, but the group came to be mostly dominated by male participants, except for one case. The participants were mainly based in the Auckland region of New Zealand, and affiliated with major software businesses in the region. All participants had professional experience with agile development, and the majority were presently practicing a form of agile methodology in their workplace when interviewed.

Invitation and Selection: Participants were mainly invited electronically through direct email messages, or propagation of these messages sent via social networking. Participation was completely voluntary with no incentives provided to take part in the study. Interviewees who participated in each phase were re-invited to take part in the subsequent phases, but no enforcements or restrictions were made. A good number of participants (4 out of 6 in phase one, and 5 out of 8 in phase two) showed interest to re-participate in subsequent phases[129]. All respondents to the invitations were selected and successfully participated in the study, except for one case who withdrew due to non-availability.

Session Timing: An individual session was held for each participant. Each session was scheduled for a minimum of 1 hour. However, flexibility was maintained throughout, and sessions were generally allowed to extend to 1.5 hours if the participant's availability permitted and if they desired so. Overall, the majority of participants showed keen interest, and engaged naturally in discussions during and post the demonstration.

Session Design: Participants were provided with a brief overview of the study as part of the invitation. Respondents were subsequently provided with a detailed Information Sheet and a Consent Form. At the start of a session, participants were again briefed about the study—now with a slide presentation to better illustrate the study with diagrams and pictures. Each briefing took normally 10 to 15 minutes, after which the participants asked questions before the demonstration began. The demonstration had

[129] This is as recorded of voluntary expressions (without prompting) from participants. Participants were not specifically asked about their desire to participate in future sessions.

a relaxed run, and participants were encouraged to ask questions at any point. Discussions normally took place after execution of each scenario, and after concluding the overall demonstration. In phase one and phase three, surveys with quantifiable answers were handed to further capture participants' feedback more systematically. The surveys were completed before closing the sessions, and participants were allowed to ask clarifying questions.

Session Medium/Venue: In phase one, sessions were held in-person, and in a private meeting room of the education institute. In one exception, the session took place in a private meeting room in the workplace premises of the participant[130]. Sessions of phases two and three were all held over video conferencing applications, as they occurred during the pandemic period.

Session Conduct: All sessions were conducted by the research candidate carrying out the study. A supervisor was however present in most sessions to provide guidance when needed. They also took notes to record verbal feedback, and provided tips after each session to help refine and improve the delivery of the next one.

Data Collection: Verbal feedback was captured manually in phase one by both the candidate conducting the study as well as the attending supervisor. In phases two and three, the sessions were electronically recorded (with participant's consent). In phase one, the surveys were carried out on paper, while in phase three, they were conducted electronically.

Demonstration Design: The demonstrations were designed and scripted in advance, and were structured in scenario-based workflows. The scenarios were designed with lessons and guidance from the studies mentioned earlier. Care was specially taken to avoid leading words or behaviour. With the delivery of each session, tips and lessons were learned to further refine and improve the delivery of the next session.

Preparation & Pilot Runs: The sessions were structured and broken down into timed sections. While the timings were not strictly enforced in the actual runs, they were still important to help manage the overall session. Most sections were either scripted or had their content well-prepared in advance. Rehearsals were also practiced to ensure smooth delivery of the actual sessions. A pilot run was also conducted before each phase.

[130] In this particular session, the participant was joined by few fellow colleagues who were keen to observe the demonstration, and they also expressed feedback that was recorded along.

Evaluation Material and Approval: The evaluation work conducted in this study has been reviewed and approved by the Ethics Committee of Auckland University of Technology (AUTEC), under reference number 19/131 and dated 21 May, 2019. Appendix VI provides the relevant application documents.

8.5 Phase One

This phase of the evaluation had a unique characteristic from the other two phases. It served to bridge our earlier work (Alshakhouri, Buchan, and MacDonell 2018) with this research discourse by evaluating the output of the earlier effort. As indicated in previous chapter, the earlier work embodied a preliminary form of the concept underpinning this work, and evaluating its output—which was in the form of a prototype tool—was seen as a valuable way to establish the potential efficacy of the concept and justify taking it further to the next level. Another valuable objective was that the evaluation presented an excellent opportunity to present the concept—albeit in its raw and prototype form—to the intended users, and use it as a medium to capture their first impressions, elicit ideas, and record user requirements. In other words, it was a unique opportunity to test and discuss the concept of this work before it was even developed, and to engage with the target users, get exposed to their actual working environment, and listen to their thoughts and needs. As indicated in sections 1 and 3 of this chapter, this approach aligns very well with recent calls of researchers in the field. It is also seen as a distinguishing characteristic from other works carried out previously in the field.

A premature perspective of the concept that is developed further in this work was demonstrated in an earlier prototypical tool named ScrumCity. In brief, the tool attempted to demonstrate the concept of synchronising development process artefacts with their design artefacts. It focused on the Scrum methodology and demonstrated a mechanism to link user stories to their implemented code artefacts. It required an external XML file, where the links had to be recorded in a predefined format. It then used a modified version of the city metaphor to present a visualisation of the software structure where the code artefacts are mapped to their design artefacts via the provisioned tracelinks. It enabled users to carry out feature and concept location activities along with other related tasks by interacting directly with the visualisation or through a primitive UI where the features are listed. Its overall goal was to unify both aspects of the development process and present them to the user in one place. The tool was presented as an Eclipse plugin—built utilising the extensibility features of the plugin Vera by (Krebs 2012)—and worked with Java source code only. While far from comprehensive, this short coverage is seen as sufficient for the purpose here, and for the reader to follow up the discussions ahead. If desired, the reader is referred to the original work for more details (Alshakhouri 2013; Alshakhouri, Buchan, and MacDonell 2018).

8.5.1 Design and Setup

In addition to the design information presented earlier, this phase was constructed to conduct a premature demonstration of the concept using ScrumCity as a medium. The demonstration was meant

to work as a conversation starter where users get to verbally express their thoughts, opinions, and recommendations after seeing the demo. A Likert-scaled survey was then physically handed out to the participants to have a systematic record of their feedback.

Rather than running the demonstration using dummy or artificial data, the tool was instead prepared with a real-world dataset to strengthen the validation and give the scenarios a more realistic meaning. This way, participants could also better relate to the presented tasks and scenarios, and could see what the concept could offer in terms or support and capabilities. Moreover, the real-world dataset presented yet the opportunity to conduct a case study, which has been carried out in parallel.

To conduct the demonstrations, the same setup and machine described in section 7.2 (also see Table 7.1) is utilised. It was further connected to a large format display screen during the demonstrations. In the next sections the dataset will be first presented, and then findings of the case study will be discussed, followed by the expert interviews.

8.5.2 Developer Reviews

Sessions Setup. Building on the results of the case study on section 7.2, we used the Cassandra dataset to design a number of problem tasks based on development problems and activities commonly encountered in practice. Six participants from local industry took part in the in the evaluation. All participants were presently practicing a form of agile development, and had industrial experience ranging between 1 to 8 years—actual values were 8, 2.5, 2, 1, 1, 1. The evaluation was designed to be carried out in two stages. The first stage was to have the participants use the tool to solve the problem tasks after being presented with a short demonstration. As explained earlier, this was later deemed infeasible in this research's setting for the aforementioned reasons, and was replaced by demonstrations of those tasks conducted by the researcher. Few adaptations of the scenarios presented earlier were also illustrated to the participants. The second stage involved the participants completing a brief survey consisting of 13 Likert-scale questions, along with a general comment box at the end. Sessions were conducted individually for each participant and were scheduled for 1 hour each on separate days.

Participants were encouraged to actively engage in the problem-solving exercises by asking questions and proposing ways to approach each problem. Each participant was then given the opportunity to provide verbal feedback, before answering the survey questions. Appendix VII provides 1)The Evaluation Briefing Sheet and Problems Set, 2)The survey Questions, and 3)The Original Responses from the Participants.

Findings & Discussion. A prominent finding from the demonstration sessions—which was observed in the survey responses as well as their verbal feedback—was the clear agreement that the tool and its underlying concept were addressing real and everyday problems that the developers could immediately relate to. The developers expressed generally their approval of the tool's benefits, and some recounted real scenarios they had recently encountered describing how the tool could have significantly helped them in those situations. For instance, two developers from two different organisations indicated that it took extensive efforts for them to locate where a given particular functionality was actually implemented in some legacy system they were required to maintain. With the availability of such a tool, they stated that their effort could have been substantially reduced. Another developer particularly noted that the specific view seen in Figure 7.4(a), which shows the distribution and locality of an entire Sprint implementation (obtained by selecting a Sprint node in the features explorer) would have addressed the problem he had recently been tasked with by his manager in their recent Sprint Review session. Other developers highlighted that the tool would be potentially valuable to product owners, business analysts, quality engineers, as well as management, and that it would support a range of activities including regression testing, planning, estimation, and risk assessment (for change impact analysis).

In addition to the positive findings just reported, several concerns were also raised by the participants. This included recommendations for particular issues to be addressed to render the approach industry-friendly, and advice for additional features that they would find highly desirable. The main concern to be highlighted was the importance of keeping the effort of tagging user stories by the unique identifiers of their related code entities to a minimum. While some acknowledged that the benefits brought about by the tool would strongly outweigh the effort of a developer having to tag their feature card with a few class or method names, they still contended that full or even semi-automation of this process would have a high impact on the tool's adoption by industry. Others expressed that collection of this data point at the IDE side would be highly desirable so that developers perform the tagging as part of their repository commit actions, possibly supported by auto-suggested entries.

> Remark. As was evident in Chapter 6, the above finding played a key, and rather fundamental, role in shaping the techniques developed for actively collecting the tracelink knowledge from developers during their regular activities.

Other concerns that were expressed revolved mostly around usability issues and some desired functionalities. This included the need to have better focused views in both the city visualisation as well as the feature explorer. For example, once a feature is selected then all non-relevant code artefacts in the visualisation, and features in the artefact explorer, should be either hidden or greyed out, one

participant suggested. Some participants preferred to have the ability to select and work on a particular set of modules rather than having the entire system being displayed at all times—pointing to (and reaffirming) the recurring problem of visual overload and the need to devise techniques that reduce it. Other comments referred to; performance improvements, the option to switch to a 2D view, an ability to group classes or modules by dependencies, grouping features by epics to minimise scrolling in the feature explorer, having a reporting functionality to print out the set of code artefacts that developers have worked on while implementing a particular feature, having the tool as a standalone application, and finally, some were concerned about the maximum city size that the tool could generate, and its practicality when the city becomes extensively large.

> Remark. It is of particular interest to note that many of these feedback items from industry users find strong parallels to recommendations and issues raised—or called for—by researchers in the field. Some of these concerns were addressed in the course of this work (for example, a dynamic detachment was implemented to offer focused views of the visualised system, and a significantly improved design artefacts explorer was introduced). Many features still remain though that would be of interest to future work.

Needless to say, these concerns and comments are legitimate points and reflect issues that could potentially hinder the practicality (and hence the adoption) of the approach in an industrial environment if not duly attended to. They are thus perceived as important findings, valuable and instrumental to properly advance and inform the research. Also, many of these problems are in fact outstanding and well-known issues in the software engineering research community, with researchers continuing to propose new solutions to them. For example, the visual clutter or visual overload issues—and rather many of the concerns mentioned above—come to fall into the category of dynamic scoping, or what is better known as “Context + Focus”. It refers to the concept of displaying to the user only the part of the information that is most relevant to their current activity (hence the word, focus), while at the same time maintaining an appropriate view of their overall relationship to the whole (hence the word, context). This problem is typically approached by a number of filtering techniques that in many cases are not necessarily generalisable to other approaches (i.e., they tend to be solution-specific), which is understandable given the intrinsically different nature of the problem contexts. A few versatile techniques do exist, however (the most popular of which is the fish-eye method, even though it still remains limited to imagery scenes). New research work in this area has come to surface occasionally in the literature (Trapp et al. 2008; Heer and Card 2004; Ghazi, Seyff, and Glinz 2013; Luz and Masoodian 2010; Hasan, Samavati, and Jacob 2014). In summary, those findings correspond to legitimate research problems that this work set out to address by experimenting with various techniques to achieve a desirable and practical solution. Chapter 6 has featured some references in their due contexts, where a feature or functionality was backed by findings from evaluation activities.

Table 8.1 presents next a summary of the key issues and comments reported by the industry participants. Figure 8.2 shows a simple chart of the survey responses to give a quantitative feeling to participants' general disposition to the research's early concept. Actual questions are provided in Table 8.2 for reader's convenience.

Table 8.1: A summary of the key issues and feedback as reported by participants in their questionnaire responses.

Topic	Description (including edited/reworded participant quotes)
City Metaphor	Participant expressed their appreciation of the "city analogy" describing it as a "good" visualisation method.
Hide Unrelated Classes	"Rather than highlighting classes", some users preferred instead to "make all unrelated classes hidden from view" to reveal the specific target classes.
Avoid Scrolling	Participants suggested to hide all unrelated features in the feature explorer when user selects an object in the city, instead of simply highlighting the related features, thus avoiding to have the user to scroll to locate the highlighted features. Similarly, the same participant suggested to group the features by Epics to further minimise scrolling.
3D Performance	Participant remarked that "3D is a good idea as long as it is fast, otherwise 2D could be better"
Standalone Application	Participant expressed desire for a standalone application, in addition to the Eclipse plugin.
Reporting Options	Participant proposed to have reporting capability so that users can ask to print all code entities that were used for implementing a given feature.
Integration with Workflow tooling	Participants expressed that integrating the tool in their way of working and common workflow routines could render the tool to be more useful to them.
Automated Data Entry	Participants wished to have the meta-data entry being automated, in reference to the process of tagging features with the unique identifiers of their related code artefacts.
Appreciation of the Concept	Participant expressed their liking of the concept describing its visual representation as useful to developers, product owners, business analysists, QAs, and management helping them to identify and uncover risk, effort, and impact or particular user cases.
Visual Overload	The same participant above expressed concerns that the concept can be "visually overwhelming" and suggested to have "separation" of "modules and packages".
Concept being Very Useful	Participant described the concept as "very useful for all aspects of development including regression testing, planning, and estimation"
Grouping of Classes	Participants suggested to have an option to group classes and modules by their dependencies and associations.
Minimise Footprint of Tool	Participant expressed the importance of lowering users' involvement by automating the tool as much as possible, as teams do not want additional overhead of systems that they would need to support.
Predefined Hot Keys	Participant described predefined hot keys (used for navigation and interaction in ScrumCity's 3D environment) to be unsuitable for "production" environment as they increase the "learning cost" to "master a system.
Method Level Connections	Participant wished for a method-level traceability between features and implementation artefacts. *Note: ScrumCity prototype did support method level traceability but Cassandra dataset which was used in the evaluation did not have the required data to demonstrate this capability.*
Domain Knowledge	Participant stated that prior "Domain Knowledge" to be essential for a user to make better use of the tool. Specifically, they referred to the "terminology" appearing in the feature descriptions as making little sense without the required domain knowledge. (*This*

	further validates the decision to run the sessions as demonstrations rather than asking participants to perform the tasks themselves—as that would require adequate time for participants to gain familiarity with the tool and the dataset in use).

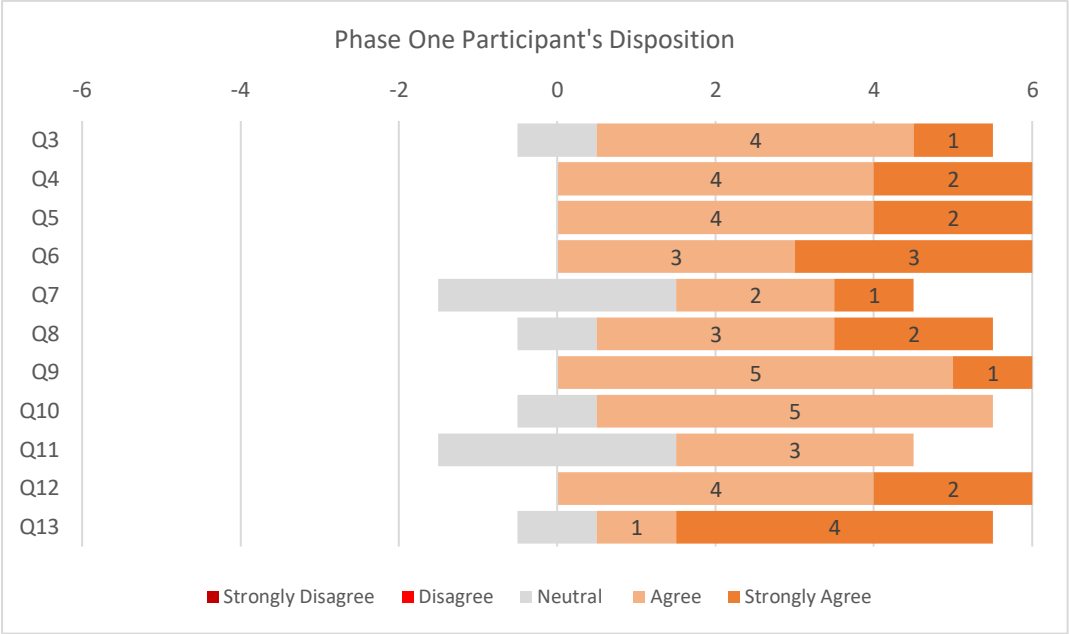


Figure 8.2: General disposition of participants regarding the early concept of this research and its demonstrated potential, as revealed by the preliminary survey. Relevant questions of the survey are provided in Table 8.2.

Table 8.2: Questions appearing in the survey of phase one's interview sessions with industry users. Only Likert-scaled questions are reproduced in the chart above, and respectively in here.

Question	Text
Q3	ScrumCity was helpful in developing quick understanding of the system's structure that is being visualised
Q4	The way that ScrumCity has presented the feature requirements in context with the visualised system structure is found generally helpful for developers
Q5	Making the feature requirements accessible to developers in context with system's physical structure has an added value in supporting some development or maintenance activities
Q6	ScrumCity was helpful in identifying interdependencies between the various system artefacts (classes or modules against original design requirements)
Q7	ScrumCity was helpful in tracing original functional requirement to their code implementations
Q8	The bidirectional tracing mechanism as presented by ScrumCity (functional requirements to code artefacts and vice versa) is beneficial for software maintenance and development activities
Q9	The concept of bidirectional tracing mechanism (functional requirements to code artefacts and vice versa) is beneficial for software maintenance and development activities
Q10	ScrumCity is helpful for design reasoning and exploratory activities of a system
Q11	ScrumCity can be helpful for newcomers to gain quick understanding of a system
Q12	ScrumCity can help developers in their feature location activities
Q13	ScrumCity can help developers to understand the distribution of implementation of a particular feature

Closing. To conclude this section, the preliminary usability evaluation worked as a self-gauging activity. It provided a level of confidence and an early confirmation that the concept underpinning this research—as presented in the prototype tool—has good potential utility for practitioners, and is worth advancing and pursuing further. In other words, it served to establish a solid justification for this research discourse. The evaluators' ideas for improvement presented yet invaluable insights that have guided this research. Lastly, it is important to highlight that the prototype tool, ScrumCity, is an artifact from earlier research, and is not a direct product of this particular work. Nonetheless, it embodied an early form of the concept, which this work has embraced to develop further and operationalise for use by the practicing community.

8.6 Phase Two

The feedback and insights collected in phase one have equipped us to develop the concept to a shape that is practical and relevant to real users in the practicing community. It provided a raw, but very important, initial understanding of how users would want to use such a tool and how it could be designed to support their activities. For instance, it highlighted the critical role of facilitating a practical tagging mechanism, and that without which, synchronising the design artefacts could hardly be brought into the realm of practice, let alone provide benefit to the user. It set the research on a path to investigate and study this issue in more earnest. However, it also served to affirm and highlight the aspects of the concept that are most useful and relevant to the user, and hence to give more focus on these areas; for instance, affirming the utility of the visual metaphor in exposing the code structure and the design concepts, affirming the relevance of the method level tracelinks, highlighting the need to reduce visual clutter, and highlighting the need to design the tool to seamlessly fit in users' way of working.

This represents good alignment and testament of the relevance cycle emphasised in Hevner's model, and how connecting to the user and their environment is valuable to better inform the research and steer it in the right path—not simply at the end of the research output, but more importantly in its early stages.

Informed by the above, a preliminary design of a tool was developed, and an interactive wireframe prototype was created to present this design. This phase of the study was intended to evaluate this design before the actual tool is built. More importantly, the new wireframe prototype allowed us to work more closely with the expert users to understand their way of working, their sets of tools, and to cross-check the design at a deeper technical level.

8.6.1 Design and Setup

This phase of the study was designed around structured interviews with expert participants from the industry, who were particularly practicing a form of agile software development.

Structure and Setup. Since this part of the study aimed to gain a deeper understanding of users' workflow and their toolset, part of the interview had to be structured with discussion and trigger points around this topic. The other part of the interview was structured to demonstrate functionalities and scenarios using the interactive wireframe prototype, in order to trigger further discussion around the technical aspects, to cross-check the design and its utility, and to gather further insights. The demonstration incorporated elements of the scenarios presented earlier. As was stated previously,

demonstrations were run by the candidate conducting the study, with each participant acting as an active observer, who was continuously encouraged to engage in and ask questions. Demonstrations were broken down into smaller parts, where after each, the participant's feedback and discussion were further elicited. Table 8.3 presents an overview of the interview structure adopted, while Table 8.4 shows a summary of the topics and discussion triggers used in the interview sessions. Rather than strictly steering the interview, those topics worked as guidelines and were adapted dynamically as needed during each interview. Appendix VIII provides detailed material regarding the setup and conduct of this phase, including: an Interview Protocol, an Interview Guide, and Guiding Scenarios.

Table 8.3: Brief Overview of the Phase Two Interview Structure. See Appendix VIII for further details.

Opening & Participant Briefing	5 mins
Demography & Participant Introducing Themselves	4 mins
Introducing the Problem	5 mins
Understanding Practiced Workflow/Toolset	20 mins
Wireframe Prototype Demo/Feedback	26 mins

Table 8.4: Summary of topics and conversation triggers/starters used as guidelines in phase two interview sessions

Typical daily workflow	Understanding Practiced Workflow/Toolsets
Main toolsets and technologies used	
How work is normally broken down and allocated within team	
Lifecycle of a new user story card (focus on final stages)	
Workflow of a change/maintenance activity to existing features	
Workflow of fixing a bug or an arising issue	
How information required for above tasks is normally obtained	
Are completed user stories/features systematically maintained as part of established workflow?	
If answer is negative, are the above still somehow accessible? How easy?	
Are there circumstances when that information had to be actually accessed?	
Impression on the idea of having completed feature cards made readily accessible within IDE (Retaining original design concepts & developer's intentions)	
Feedback on above concept + any expressed relevancy to one's own work/experience	
Other generic feedback to having original requirements integrated with code	
Reaction to the concept of feature-to-code tagging as demonstrated in the wireframe prototype	Feedback on Demonstration
Impression of the visualisation, and the technique used to map design concepts/artefacts onto it	

Perceived utility of the visualisation and mapping, and its relevancy to participant's work	
Feedback on the tagging mechanism as demonstrated	
Suggestions to improve the tagging mechanism, or to reduce its footprint	
Feedback on demonstrated application scenarios, including enhancements and suggestions	
Suggested new application of use	
Suggested enhancements to the visualisation facility	
General perceived value of tool & concept	

Participants. A total of 8 participants voluntarily took part in this phase, some of whom had participated in the earlier phase. Their years of experience ranged from 3 to 16 years of software development in industry as follows: 3, 10, 3.5, 12, 16, 11, 11, and 5. They held professional roles in the following areas either as their current or prior responsibilities: full stack developer, technical lead, Business Analyst, Developer Lead, Quality Assurance Engineer, Product owner, Tester, Team Lead, and UI/Backend Tester (order has no association with years of experience order).

Pilot Run. A pilot run was initially conducted with an industry user at an earlier stage. This proved helpful in refining, structuring, and better setting up the actual interview sessions.

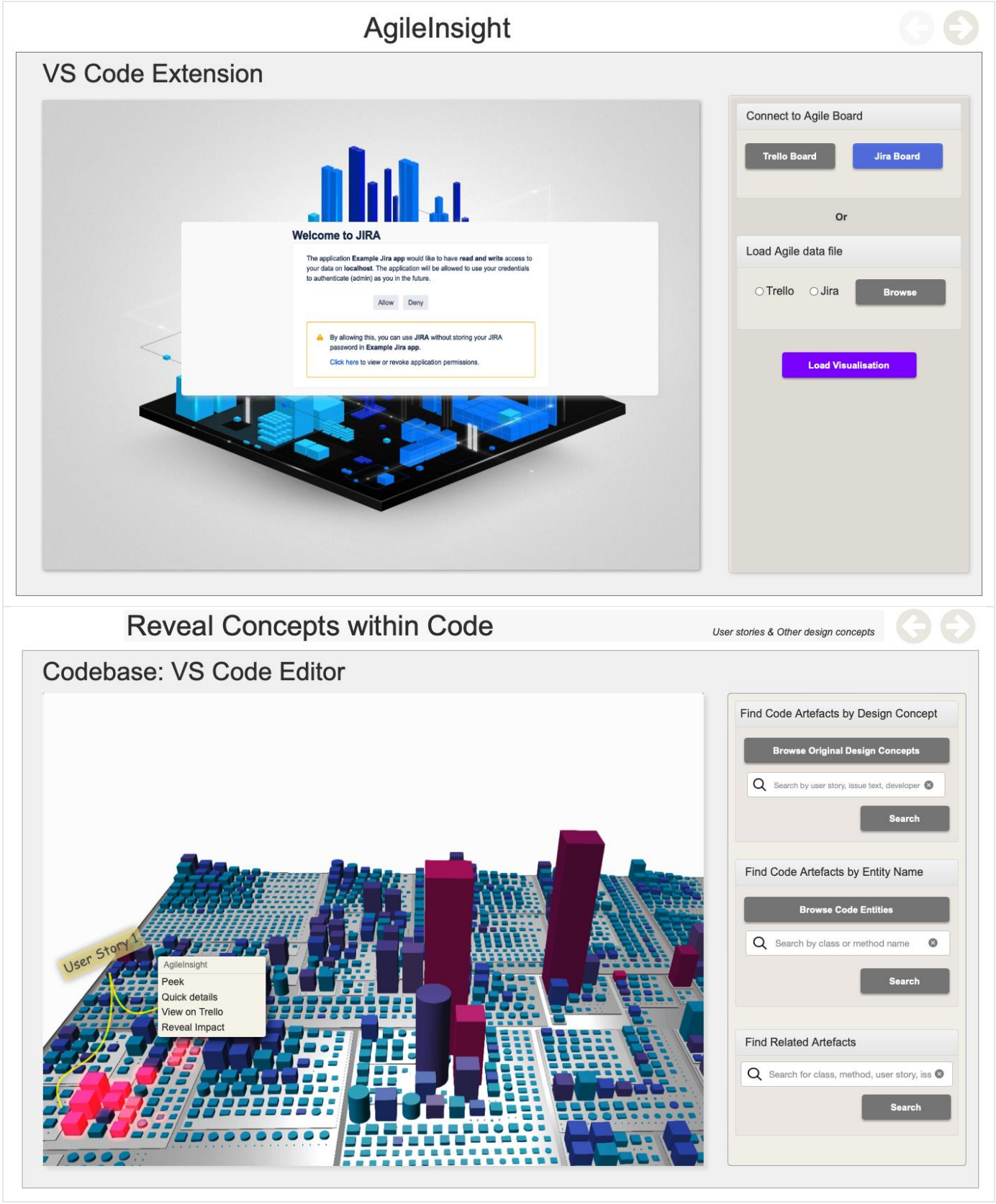
Running the Sessions. The pilot study and the first interview session were carried out in person. The rest of the sessions had then to be conducted over video conferencing applications due to the pandemic at the time. Other than these differences, the same protocols mentioned at the beginning of the chapter apply here as well.

8.6.2 The Wireframe Prototype

The wireframe prototype was designed to capture and represent the key aspects of tool, where feedback and input from industry users was seen as most important. This was mostly centered around the following areas:

1. How the tool would fit and integrate within the IDE
2. How the mapping of design artefacts to code artefacts is represented on the Visualisation
3. How that mapping could support a number of activities relevant to the user
4. How the tagging mechanism could be implemented
5. Applications and functionalities that become possible thanks to the above facilities

In order to provide a helpful reference and give proper context to the discussion, three screenshots of the prototype are provided in Figure 8.3. A complete set of screenshots is provided in Appendix VIII.IV



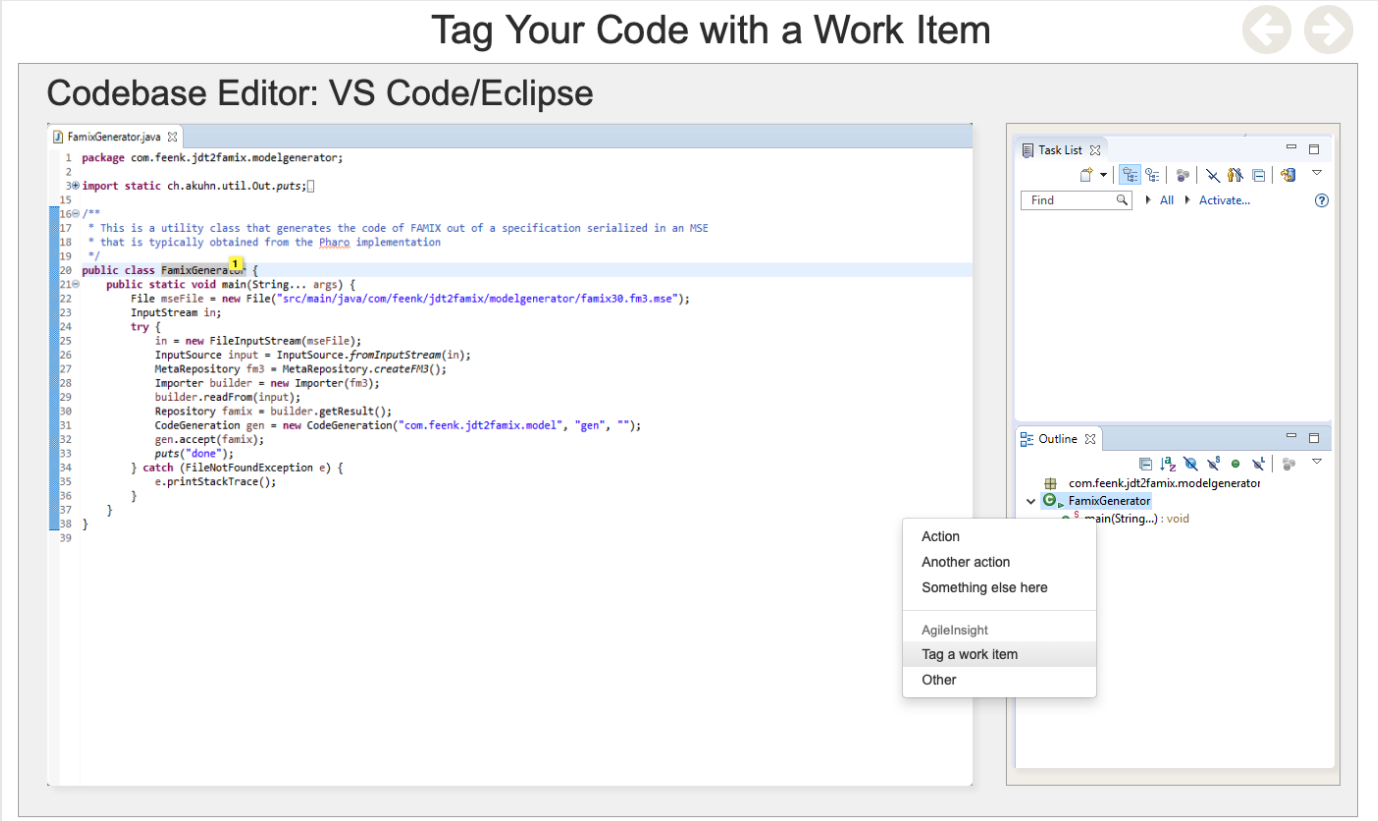


Figure 8.3: Sample screenshots illustrating selected views from the wireframe prototype used in phase two expert interviews. Features or functionalities depicted here are strictly for the purpose of the interviews to elicit conversations and give context to discussion. They are not to be taken as indication or reference to the actual implementation that was developed afterwards.

8.6.3 Discussion and Key Findings

As the primary concern of the interviews in this phase was to inform the design and the building of the actual research tool, the key findings are naturally focused around this purpose. The findings are also shaped by the topics presented in Table 8.4, which serves well to present a general scope and theme to those findings. Major areas of the findings are discussed next, and selected participant feedback is reported afterwards in Table 8.5.

1. Lifecycle of a Card

An immediate finding that became quickly evident is that original cards are almost always treated as discarded objects once they are closed. Statements like “once it’s closed, it is within the system but we don’t care about it anymore”, and “…stays somewhere there in Jira”, were heard from participants. This came as further affirmation to the anecdotal evidence that is often found amongst the agile practicing community (as well as in literature, to some extent). It can be well understood why such position is prevalent among agile practicing circles given the original and historical motives of the agile methodologies—e.g., the stance that user requirement management was over-attended to and overly

practiced in the traditional (pre-agile) approaches, often leading to unjustified costs and wasted project resources. The surprising fact, however, is that almost all participants indicated that those cards are not completely disposed of—an outcome that is reportedly the typical fate of such cards prior to the rising adoption of electronic dashboards, or for teams that still use physical cards and boards. It was found that, despite teams having no systematic or disciplined mechanism to retain those cards, many of them were still relying on their presence on a number of occasions. For example, participants stated that they would go back to closed cards on occasions where fixing a bug or introducing an enhancement to an existing feature required them to better understand the existing feature or its original design requirement[131]. In fact, the same participant that indicated earlier that the card '..is within the system but they do not care about it anymore', explained later that "when there is a defect, they we will go to the actual card to reopen it"—which appears to suggest that the availability of the card was beneficial by good luck, rather than being a result of a disciplined practice. Exceptions were, however, present as well. One participant reported that they could never recall a situation where they had to go back to a closed card—but yet they still affirmed the value of being able to.

Conclusion. This finding appears to indicate that, in spite of the fact that most agile teams would have originally discarded their physical cards, the presence of electronic dashboards—and the ability to simply archive a card instead of completely deleting it—has enabled some agile teams to begin to notice their value and find ways to fall back to them when they really needed them. Nonetheless, the practice was clearly far from systematic, let alone officially recognised and adopted by an organisation—it appeared instead to be often casually resorted to at an individual level, and only by some developers.

2. Existing Practice of Tagging

While no systematic and disciplined practice was identified for maintaining and referring to cards after they were implemented and closed, participants reported that their team encouraged some sort of practice of linking one's work to the original card or task they were implementing. One participant reported that their team was enforcing the practice. The practice, in those few reported case where it was encouraged or enforced, almost always occurred around the commit action or/and its accompanying message. One participant reported that their team placed the commit ID in the actual story card—acknowledging that while their specific team was not doing it, most other teams were. Another participant indicated that their team was including the task number in the commit message, and that "most people do it'', in indication to its non-systematic exercise. The participant that reported

[131] Developers appeared to prefer to try to locate the information themselves, and only resort to reaching out to the original developer if all their efforts have failed, as indicated by some of the participants.

their team were enforcing the practice, stated that their team enforced naming a branch after the user story or task it is implementing. A pull request would then automatically pick up this information and make it available through GitHub's 'gitblame' feature. When they needed to find the original card or task, a developer would need to go to the Github site, locate the relevant git blame, to find the pull request, which in turn would contain the card or task name/number. They quickly remarked then that a pull request would only pick up the name of the card/ticket if there were no conflicts in branch naming, otherwise the developer would manually provide the name. Yet, another participant reported that their team would "most follow the rule of" putting the user story number in the 'header' of a source code file (in reference to the top of the file). Other participants reported that their team had no practice of linking their code to the original card they are implementing.

> Remark. This finding is in line with the earlier remark of the rising practice among developers of mentioning a Jira issue number in their commit messages. In fact, teams who make use of GitHub issues will benefit from similar capability provided automatically by Github. Furthermore, during the later stages of this work, it was found that Bitbucket had recently introduced a feature to allow users to link their Jira or Trello cards to their Bitbucket commit messages or branch names. A power-up for Trello was also found to be recently available that allows developers to link their cards to GitHub issues.

Conclusion. The above findings seem to indicate that, despite the traditional view among agile practitioners that a card's purpose was transient and they held no value once they are closed, the community appears to find itself increasingly making use of them, thanks to the electronic dashboards making them constantly available. More importantly, teams appear to be realising the value of keeping a reference or a link between an original card and its eventual source code implementation—this finding happens in fact to affirm the position promoted in our earlier work. Of particular interest to this work, is that there is clearly no standard mechanism or a disciplined approach among teams. Instead, those teams who recognised the value of such links appeared to rather improvise their own practice to encourage or enforce a linking mechanism.

3. Reaction to Introduced Tagging

With the tool at the stage of a wireframe prototype, the tagging mechanism was still raw and did not have a fully developed concept—compared to the eventual mechanism that is arrived at in the actual development. Nonetheless, there was almost unanimous approval of the demonstrated tagging mechanism in the prototype. The level of approval varied between participants; a larger number expressed keen interest and readiness to use such a tagging mechanism, whereas a few were simply on the acceptance side. All participants were, however, seeing and appreciating the value it carried.

The impressions came mostly (and almost always) as an initial 'that is really good' position, followed quickly by a concerned 'but' remark. Participants liked the principle of a facilitated mechanism to link their code artefacts to their original card or design artefacts—particularly, one that is readily available within the IDE. They all had concerns though of how to make it more automated, and how to reduce a developer's involvement. In fact, this area of the demonstration received the highest attention and interest from participants, and a large number of suggestions, insights, and ideas for improvement were enthusiastically offered without hesitation for this particular area. This interest and earnest engagement is seen as an indication of an actual need, and perceived real value. Those suggestions are duly discussed in their relevant point below.

To provide examples of participants' actual impressions, Table 8.5 provides a few selected quotes and remarks on the demonstrated (preliminary) tagging mechanism.

4. Impression and Perceived Value of the Visualisation

Except for one participant, the majority expressed praise and endorsement of the presented visualisation, and the mapping technique it offered. The wireframe simply used images from the earlier prototype tool—but now annotated and presented in the context of a number of scenarios—to demonstrate the proposed applications (see Appendix VIII.IV for details). Nonetheless, participants showed excitement for the approach and offered a number of potential applications that they could readily see.

For example, one participant remarked that the view would make identifying change impact faster, another said that the visualisation would be helpful for testers to identify what code parts to target in their testing. Another participant commented that in addition to change impact analysis, the views were revealing how changes were 'scattered amongst the multiple different packages', and gave insight to potential refactoring when a user story implementation is found to be in multiple packages. One participant stated that the visualisation would enable them to 'move around' within their solution's point of views and offer an overall picture of what is being modified, within the context of the rest of the codebase. They further added that the layout and the sizes of the buildings were helpful to expose complexities such as god classes and identify where refactoring could be needed, and that the layout could be particularly useful to reveal a number of 'insights' about a codebase.

Example of participants impressions and reactions are provided in Table 8.5 for further reference.

5. Suggestions to Improve the Tagging

As indicated earlier, many suggestions were enthusiastically offered by participants to improve the tagging mechanism, mainly to make it as transparent and as naturally embedded into their daily workflow as possible. Unsurprisingly, participants whose teams already practiced some form of manual linking (four participants out of 8), showed more engagement and shared more ideas and insights. The suggestions can be grouped into the following themes:

- Automation
- Auto-suggestion
- Integrate into git commit action
- Remind User to Tag
- Maintaining Tags

All participants, after readily expressing their approval and support for the feature, immediately emphasised that it would be 'great' if it could be automated as much as possible—keeping user involvement as more of a confirmative action towards the end of the process. It is worthy to note that at this early stage, the demonstrated method in the prototype was a simple list of suggested design artefact names that a developer could choose from (or search for new ones). The list was offered as a contextual right-click menu that could be invoked on a class or a method name (see the third picture in Figure 8.3 and Appendix VIII.IV for more).

Participants were quick to highlight the potential of automatically retrieving information that is readily available (such as the number/name of an issue or a user story). For example, a participant whose team named branches after user stories being worked on, suggested to simply pull the name of the branch that is currently checked out. Another participant, whose team used to manually put a Jira ticket number in their git commit message, suggested to pull that information from the commit message. It was clear that the participants found the concept useful to them but at the same time they wanted to reduce their involvement. Most participants, after offering some suggestions for automation, came to conclude though that a manual option is still important to keep available. They preferred an approach where, as a default behaviour, the tool would do its best to predict the user story that is of interest but still offered the user to correct the information if needed.

As discussions were deliberated further, many participants came also to highlight the need to be able to modify or update the tags as the lifecycle of a project progressed. One participant commented that people do make mistakes when they code, so too they will make mistakes when they tag. They

emphasised that it should be possible to update existing tags. Another participant stated that some developers would sometimes commit their changes to the wrong branch, so in the case where the information was automatically picked from a branch name, this would result in a wrong linking that would need to be manually corrected. Aside from correcting mistakes, a participant also highlighted that as a project progresses, 'things move and are dynamic', so users should have the opportunity to update the tags accordingly, and to be able to add new tags to those already existing on an item. A few participants suggested that verification of tags should be included as a Quality Assurance step during code reviews or pull requests, and justified that when making a decision users will need to feel assured that the information is trustworthy—an issue that is well-researched in the traceability field.

Another interesting suggestion that a few participants pointed out—often towards concluding their thought deliberations—is that since developers often work on a single feature at a time, the tool could potentially identify that association without input from the developer. A good developer, one participant explained, would keep their commits small and specific to a single feature at a time. If the tool could pick up the parts that changed and automatically link them to the user story that they were working on then the process could be nicely automated, and then introduce the opportunity to edit manually at the end. A variation to this thinking was also suggested—such as to offer the process at a pull request stage—but the highlight was always the same principle, that integrating the tagging with a commit action was the most natural way for a developer to perform the tagging.

In an affirmation and acknowledgement of the importance of the tagging, a number of participants proposed to remind developers to tag their code. A participant whose team placed the user story number in a file's header comment, suggested to remind developers—just before they closed their editor—to provide the tagging if it is not done, so it can be properly enforced. Two other participants highlighted the need to provide some reminder mechanism to encourage developers to perform the tagging, when it is not done.

Lastly, one particular participant was concerned about where the tags would be maintained: was it going to be in vscode, in the local source code repository, or in the cloud-based repository?

Selected quotes and remarks are provided in Table 8.5 to offer a convenient reference to participants' original thoughts.

6. Suggestions to Enhance the visualisation

Some suggestions were also received to enhance the utility of the visualisation views. One participant suggested to offer more meta-data when the user hovered over a building. They wanted to see 'hot

links' to 'design elements' and to 'code pieces'. Another suggested to allow tagging directly from the visualisation, within the scene. Others wanted to see the packages annotated so the user knew where they were when they move around—affirming the need to address the orientation problem, a known and well-highlighted issue in the software visualisation field. One participant disliked the suggested idea of floating labels, stating that they would result in the scene being 'too cluttered'. Many other recommendations were offered, in fact, under this particular category, of providing interactive and contextual actions within the visualisation scene in order to make it more useful and expressive to the user. Different users wanted to see different information, that they saw more relevant to their work. One participant, for example, wanted to see interdependency hot links when hovering over a building.

On another matter, a participant was interested to know if the visualisation would support a single version at a time, or if it could somehow accommodate different versions. Another participant wondered how the metaphor would work for microservice-oriented architectures. Lastly, one concern raised was whether the visualisation would be produced based on user's local repository or their remote repository.

7. Other Suggested Functionalities

Other utility features were also suggested around putting the collected tracelinks to good use. For example, participants suggested that the system should implement some sort of a badge icon over code elements in the editor to indicate if that element is tagged or not. Other participants proposed a similar feature to highlight user stories that are not tagged. A few suggested further to have user stories within the IDE to be hyperlinked so they can quickly navigate to them. A participant suggested to offer concise information about a user story directly in the editor, in addition to the hot link. Lastly, one participated wanted to be able to select a code element and tell what user stories are related to it.

Table 8.5: Selected remarks and quotations capturing participants impressions and feedback at phase two of the evaluation study. These are provided to give a feeling of experts original thought lines.

Impression of Introduced Tagging	• "It would be awesome if it could be automated fully" • "I think the idea is great, but the question is how are you managing, how will you maintain this... ok so tagging is not a problem, people are happy to do the tagging...I assume people will be willing to do that because of great benefit later, so we can invest a little bit of time now" • "Really interesting! too much helpful" • "Good—Instead of writing in heading comments" *(participant was referring to their team's practice of placing user story number in source code's header/comment section of a file)* • "At the face of it, that is a great idea" • "One problem that we run into quite frequently not just at [###] but other companies I worked at, is having a decent sort of integration between your source control actually and your CMS, your Jira's and your boards… what happens is that we have that integrations that don't actually work so you actually tying these pieces of code with user stories something like this actually would be would be really really useful" • "in terms of the core problem… the linking of the work items to the code I think that is that is fantastic..." • "it is a sort of a single point of call to be able to actually look at a piece of code and tell where's it were and actually straight away see basically the parts it's involved in that particular change and how it effect the rest of the solution yeah... I think that is great"
Impression of Visualisation & mapping technique	• "The visualisation says that ah there is this class that is impacted so let us first test it" • "dependencies become easier to find" • "more important if new to code" • "Really good. Visually easier than searching through code. Used to ask developers for impact analysis document" • "Useful to learn about code base or impact analysis" • "Size of building not important, only which things are linked" • "definitely use it" • "Ah that actually looks great, I have not seen any sort of visualisation like this at all really, I think it is a great idea, I like the layout. It is not just for developers to get sort of a view of change impact but also it gives a good reflection of the design…, that is sort of a benefit on the side that you actually get from looking.."
Suggestions to improve tagging mechanism	• When asked if they find having to tag the code to be annoying: participant answers: "obviously having to do it manually is annoying like I might be ok with it but not other developers... they will get lazy, because tagging the commit and pull request is compulsory so to have to tag the code as well is gonna be annoying". When suggesting to integrate the process instead into the commit step itself: participant answers: "Yes, yes" • "My concern is how it is going to be maintained... when people code they do mistake, so when they tag, they will also do mistakes... so how can you prevent people from doing these mistakes.... aaahh, maybe in code review or pull

	request they also review the tagging. Otherwise people will need to know if they can be certain of this tagging” • “Tag at commit?!—makes more sense” • “… need to be reminded to tag before leaving code” • “I have second recommendation for you. The developers work fast jumping from file to another and they don't want to disturb their workflow to stop to do the tag... so I suggest…you know in vscode the way you work is when you pick a ticket it has a number… so when you commit the code, everything you touched is related to that feature right.. so maybe you don't need people to tell you, you can automatically do it,… so maybe you can do it through the code you commit and all the change…” • “Maybe that is the point… If you can say that all the changes in that commit are related to only that feature, then you can do it this way. because that is how developers should work... I mean if they have different changes in a commit related to multiple features then that is a bad practice anyhow. If developers follow good practice… they keep their commits small, and their commit to the feature, then you can actually automate it instead of them manually do it. • “That is what I think could be possible, and then you can introduce the opportunity to edit manually at the end, but you should automate it and then option for manual”
Other Perceived Utility	• When asked about displaying the impact of features mapped onto the visualised code: “yes, I do believe this is very one (sic), especially for code review, especially for tech lead or senior... they always want to know what is impacted.” • “Helpful for review to have this for code review (will make it quicker)” • “I definitely like the impact, I think that’s really good... but if you can also go a little bit further and describe a bit of the references so just like you provide info when you however over those buildings, you can also provide the references, how many they are back to those classes” • “To be honest, this is such a very useful one, because it is not just could be used by us but by testers as well. Tester would be like I do have limited time so instead of asking the tester what was impacted.. they can open this tool and they like ah this is the story and this is the story so they can bring up the acceptance criteria”. • When asked if projecting test coverage on the visualisation would be useful: participant answers: "Yeah, it would be awesome"

Closing. In concluding this section, it is relevant to draw a connection between the several features and functionalities that were presented in the earlier chapter and the findings reported in this section. It becomes clear that the feedback and numerous insights gained through this evaluation activity, have strongly informed, shaped, and guided the design and the implementation of the research tool. This is particularly true around the tagging mechanisms and the traceability applications offered by the tool. In fact, we consider this particular phase of the evaluation to have played a crucial role in enabling us to develop the research concept and its implementation tool in a shape that is relevant to the real-world user, to their way of working, and to their environment and its tools. The discussions with the expert users were also illuminating in that they asserted a number of concerns and topics that are traditionally raised by software visualisation researchers, and others raised in traceability research—as was indicated in few locations above in the discussion context.

It must nonetheless be acknowledged that this level of qualitative—and explorative—evaluation is far from a rigorous discourse that would be aimed at establishing the broad validity of the concept and its implementation as a tool. The expert interviews have instead enabled the research to establish strong relevance to the practitioner and their environment, hence allowing us to work towards the research objective of bringing the software visualisation application closer to the user and promoting its utility. In fact, at such a stage of design science research, explorative evaluation approaches such as expert interviews and demonstration sessions are perceived as especially suitable for the purpose as they allow sufficient flexibility to engage with the user on their terms, to understand their problems, and their needs. Thus, the research gets to be shaped and developed in accordance with the application domain that it proclaims to support. More rigorous empirical validation could come at later stages to assess and verify the claims of the tool. Expert interviews are found to be, in fact, common in steering design science research, as evident in examples from the proceedings of the 2015 international conference (DESRIST) on the field (Ruf and Back 2015; Kiesow, Zarvić, and Thomas 2015; Daeuble, Werner, and Nuettgens 2015). Some researchers have also expressed similar lines of thoughts around the potential of expert interviews, not just for design science in general, but particularly in the area of general visualisation research (Tory and Möller 2005).

8.7 Phase Three

This phase of evaluation was undertaken after the research tool—i.e., AgileInsight—was developed to a stage where it had sufficient operating functionality to demonstrate the research's core concepts. It comes to conclude the third cycle of the research and design journey, and represents the amalgamation of the lessons learned from the prior two evaluation phases and how they informed the research. At this stage, AgileInsight had been developed as a Vscode extension, and had the functionalities implemented and operating as described in Chapter 6. Other supporting functionalities were also implemented but they were not covered in the previous chapter. Some of these were of a user support nature—taking advantage of the now available traceability knowledge and the visualisation medium to offer the developer in-context information that supports some of their daily tasks and decision-making activities. Some were experimental in nature, and intended for preliminary assessment. Collectively, these functionalities represent improvements, refinements, and some newly inspired applications that were the subject of the evaluations and usage scenarios discussed in the earlier two phases. As stated earlier, the overall implementation of AgileInsight is a result of the experts' feedback that allowed the design to be fine-tuned to the users' needs, and the tool to be implemented to fit in with their application environment and way of working.

This phase was designed to conduct a preliminary evaluation of that implementation, offering early assessment of AgileInsight and its functionalities, in action. The evaluation consisted of 9 expert interview sessions involving demonstrations, verbal discussions, and detailed surveys. The demonstrative sessions made use of a real-world dataset for increased robustness, and this dataset was further the subject of a brief case study (see section 7.3) that additionally assessed the capabilities of the tool and its underlying concepts.

To assess and demonstrate the agnostic capabilities of AgileInsight, a number of open-source applications were briefly presented as further mini-case studies in section 7.4—where the focus was solely on the agnostic parsing and visualisation capabilities.

In summary, this phase of evaluation is shaped by four components, the first two of which were covered earlier while the latter two are covered in this section:

- A Brief Real-World Case Study (covered earlier in section 7.3)
- Open-source systems as mini-case studies demonstrating agnostic capability (covered earlier in section 7.4)
- Demonstration-based Individual Interviews with Experts
- Participant Surveys

8.7.1 Design and Setup

Similar to the earlier phases, the evaluation process of this phase was structured around demonstration-based interview sessions with experts from the agile practice industry. For the same reasons mentioned earlier, the demonstrations were again run by the student conducting the research, as opposed to having the participants carry out a set of tasks. Unlike the previous phase though, the demonstrations were at this stage run using a functioning tool, and a real-world dataset. In its broad structure, the evaluation followed the same general protocols outlined in the beginning of this chapter. However, there are a number of differences—some of which are distinguishing. This section briefly presents the key components shaping this phase, and in the process highlighting those differences. The details of each part is provided separately in following sections, where necessary.

Session Structure. The sessions with experts were broken down into a number of manageable and timed parts. The key aspects of the session involved a participant briefing delivered with the aid of presentation slides, followed by structured and interactive demonstrations delivered over three workflow scenarios. Each scenario was composed of several interwoven tasks inspired from prior phases and scenarios that overall represent the way of working as learned from earlier activities. Verbal discussions took place intermittently after major parts of each scenario were completed, in order to engage with participants and collect their first-hand and immediate feedback. Similar to earlier phases, participants were also regularly reminded to ask questions at any point during the guided demonstrations. After completion of each scenario and its feedback discussion, participants were asked to complete a related section in an electronic survey (see below). Similar to that in phase one, the goal of the survey is to capture a quantitative result that reinforces the qualitative feedback captured from participants' verbal discussions and comments. The sessions were designed to run within a minimum timeframe of 1 hour. Flexibility was administered, however, depending on each participant's availability, and where possible, verbal discussions were extended accordingly to capture as much feedback and insight as possible. Table 8.6 summarises this presented structure.

Table 8.6: Brief Overview of the Interview Structure of Phase Three. Presentation slides in Appendix IX provide further details of the session flow and topics covered

Opening & Getting Comfortable	~2 mins
Overview of session - graphical overview using slides & illustrations (see Appendix IX.II) - plan (3 demos/scenarios + summary survey questions) - timing	~3 mins
Introducing the tool - Motivation—why AgileInsight - Tool overview—illustration - Answer participant's questions	~3 mins

Overview of Scenarios & Dataset used	~2 mins
Launching Online Survey - (runs parallel with scenarios) - Complete Consent & Background	~2 mins
Running Demonstration of Three Scenarios, where each has - brief one statement overview - running of scenario tasks (by researcher) - brief verbal feedbacks - completing related survey questions	~40 mins
Concluding Discussion & Closing - complete final overall section of the survey	~8 mins

Tasks and Scenarios. In contrast to the interviews in phase two, where the objective focused on understanding the practitioners' ways of working, and then assessing the prototype functionalities in relation to this, the interviews in this phase focused primarily on evaluating the functionalities as now implemented by AgileInsight. The goal was to obtain early assessment of the tool's utility and potential benefit to the practicing community. In that regard, one purpose of this phase was to conduct a preliminary usability evaluation. However, specific attention must be given to the tool's seamless integration with existing users' workflows and ways of working. A secondary goal is to collect further feedback and insights in order to inform future development and improvement.

To satisfy the above, a particular approach had to be followed. There were two aspects that must be incorporated into the evaluation—demonstration of the tool's features, and demonstration of how these features facilitate and support common user activities. However, demonstration of each feature individually is not favored as individual tasks/actions hardly capture a meaningful use case. It would also miss encouraging participants to engage in a relatable scenario, and consequently missing the opportunity to obtain more accurate or complete feedback based on personal experiences. A workflow scenario involving a number of tasks is a truer representation of real world usage. This is in line with recommendations of other researchers as indicated earlier. Thus in order to produce workflow scenarios that feel familiar and natural to the participants, while at the same demonstrating the tool's implemented functionalities, and cover the planned user tasks, two steps were taken:

1- Implemented functionalities of AgileInsight were first arranged alongside the user tasks that they could potentially support, in a visual diagram.

2- A storyline was then developed that captured a real-world scenario relatable to developers, where a number of those tasks are interwoven into the storyline, and where those tasks were facilitated by the implemented tool's functionalities.

Three workflow scenarios were produced following the above process. To ensure a smooth and consistent delivery of the scenarios to all participants, a script was produced and practiced. Section 8.7.2 provides an overview of the three scenarios, while the full scripted versions are provided in Appendix V for reference. It must be noted that the scripted versions were purposely written in a relaxed and lay language. It intentionally avoided academic terms, replacing them with industry or other conventional terms that are more familiar to users from the development community.

The Survey. While verbal feedback was actively sought from participants after each set of tasks, capturing quantifiable answers was also important for the study. Unlike the initial survey conducted at a very early stage of the study—which was only intended to provide a first assessment of the research's concept as perceived by industry users—an elaborate and well-designed survey had to be developed for the purpose of this phase, which was now intended to evaluate individual functionalities and applications of use in detail.

An online survey was set up on the Qualtrics platform and designed to be completed by participants in a parallel manner, alongside the demonstrated workflow scenarios. It was felt that this approach would allow participants to record their feedback in a timely manner, while impressions are fresh in their memory. It was also seen to possibly reduce the tendency to fast-pace through the array of questions when handed out the full survey to complete at once. The survey was broken into a number of parts and integrated into the different parts of the interview (see Table 8.6). Participants were introduced to the procedure, and requested to launch the survey after the introductory part of the session. After completing a background section, they were asked to keep the survey open on the side throughout the session. Each time one of the three demonstrated scenarios was completed, and the verbal feedback on it concluded, they would subsequently return to the survey to complete the part related to it. To reduce the threat of influencing participants' answers (acquiescence bias), they were reminded that their answers would not be seen by the researcher conducting the study until the whole phase of interviews was concluded with all other participants, and that answers and results would not be associated with individual participants.

> Item-Specific Likert Style. An important aspect to highlight about the survey is that, unlike the generic Likert rating adopted in phase one (traditional agree/disagree scales), the survey in this phase adopted item-specific Likert scales where answers were tailored to suit each question. This approach is supported by research to result in more accurate answers, as participants would be less likely to make sense of the set of answers provided, without first reading a question properly (Liu, Lee, and Conrad 2015; Saris et al. 2010). The traditional agree/disagree scales are on the other hand reported to encourage the practice of 'straight-lining', and could inadvertently influence participants' responses under the inclination of wanting to be likeable (acquiescence response bias) (Sauro and Lewis 2016; Baxter, Courage, and Caine 2015).

The full survey and responses received are provided in Appendices IX.III and IX.IV, respectively.

A real-world dataset. This phase of evaluation utilised the real-world dataset iTrust that was reported earlier in section 7.3.2.

Participants. Expert user participants were invited from industry in a similar fashion to that in the earlier phases. A total of 9 invitees voluntarily accepted to take part in the study, and they were all selected and successfully participated—no incentives were offered. Five of the participants had previously taken part in one or both of the prior phases, while four had not participated in either phase and were new to the study. Three had industry experience between 3 to 6 years, whereas the remaining six all had experience exceeding 6 years. A third of the participants held a current role of a software engineer while another third held a role of a team/technical lead. The remaining participants were a software developer, a software tester, and an 'other'. Other roles held previously by participants included software architect, Scrum Master, DevOps Engineer, and a project manager.

Pilot Run. Before conducting the study, the complete execution of the session parts and its workflow scenarios were properly practiced. A pilot run was then carried out with one of the supervisors acting as participant. As a result, important improvements were introduced to the setup, the flow of the session, and the delivery of the demonstration scenarios.

Running the Sessions. All sessions in this phase were conducted over a video conferencing application, and were fully recorded. The demonstrations were conducted on a laptop machine with the Vscode environment running the AgileInsight extension. The dataset involved in the study was made available on a Trello board and a Github repository (see section 7.3.2). The vscode environment also had a corresponding local repository setup, and opened in its workspace. Table 7.3 shows the specifications of the environment used in the evaluation.

The next section presents a synopsis of each of the three scenarios, followed by example of tasks scripts. The evaluation results are then discussed in the following section.

8.7.2 Evaluation Scenarios

It was described earlier that the interview sessions were based around scenarios demonstrated to the participant by the student conducting the study. The scenarios were arranged into three themes based on the primary objectives of each:

Theme One: In-Development Tagging, followed by Selected Applications

Theme Two: Tagging Existing Code

Theme Three: Additional Visualisation and Traceability Applications

Each scenario is built around a number of development tasks and activities. These are then executed with the help of AgileInsight features, so by the end of each scenario, the participant would have observed those features and functionalities in action, and could provide feedback on them.

Scenario One Synopsis.

The objective of this scenario is to introduce the participant to the tagging concept implemented by AgileInsight. It demonstrates how AgileInsight enables a developer to tag their code on the go, as part of their development activity. It introduces two methods in this regard, and the focus is on assessing how well they integrate into a typical development routine. Some immediate benefits and applications facilitated by the collected tags are then demonstrated.

Features Covered:

Workflow for Tagging a Code Item In-editor	Related Code Items Viewlet for Feature Traceability
Workflow for Tagging a Design Item in Dashboard View	Revealing Visual Impact using the Visualisation (Visual Feature Traceability)
Design Item Lookup and Select	Design Item Quick View for in-context Design & Requirement Knowledge
Code Item Lookup and Select	Tagging Reminder Prompt upon Commit Attempt
Three-legged Tagging Operation (TTO)	Design Items Quick Actions
Feature Location to Support Development	Timed Tagging Reminder

Scenario Two Synopsis.

The objective of this scenario is to introduce functionalities of AgileInsight that facilitate tagging existing code. It aims to capture participants' reactions, thoughts, and perceived benefits (or interest) in post-development tagging. While in-development tagging is the ideal practice desired, post-development tagging could be of interest in some particular situations such as for obtaining certain safety certifications or achieving a quality standard.

Features Covered:

Code Lens for in-Editor Tagging Recommendation	Untagged Code Items Viewlet for Post-Development Tagging

Code Lens Quick Action	Untagged Design Items Viewlet for Post-Development Tagging
DI-Initiated Tagging	Using Visualisation to Identify Items of Interest/Priority to Tag

Scenario Three Synopsis.

This scenario is dedicated to introducing the applications and advantages of AgileInsight's visualisation components. It focuses on the capability of the visualisation to readily expose the source code structure, highlight areas of interest, and its potential in promoting a shared mental map of a codebase among team members. It also focuses on potential applications made possible by the synchronised nature of the visualisation—i.e., benefits that are enabled thanks to integrating the design artefacts within the source code.

Features Covered:

Utilising Visualisation Core Capabilities to learn about a new codebase, expose its structure, and complexity	Reveal Related Design Items from Source Code (Code-to-Feature Traceability)
Potential to promote Shared Mental Map	In-Editor bi-directional Traceability
Reveal Code Impact to expose implantation locality and distribution	In-Editor Design and Requirement Knowledge
Reveal Related Design Items from Visualisation (Code-to-Feature Traceability)	

8.7.2.1 Example Tasks and Scripts

The full scripts for all three scenarios are lengthy in nature. This section presents examples from the actual scripts for each scenario, giving a sense of the material. Readers interested in the full script for all three scenarios are kindly referred to Appendix V. A number of the workflows featured in these tasks are also showcased at the tool's profile page[132]. It must be noted however, and as has been as indicated earlier, that the scenarios were written intentionally in relaxed lay language, and are reproduced here as is. Clarifications are provided in square brackets when necessary, denoting executed actions. These should not be confused as part of the spoken scenario.

[132] AgileInsight's profile page: https://blaiski.github.io/agileInsight.page/

1. Example Script from Scenario One

On-Commit Tagging: A developer who is active in a development session, has just finished implementing a piece of requirement (or part of) and is about to commit their changes. They are shown instead AgileInsight's tagging functionality, which captures the tracelink knowledge from them and—as an added bonus—takes care of the commit operation for them as well.

> *"Let us say you had this UC card to implement [card shown on trello] and that you just completed writing this class with these two methods as part of your implementation [pointing to source code block in an open file]… you have also done your unit testing. Now when you are just ready to commit your changes… imagine you can do this [show, then click the hover tag action], right at your place, to tag what you have written of source code with the Id of the issue that you have just completed. Here… you now get this list (show Design Items Palette) where you can lookup and select your issue (UC, in this case)… You can also add your commit message here… and it is done (vscode shows tagging progress and displays feedback notifications). Now, your code item is permanently tagged in your repository with this issue you completed, which could be partially for now, that is fine. Your Jira (or Trello) ticket itself has also been tagged with this code item Id [show to participant]. Above here [pointing to source code block]… you see a tag has also been inserted right above the method name… and what is good is that your commit operation has been taken care of so you do not need to bother yourself about it. Here, let me show you what has just happened [participant is shown what has happened behind the scenes… the tags inserted automatically in 4 places].* ***So, no need to switch tabs, do a stage, and then a commit. Just click this hover, enter your issue Id, and you are good to go…*** *You have now created visible traceability in your source code, in your repository messages, and right in your ticket or card at the board. By doing this, we think you are also displaying transparency by letting your team know what parts of your code fulfils which requirement."*
>
> Note: pictures showing the above workflow appear in Figure 6.29.

Conversation Starter:

> *"We wanted this kind of tagging to be as low effort as we could so that developers can do it as part of their work routine without seeing it as an extra overhead… ideally we would expect the person who wrote the code to do the tagging as she or he are the origin of this information. However, we thought the tagging could still be done by a colleague doing a code review, or completing a unit test. What do you think?"*

Remaining scenario script appears in Appendix V.

2. Example Script from Scenario Two

Conversation Starter:

"After seen some benefits of having our code tagged with its original requirement, would you be happy if your team considered starting to tag your existing code on a slow gradual pace?... Ok, what do you think of tagging being done during a code review session? Or a by a quality engineer who is tasked with doing quality check?"

Basic Post-Development Tagging: A developer wishing to tag previously contributed code is shown a basic mechanism where code items recommended for tagging are highlighted across a source file that is currently open in editor. Code Lens feature is utilised to enable in-context tagging action.

"So, to start… let us assume you have already checked out the branch of your repository that contains the source code files you want to tag… so you have the files accessible in your vscode workspace. At a basic level, there is this simple method that could be used if we wish to go through scanning the source code and tag whatever parts that one has contributed. See, once we enable this code lens feature [Code lens is activated through command palette] then it sort of offer a visual guide highlighting code items of interest as we go throughout a file."

"Ok, here… imagine you spot this method that you have contributed few weeks ago. You read it quickly to refresh your mind of what it was for… and ok maybe you could remember a couple of keywords of the original issue or user story it belonged to. By hitting the code lens above here… we get this design items palette, which is basically a lookup tool that pulls up all your issues, bugs, user stories, etc., on any board on your Trello or Jira account… based on the keywords you are typing. As we start typing few keywords… we start seeing the relevant design items found… Here, this is the one we are looking for… We can select multiple items of course in case our method is related to more than one. Now, because part of the operation involves an automatic commit, you can edit the commit message if you like,…. but removing the tag is not permitted obviously. Now we just confirm, and that is it. We see the operation progressing… and, your method is now all linked up with your issue or issues… in the source code here, in your Github repo, and on the dashboard."

Remaining script appears in Appendix V.

3. Example Script from Scenario Three

Exploring a New Codebase: A developer uses the visualisation to learn and explore a new codebase. The visualisation technique helps to expose their code structure, reveal aspects of its complexity, and helps build a shared mental map.

"During the previous demonstrations, we had brief exposure to the visualisation… We think the technique itself has potential in enabling us to learn about our code… it appears it could give us a good sense of the code complexity and also its structure… like where things are situated across the codebase. We are wondering if it could offer help in this aspect, especially for a new comer for example..."

"Let me show you what I mean… see this is for example the iTrust codebase all visualised as we have seen earlier. You see, from the first glance, I could spot that there are two main modules… this one over here, and that one over there. By pointing over this module, I can now recognise that this is where all the Webroot files reside (Labels rendered directly on buildings were not implemented yet, so demonstration relied on basic labels revealed in status bar upon pointing). I can point here, and see right away that these are all CSS files… over there it's all html ones… and these over here are all JavaScript files. We are wondering if this could help in discovering the code to find out where things are located… Do you think the way these buildings are laid out reveal or communicate the structure of the code base? Do you think it could reveal insights about its architecture? Ideally, those buildings will be readily labelled, especially the outstanding ones… but we can of course actively explore around. For example, looking at all these buildings, we could tell by their whitish color, that they are all classes…[mouse pointed at one] ah, they are Java classes… and these ones on top.. are all methods! Those bluish ones over there are all functions… but why are they all nested up like this??… Ah, there is even a large class nested deep inside that big outer function! Do you think such activity is useful to identify irregularities about the code structure? Like to spot if a module is likely not conforming to the team's guidelines or coding practice."

Remaining script appears in Appendix V.

8.7.3 Expert Interviews—Results and Findings

The nine interview sessions with experts from industry were analysed to reveal any outstanding themes and key findings from the discussions with participants and their verbal feedback. As described earlier, opinions and conversations with participants were actively sought at certain points throughout the demonstrations, and these proved to be a valuable source of findings. This section presents the findings after studying and analysing the verbal feedbacks. It then presents the results of the survey to support those findings, and offer a more robust picture of the evaluation outcomes, and of AgileInsight's value as perceived by the experts.

Overall Picture. If the findings were to be summarised in one statement, it would be that the results of this evaluation phase came in strongly aligned with the previous phases, except that it offered a lot more detailed feedback and insights. A positive reception dominated the outcome. Indeed, as will come to be observed shortly in the discussion, the feedback, comments, and suggestions received started to repeat and converge, pointing to a sense of saturation around the key aspects. A good number of reactions, insights, and suggestions overlapped across the participants themselves in this phase, but also across the other two phases, and especially the second. A valuable point to be made here is that the fact that the evaluation was carried out with a functioning tool this time allowed the participants to see the research concept in action, and thus allowed them to navigate more intricacies around the implementation, its practicality, and to provide more detailed insights and suggestions as a result.

8.7.3.1 Key Findings

Tagging Mechanism as Implemented:

The point that stood out most around the implemented tagging mechanism is an immediate sense of perceived value and agreement, mixed with a desire for further automation of the mechanism. After demonstrating a few ways to execute the tagging mechanism, most responses were along the lines of 'that is very good, but…', followed by either a suggestion to further reduce the developer's intervention, or a question inquiring what the behaviour would be in a certain scenario. Almost all participants found the mechanism to fit well into their way of working—which was an important goal that was actively sought. In fact, a number of participants complimented the different methods offered to perform the tagging, describing that it offered flexibility and catered for the different ways people work, which they saw as valuable.

Fitting in with Ways of Working. With regard to practicality, ease of use and fitting in a user's workflow, a participant commented, "…it's pretty cool, I don't think it's adding too much of a burden", while another described it as "I think that it can fit into a process… without disrupting". Another added, "this makes complete sense". However, a few participants appeared to have a reserved reaction, and did not respond with explicit approval or the contrary, offering some enquiries instead.

Comparison to One's Existing Practice. A few participants drew comparisons to their current way of working, describing the level of ease and convenience the introduced method offered them. For example, a participant lamented on the series of steps they perform from manually copying a ticket number, tying it to their commit message, and then going over to a JIRA/GitHub web hook to complete the process by further entering it again manually. Adding that on top of that, when the time came to make use of those tags, they had to dig into threads of multiple commits and never ending scrolling. In fact, the same participant interrupted early in the session, asking if they could readily download the extension so they could use it.

DI-Initiated. A few participants seemed to prefer the DI-initiated method of tagging offered, as it allowed them to tag multiple code items in one shot. A participant described it as offering them the opportunity to focus on their current task, and delay the tagging toward the end of their session in a retroactive manner: "… OK, cool. Pretty cool. Yeah, that sounds good. If you… like been doing all the work and you want to do the tagging at the end, you can just go and do it. …I suppose that would be the use for that, or I guess retroactively tagging things, right?"

Recurring Theme for Automation. A theme that kept surfacing in almost all sessions was participants questioning if there was a way to further automate the process. While the process automated all aspects of the tagging on their repository and dashboards, they still did not quite like the fact that they had to actively do a lookup to select a design item (or code item, in the case of the DI-Initiated approach). For example, a participant asked "…do you have to go in and tag those independently, or can you just get them all kind of automatically tagged?". Another added "we probably need to come up with a way where engineers are either prompted to do it, or it just happens automatically as part of their workflow". A manager described his engineers as focused on current tasks and maximising efficiency, and are thus less worried about a future engineer or even themselves in future, asserting that to get adoption and a consistent result, the process must be automated further. It must be noted though, that the tool did offer a prompt as shown earlier. Most participants wondered if it was possible to automatically identify the changed code items, and then correlate automatically to the design item under development. A couple even suggested to let the user select the card (i.e., design

item) they will be working on at the beginning of their coding session, so that the tool could then automatically create the correlation—probably only asking for user confirmation. Interestingly, this strong call to further automate the process, and the insight to let the user declare their work item at the beginning of a session, coincided very well and reaffirms the position reached towards the end of the development cycle, as explained earlier (see section 6.10). At the time of evaluation, AgileInsight did implement automatic identification of changed code items, and it is used to facilitate a number of offered functionalities such as the reminder mechanism. It did not, however, offer the user the option to select their work item in advance.

Supporting Different Ways of Working. An interesting theme that surfaced frequently was that some functionalities would appeal to and fit very well into a participant's way of working, while others found it either inefficient for them or simply did not suit their existing way of working. For example, a participant (who also participated in phase two) was keen for the tool to pull the ticket number automatically from the checked-out branch name, while another participant explained that this method would not work for his team, as they combine multiple features into a branch. He preferred to select the design item at the beginning of the session instead. After an engaged discussion, they concluded that it is best that the tool be configured to suit different ways of working. "if you can set it up so that you can tag your code specifically to the way you work, that would definitely make it very usable. Because… everybody works differently anyway", the participant explained. In fact, after demonstrating the multiple ways offered by the tool to accomplish a certain task, some came forward readily, expressing their approval and appreciation of the fact that the tool could support different styles of working. For example, a participant commented on the four methods of tagging offered, "…to have this flexibility so it can be adapted to different… ways of working, that's really nice to hear. I absolutely agree… different people will have different ways of working", they further added in a later stage: "actually, the four methods which you have mentioned… the thing is that it is dependent on *(the person). For example, maybe I am a person who is good in visualization and so I can use that feature (in reference to tagging from the visualisation)".

Other feedback on the tagging process revolved around intricacies and refinements. For example, a few participants wondered what would happen if two developers were tagging a particular item simultaneously, or what the behaviour would be—and if the tool facilitated—modifying the tags in future or correcting a mistake. A participant was concerned about the inline tag labels attached to code items in source code growing too long if they became related to a large number of design items. A couple of participants wondered if the process would commit other unrelated changed files, or

correlated unrelated changes in the same file where an item is being tagged[133]. Yet another participant wondered what would happen if they did not know the ticket number, but then were reassured when it was explained that users could search using any keyword from the ticket title or description. Feedback along those lines offered insight for future improvements, while a few others were explained as already facilitated by the tool, or planned in future work.

Post-Development Tagging:

The overarching result and feedback received on this functionality was dominated by a sense of it being 'overwhelming'. Nonetheless, participants did express the value and importance they perceived in being able to tag their code in a retroactive manner. For example, a participant stated: "Yeah I can see where that come in handy…", while another commented: "yeah, that's a good idea…". Many described it as particularly suitable for a new project: "I can see this useful if you have a relatively new project, something that is … within reasonable amount of work to be done. Yeah, it's definitely helpful." All participants, however, expressed reactions along the lines of "I think it might be a little overwhelming to come in and start tagging an existing codebase if you got a huge project already…". The feature was indeed not targeted to cover large existing codebase and legacy systems, but rather to facilitate coverage for relatively new projects. (Admittedly, this point was not made clear during the demonstrations, which was a shortcoming.) Interestingly, it offered a good opportunity for feedback as participants provided a number of insights on how to make this practical for existing projects.

Most suggested to introduce checkboxes so users can quickly tag multiple items at a time[134]. Some suggested a mechanism to link legacy code under a few large container tickets in order to get them out of their way. Another suggested to customise the list based on what the user is currently working on, in order to encourage them to address those parts that are most relevant to them. The concluding remark was that a way should be introduced to approach untagged items in a large existing codebase to allow for graceful and gradual handling.

Nonetheless, most participants favoured the various list views as compared to the code lens mechanism, except for one participant who found the code lens to offer a relatable context to a developer working on a particular file—as it provided a scoped down view to only the part they were

[133] The tool does not commit unrelated changed files, and while unrelated changes within the same file where an item is being tagged will be committed at present, they will not be linked or tagged. Future improvement will introduce partial staging mechanism to further exclude unrelated changes from the commit, even within the same file where an item is being tagged.

[134] This was a straightforward and natural way to approach such feature, except that present Vscode extension API does not offer much flexibility with their native view components. Either a custom view needs to be developed to cater for this, or future release could see more flexibility introduced to Vscode Views.

working on. Participants also expressed comfort with the 'Untagged Design Items' view, as it offered a far less daunting picture by showing instead the cards (or tickets) that are untagged. Many found it, however, as more suiting the needs of a product owner or a kind of manager, rather than developers.

An interesting remark was received on whether the untagged lists views were synched across multiple users. One participant questioned: "have you thought about how you sort of manage the state across multiple users of this kind of thing?". Fortunately, thanks to the design decision to build AgileInsight functionality to run in a complete live on-the-fly manner with no hidden middle files or caches, such issues—along with others of a similar nature—become naturally irrelevant. This point is further explained at the end of this section.

Lastly, an interesting comment received was to consider building a process that automatically extracted information from existing repositories in order to automate the process of tagging already existing codebases. As explained earlier, the process created to load the iTrust dataset does indeed cater for this kind of automation—but the data need to be readily offered in a file. Tools to extract such data from repositories also do exist and have been utilised by researchers, as explained in earlier sections.

It must be noted in concluding this part, that it was to some extent surprising to find expert developers expressing interest and openness toward tagging old and legacy codebases. Before the evaluation, the impression was that the idea would be outwardly dismissed. The mechanism was implemented in AgileInsight to mainly offer a retroactive opportunity for relatively new projects, or for special application contexts where complete coverage was required to obtain a standard or certification.

Reminder Mechanisms:

AgileInsight offered two reminder mechanisms to prompt developers to tag their code. All participants were pleased and expressed approval of both functionalities. In fact, quite a few participants wanted to go even further and enforce the tagging. One participant suggested to use git hooks so that a developer would not be able to commit unless they have tagged all their changes. Another suggested to implement a static analysis similar to linters so that the IDE would not allow the code to build until tagging was completed.

One participant raised the interesting concern that, given the option of providing a file and a class level tag, they feared that when faced with reminders, developers might rush to link their changes to those higher levels in order to quickly get the task out of their way. They added that this would then affect the accuracy of the traceability applications and uses offered by the tool.

Dashboard View:

Participants' feedback was elicited with regard to having the design items (i.e., dashboard content) integrated into the IDE and to have the cards made readily accessible on the spot. Almost all participants expressed approval of the approach. One participant expressed: "…I like that a lot… I mean that just makes life a lot simpler.", while another elaborated: "… I do like the idea of minimizing context switching… gives you a little bit more flexibility. So nice about it". Nonetheless, almost all participants also recommended a form of grouping to reduce the number of cards displayed and to reduce scrolling. Many quickly drew attention to grouping by epic stories or similar. Interestingly, this feedback seemed to reaffirm our design decision around building the Dashboard Tree Providers to be faithful to the original hierarchy and structure of a user board. The Dashboard View implemented a dynamic approach to fetching and representing items and it was argued that it was important to respect the original structure so as not disrupt the view that users are familiar with on their original board (see section 6.4.2.2). When loading the iTrust dataset, a very simple structure was put in place on a Trello board. Many user stories/cards were loaded across a few lists, hence resulting in some having large numbers of cards (e.g., one had 40 cards). During the demonstration, AgileInsight simply reflected this structure, which led participants to assume that such a representation was static and applicable to all boards. It was clarified to participants that the dashboard view is actually dynamic and designed to reflect whatever structure was found on their boards.

Nonetheless, this dynamic mechanism is admittedly not yet well-tested and could definitely benefit from further improvement. The finding remains important, however, and is seen to back the original design objective. It would be interesting to see further tests run on actual industry boards, particularly when the Jira Viewer is implemented.

Code Visualisation & Visual Code Impact Reveal:

The evaluation of the visualisation aspect focused on two key areas—benefits of the visualisation's core capabilities, such as code understanding and exposing code structure, and benefits of its applied uses in the area of traceability applications.

Diverse Applications. The reactions and feedback received on the potential value and uses of the tool's visualisation capabilities were widely interesting and unexpectedly diverse. One point all participants seemed to certainly agree on was that they saw the visualisation to be 'very' useful. However, each had a different inclination to how and to whom it would be mostly useful. In fact, a few of the participants spent a good amount of time keenly elaborating on the different uses and applications in which they thought the visualisation would help them (or their colleagues). Of special interest is the apparent

disagreement observed in some of the responses. Upon first glimpses of the visualisation, some participants immediately reacted that it was especially good for non-developers, while others saw it as exceptionally useful for developers, and elaborated on specific scenarios. Yet, some expressed its value for business people who might not be familiar with code and technicalities, while others described it as very unsuitable for business people because they did not understand code. Similar thoughts were also expressed around Scrum Masters and Product Owners. In fact, while a manager insisted that Product Owners and Scrum Masters would not understand the visualisation, a participant who had years of experience as a Product Owner and in testing, expressed that the view was of particular appeal and interest to them, enabling them to see an overview of progress: "… a Scrum Master… he can say this much work is completed and whatever is pending that needs to be taking... I think the Product Owner is happy to see the visual impact because they will not be able to understand… code and all these stuff, which is kind of maybe boring for them as well. So this is a good representation for the work which is completed".

Another interesting case was a participant, who upon seeing the view, described it as "awesome" but added immediately that it was not for them. They asserted its potential value to "other people" , but described themselves to most likely use it to "play around" initially, before resorting to the tool's traceability lists and hierarchies views. Soon after however, as the participant kept freely navigating the potential cases, they related to personal scenarios and described the visualisation as useful for performance review, for retrospective meetings and discussions, and as particularly helpful for story point estimates.

In fact, at least two participants described the visualisation as helpful for estimating effort and story points, as it enabled them to assess the complexity of the modules that they would be working with. The same participant who first stated that it was not for them, elaborated that they could visually explain to their manager why some work required more time.

Areas of Most Perceived Value. Exposing the code structure and illuminating aspects of its health, and revealing the distribution (or locality) of features, their implementation, and planned changes were among the mostly widely cited applications. After demonstrating the visual code impact of a few iTrust features, a participant commented: "I suppose this view can have a usefulness to see… this particular change looks like is touched all these different places, versus changes that might be more localised… it's more of a way to get some insight… every time that we make a change. Do we touch localized files or we *are touching everything every time. So it *is as a way to get… a quick insight to say do we have the right level of abstraction or the right architect? You know split of work kind of thing, right? That's what I will perhaps use it for."

On the code health aspect, a participant commented that it could “in a form *give a wider perspective in terms of looking at the health of the codebase and kind of dependencies and things like that.” On code understanding, one participant explained: “I think from a developer's point of view, this would actually be very useful… to understand how those parts are, and what each one means”. With regard to examining features’ locality, a participant described the tool to allow them to “… visualize and see the changes that are happening in the code base”, and described it as offering ‘succinct’ information: “… I'm not aware of any other tools that you can… generate this. So succinctly you know… out of the box. I think it's as a lot of benefit using something like this.”. Lastly, a participant elaborated on how the tool could enable them to assess the impact of their changes on the codebase: “…You'll be able to see like how widely… the number of things that have been affected by your particular *implementation… sort of alert you to the fact that I shouldn't really have been touching there… I need to refactor or kind of put some sort of an adapter… just in shortly and visually, that's a huge help…. as humans we sort of work better with pictures, so that's definitely be helpful”.

Below collection of topics represents a summary of the various uses and applications of the visualisation as perceived and verbally expressed by experts during their sessions. Table 8.7 provide selected remarks and quotations where such uses were cited, and offers an opportunity to examine experts’ original lines of thought.

Offer raw sense of architecture	Verify impact of change before committing (identify if ‘touched’ unnecessary modules/areas and if introduced change is well-contained)
Helpful for code review to spot problems potential refactoring	Inspect Health of Codebase
Effort and Story Point Estimation	Saves Mental Effort
Onboarding new team members	Helpful for Scrum Master and Product Owner to inspect Progress and Visual Impact
Encourage Discussion among Developers and Team	For Sprint Retrospectives and Reviews, to inspect impact, conduct analysis, and demonstrate progress
Particularly Helpful for Developers	To present to Higher Management
Helpful for Architects and System Designers	Performance Reviews, to demonstrate work completed to team and manager

For new testers and test engineers to get started with learning the codebase	Assess locality of planned Change

Discussion. The above diverse impressions and personal penchants regarding the potential uses and application of the visualisation attracts special attention. In one aspect, it highlights the 'unexplored' nature of the technology. Despite at least two decades of research, practitioners in industry and the development community had not had the opportunity to explore, to navigate, and to use software visualisation tools. It certainly highlights the keen interest of users to try and potentially employ such applications. While all the expressions and feedback above do not go beyond the realm of 'qualitative' assessment, it nonetheless affirms the existing potential, the existing interest, and the lack of offerings of this sort of technology to users in the outside world. At this very early stage, it is unwise to claim advantages and benefits to any of the cited applications above without more rigorous empirical measurements. However, the most important finding observed here is the experts' keen interest in and their perceived value of the technology. We believe that the fact AgileInsight is offering the technology in a way that the experts saw as very accessible to them had an important influence in this result.

Visualise Design Items. Some very interesting suggestions were also received regarding the visualisation of DIs. A participant who expressed that the visualisation was particularly suited for developers, but not business people or Product Owners and Scrum Masters, contemplated a different style of visualisation for those users. They suggested that these users are mostly interested in 'terms of how many', such as how many user stories, cards, and tests completed. Or on how much money was spent on different features, where effort needed to be focused more, and where resources needed to be place. They suggested instead to find a way to visualise the 'design items' themselves, for those users, rather than the physical code structure. Interestingly, such perspectives were indeed contemplated during the earlier stage of the research, but planned effort in this direction had to be dropped due to scope and time considerations.

Dynamic Mapping/Scaling. Another interesting remark was related to the mapping and buildings' scaling. A participant noticed that, in the visualisation scene, a few buildings were 'too small' while very few others were relatively large. They then recommended to have a scaling mechanism so there was a control on the proportion of the sizes so those particular objects are kept within reasonable representation. They recounted that, at their workplace, quite a few classes are exceptionally large, recalling 'god classes', and that they would 'skew' the visualisation to produce very disproportionate results. The participant was then showcased the effects of the multiple dynamic mapping functions implemented by AgileInsight. The default view was set to a 'normalised' mapping for both primary and

secondary containers. After switching to the linear mapping, the scene became extremely disproportionate. They then expressed that they saw value in both mappings, but highly preferred the default normalised view.

Agnostic Nature. Lastly, a participant who was involved in the early phases of the evaluation was particularly interested in the visualisation but wondered if it would work for their codebase. They described that they use 'a lot of' Go language and JavaScript, in a mostly functional programming paradigm, with rare uses of classes, and were wondering if the tool could visualise those codebases. Given that the dataset that was the subject of the demonstration (iTrust) was a Java based system, they suspected the visualisation technique to have been designed to expose Java code structure only. It was then explained that the visualisation was actually designed to work with any language regardless of paradigm, and should be able to expose any structure. They were then alerted to the fact that the visualisation view they were observing had the jQuery module represented (a JavaScript library), along with the Webroot folder of iTrust, which included Jsp/HTML files.

Bi-Directional Traceability support in IDE

Aside from the visualisation applications demonstrated, participants expressed very keen interest in and perceived value of the traceability functionalities offered. This included the in-editor or in-context features, along with those accessible from the dashboard view. Similar to the visualisation applications, a few participants offered extended elaborations and appreciation of the traceability features. In fact, one participant cited those particular features as "the thing that will really be useful for" them as it allowed them to better navigate and trace the history and their team's activity. Comparing their existing practice to what the tool offered, they noted: "… you have to search the story and then you have to trace which person who did that thing…. So if it's all in one tool, that would be great".

Also in this regard, a participant ardently lamented the difficulty they were facing to trace not just others' work, but their own work. They described how the pace of work makes it very easy for developers to lose track of their changes: "… it doesn't even need to be two months later. It can be two days later and I still will have forgotten. Hard to follow even one's own work… on large projects you take a story… you finished story, you do your unit testing and then you pass it on to the testers and you'd grab the next story.. you know a couple of days… just a couple of days ago nothing anymore until you go back and you are orienting yourself with what's going on". Then they commented on how the demonstrated features could simplify and cut short their efforts: "… and I think that's a really, really good that that you would be able to immediately see… OK, where am I actually going to be looking? I don't have to sift through all my code again to try and work out what I'm actually trying to look for."

Another participant described how the tool would be helpful for them to pick up where they have left off on their own work, where they work in small increments to fulfill features: "… you know if you're working on a feature where you have a sequence of commits so… working… in multiple small increments, right, it means that you can very easily go back and say okay what were the previous changes that are made for this particular ticket. So you can kind of… if you go home for the day and then start up again in the morning on the same feature… You can then just go and click on that ticket and find out where all the code changes that are made, and the files that I was working in yesterday, right… it gives you that very like immediate like context of where you are". They then added that it could also help when multiple developers were working on the same feature, allowing each to see the others' changes immediately: "… and I imagine could also be useful if you have a few engineers working together on a on a feature just live... as people make commits… other engineers working on the same problem can immediately see what changes you've made and where. They can access that directly, so that might be another you know good benefit as well."

Many participants noted that the traceability features would be especially useful to them around bug fixing. One participant recounted an exercise they regularly performed in their workplace to trace back bugs to their related files, and then identify those files where most bugs originated. They would then try to reduce the complexity and refactor those files. They described that the tool would allow them to potentially reduce the effort involved by directly identifying files and specific methods or functions where the bug fixing had happened. Others cited support for debugging activities.

Further examples and quotations from participants are provided in Table 8.7, displaying original wording and thoughts. The full list of quotations are then provided in Appendix IX.V for interested readers.

Handling Multiple States Or Users

An aspect that surfaced in the sessions, and which was of particular interest, was that of maintaining the application state across different users. A few participants raised concerns around how the state would be kept consistent among multiple users. This was particularly with regard to two cases; users tagging the same items simultaneously, and maintaining consistency of the untagged code items views across multiple users and machines. As expressed earlier, the design of AgileInsight has fortunately avoided such problems as it was structured and built to use no caching or hidden middleware files. As detailed throughout Chapter 6, operations are executed on-the-fly and live, so that when users are performing actions, or viewing information, it is always up to date. This includes the tracelinks. While AgileInsight displays tags to users in different contexts for their convenience, the original tracelink

datapoints are stored under the design item objects in the users' board (ideally in a hidden field). Any time information is displayed to the user, data is fetched on the spot and asynchronously from the user's dashboard system. If the design were built instead around fetching tracelink data from the repository, such timely and continuously up-to-date capability could hardly be feasible—let alone the greater complexity involved.

Overall. In closing the evaluation sessions, many participants emphasised their preference to see further automation of the tagging, and to implement a mechanism to enforce it when desired. A few participants were keen to see a Jira connection in the tool. The overwhelming theme that dominated the closings, however, was experts recounting the potential value and uses they saw in the tool. Almost all participants expressed their desire—or recommended to the student—to see the tool be made available to the outside world.

Table 8.7: Example quotations capturing participants impressions and feedback at phase three of the evaluation study. These are provided to give a feeling of experts original thought lines.

Feedback/Reaction on Tagging Mechanism	• A participant prefers the DI-Initiated method, but still see need for further automation: "I say to do it manually for every method that would probably be unusable to do it. The second way is better, but I wonder if there's a way to actually automate what you've done." • "what if I made the mistake of tagging a file? And can I change my mind in the end or something?" • "Yeah, so far looks quite unintrusive, so it's more about publishing the practice I suppose… it's good, that is straightforward"
Feedback on Post-Development (retroactive) tagging features	• "I can see this useful if you have a relatively new project, something that is, you know within reasonable amount of work to be done. Yeah, it's definitely helpful." • Participant comments on untagged list: "Yeah, yeah, that's a good idea. Well, but again it is not so good idea if it's a huge project that has been going on for years because no one wants to spend time tagging... But if your example, if it's a new each project, then yeah, that's a good idea." • "Yeah, I think code lens is probably going to take a while. Whereas the other one is better (referring to Untagged Lists)… I think what would make it really good is that you would have like a checkbox, maybe at the top if you were wanting to select all of them, which would then allow you to just skip having to go down and individually do…"
Perceived value of AgileInsight's Traceability Features	• Participant commenting on ability to pick a design item (e.g., a bug) and instantly reveal all its related code items, or vice versa, to pick a file and instantly reveal all related design items: "Yep good, yeah, I think that will be powerful. I guess I would say this is one of the most powerful. It's fairly well I will use it, if I can… so give me the file that has most of the bug related to it and then, if then in the visualisation I can find OK this this file got like 20 bugs

	related to it and then OK this is a problem file, how we can fix that file? Yeah I think yeah cool good." • Participant describes how DI-to-code traceability could support his developers who work on small increments as part of their continuous delivery scheme to be able to pick up where they were: "…you can kind of go home for the day and then start up again in the morning on the same feature. You can then just go and click on that ticket and find out okay where all the code changes made with the files that I was working in yesterday, right." • "it gives you that very like immediate like context of where you are" • Participant comments on bi-directional traceability functionality of AgileInsight: "… about if you're coming to fix a bug, that's actually really handy what you've got there, because generally when you get a bug, there's no direct correlation between the bug and the code. There's no definite… it's this method or it's in this file that you have to go searching through your code… can be for days. I mean you know on a big project it's just ridiculous."
Feedback/Reaction on Visualisation & mapping technique	• "Awesome. But for me personally I might not… I think it's good to use at first, but for me.. maybe no, I'm only gonna use it like to play around. Then afterwards I'll stick to the tree hierarchy. So that's just me. Personally, I think some people are more visual people, I guess." • "…once you get the labels… working, being able to move around and get that sort of instant *view of these individual affected components was really good too" • Asked if the visualisation help in exposing code structure, participant answered: "I think from a developer's point of view, this would actually be very useful… to understand how the how those parts are and what each one means"
Visual Code Impact Feature	• "That's good, yeah, it's like you know the information you can get out of that… This representation here is kind of good for developers and also for kind of technical leads… Another can sort of visualize and see the changes that are happening in the code base, and I'm not aware of any other tools that you can… generate this. So succinctly you know… out of the box. I think it's as a lot of benefit using something like this." • "I think the product owner is happy to see the visual impact because they will not be able to understand… this code and all these stuff, which is kind of maybe boring for them… so this is a good representation for the work which is completed, yeah" • "This is really helpful and it saves the kind of mental effort of actually… scanning through the files in a particular change that you've got to commit to see what it is that you've changed" • Participant describes potential benefit to identify feature distribution across a codebase: "… also maybe identifying problem areas in an application right, what are the features that light up this whole map, what are the things that are spread throughout the application and not very well encapsulated right that sort of stuff you can probably identify with this map room."
Suggestions to further Improve Tagging Mechanism	• Participants recommends further automation if we are to get user adoption: "I think… engineers are fundamentally… lazy about their day to day work, they want to automate things as much as possible and minimize the amount of admin work that you do for each task. So if you help engineers you know

	automate this process as far as possible. Then that that's when you're going to see adoption." • Participant react to the suggested idea of enabling users to check out a feature at beginning of session to help further automate the tagging: "Yeah I mean, I think that makes sense. You might be able to simplify even further by just saying that when the developer makes it commits at that point link everything that they did to the ticket right." • "…we probably need to come up with a way where engineers are either prompted to do it, or it just happens automatically as part of their workflow, so it doesn't depend on them always remembering to do this thing right"
Suggestions to Improve Visualisation	• Participant comments on the need to properly scale the buildings ensuring they don't grow exceedingly large—supporting our argument and approach to adopt dynamic mapping functions: "in terms of the class size… You may want to come to the surface in how you might sort of scale that… you know some of those bigger or legacy codebase out there, we've gotta some really big classes… god classes. But then you'll have some very very small ones as well. So maybe scaling could be something you might wanna make sure works" • Participant comments: "might be good to have the name of the folder there…", in reference to the lack of the labels in the visualisation
Tagging Reminder	• "I did kind of wonder, you know what happens when some people do this and some people don't. Uh, so seeing that there's a reminder there too. Uh, it's quite cool" • "Yeah, that's good. You could even… enforce that a little bit further, like you could actually fail the build like if you detect some changes in a commit that hasn't been linked… sort of forced everyone the same way… have sort of quality checks.."
General Impressions & Feedback	• "I think particularly around extending the tagging and how you might enforce that. But I think from first look that covers visualization and the ability to relate chunks and classes of code to tickets and back again. There's is really, really helpful and having that altogether succinctly… in one tool like within the IDE. It's really great. I can see a lot of time being saved there with them" • Participant comparing their existing practice versus what the tool could offer them: "We otherwise have to sort of chop, chop back and forward to update tickets and things with descriptions and things… like you know the work item tickets and also within source control too. So having this thing to be able to tie them all together is really good. I think it's going to increase productivity... so it's good here" • Participant asked early in demonstration if extension is already available for download, then commented: "…OK great, we will use definitely when available" • "My takeaway would be definitely the bug fixing because it doesn't matter how carefully you code, doesn't matter how well you think you've done your unit testing, bugs just come out of nowhere from sometimes things you didn't even realize you know… so that would be really helpful, really good. ", participant added in concluding the session, referring to the bi-directional traceability features offered by AgileInsight as mostly useful to them.

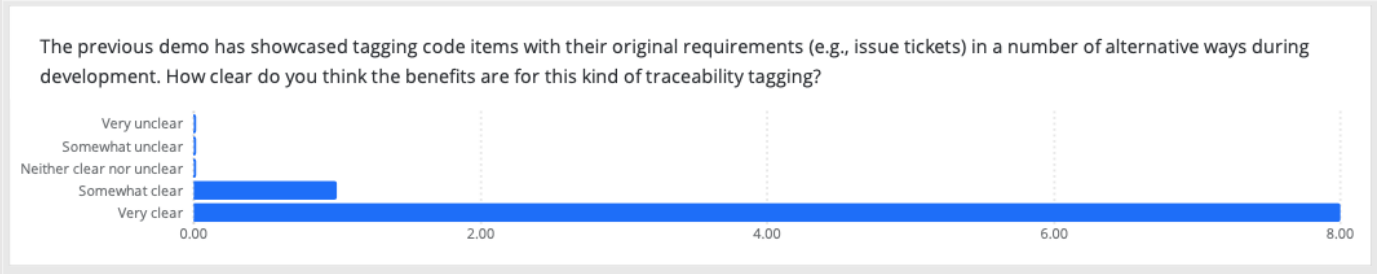
General Suggestions	• "… I think it's a cool tool it's something that you definitely have to kind of play with it a little bit to figure out how to best use it, but I can definitely see applications." • "… but no really good presentation. I love the idea. I think you've got a winner there. If you were able to get that commercialized and as an extension to… dev programs that would be awesome"

8.7.3.2 Survey Results

This section presents the results of the survey taken by the 9 experts who participated in the demonstration sessions. As explained earlier, participants completed the survey over three stages during their sessions—they were given time to privately completely the related section of the survey after each part of the demonstration was completed. The results are presented here with brief commentary, and an overall discussion is provided at the end. Participants' demographic information is not included in this discussion, however, the full survey questions and results are provided in Appendix IX.IV. Picture insets were included in the survey to give better context to the participants, which are also reproduced here as they should be helpful for the reader to understand the task at hand. Moreover, the video demos at the tool's profile page cover a good extent of these and other similar tasks[135].

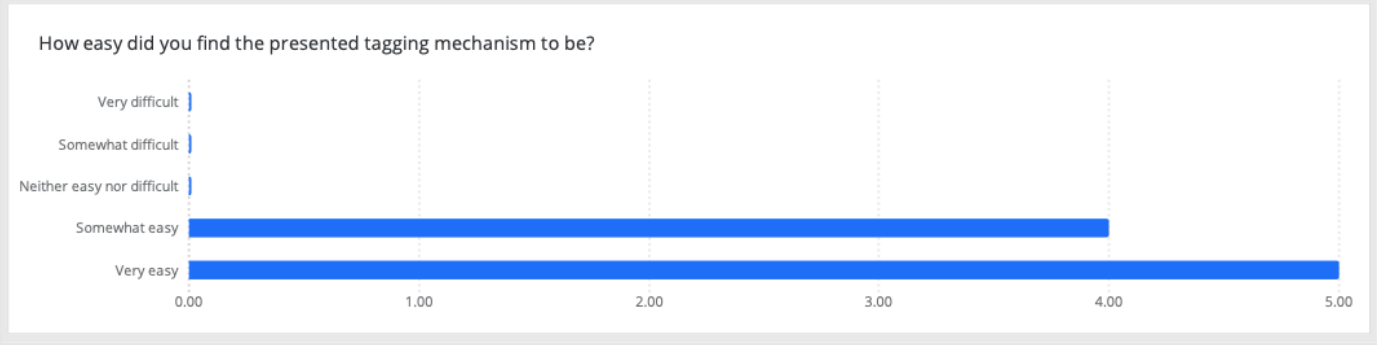
1. Experts' Perception of Tagging Benefit:

To assess the value that developers saw in tagging code items with their original requirement, they were asked how clear were the benefits to them, after a few tagging mechanisms were demonstrated. Eight participants had very clear perception of the benefits.



2. Ease of Tagging Mechanisms implemented:

With respect to the usability and ease of use of the alternative tagging mechanisms offered by AgileInsight, five participants considered it very easy, while four saw it as somewhat easy.



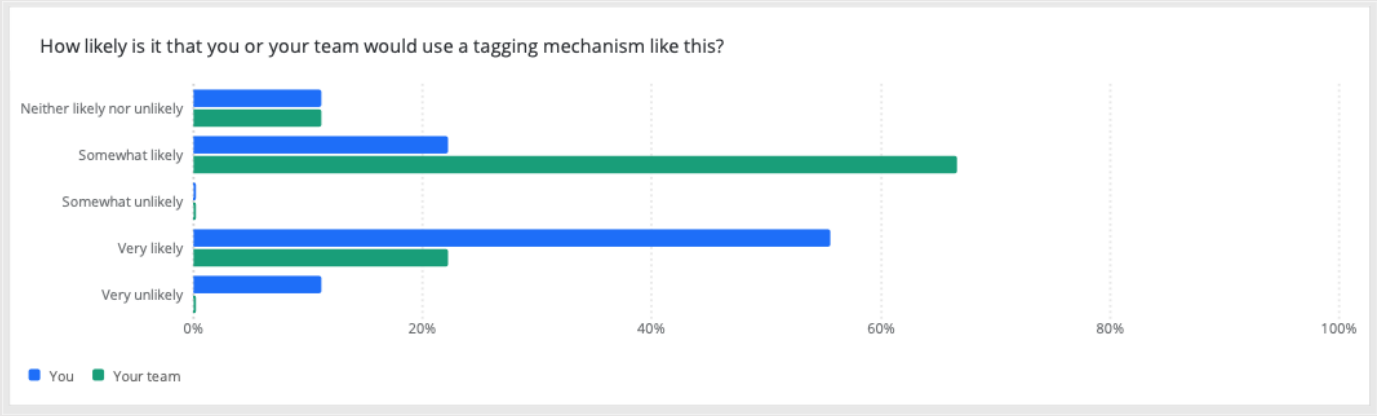
[135] AgileInsight Profile page: https://blaiski.github.io/agileInsight.page/

3. Likelihood of Using the Implemented Tagging Mechanism:

After seeking their perception of the usability of the tagging mechanisms, experts were asked how likely would they or their team use such a mechanism.

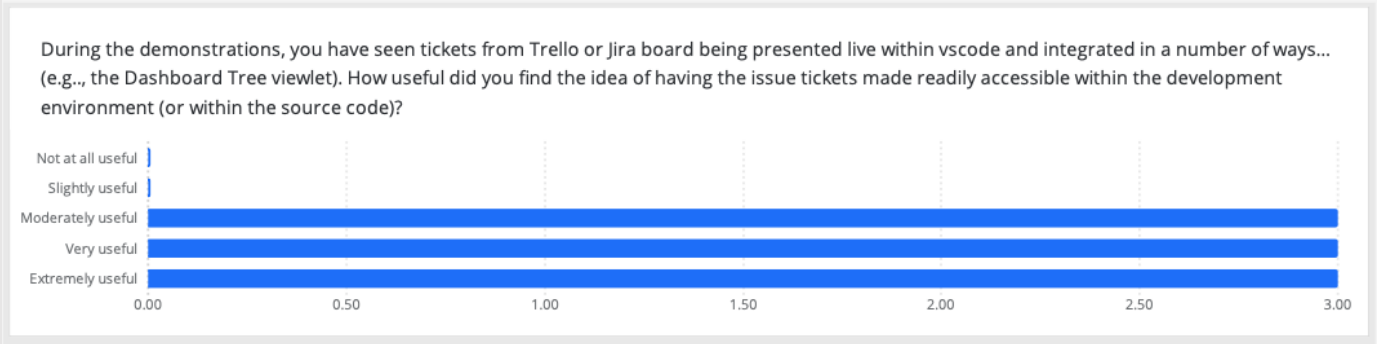
On a personal level, five of the nine participants answered that they would very likely use it, while two indicated that the chance of them using it was 'somewhat' likely. One person was neutral, while another answered that it was very unlikely that they would use it. It should be noted that participants were considering their current roles when answering the question. For example, the person who answered 'very unlikely' remarked that this was in consideration of his leading/managerial role (and they answered 'somewhat' likely for their team).

On a team level, the majority (six participants) expressed some level of uncertainty on whether their colleagues would be willing to use such a mechanism, indicating the likelihood as 'somewhat'. Two expressed confidence in choosing to answer as 'very likely', while one participant was neutral.



4. Dashboard View, and Presenting the Design Items in IDE:

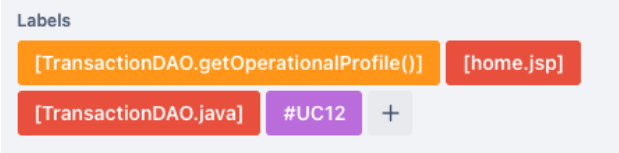
Participants were asked about their perceived level of usefulness in making issue tickets readily accessible within the IDE. This included the structured dashboard view and the inline contextual accessibility, as shown in the figure. All participants expressed that it was useful, with three labelling it as 'extremely' useful, three as 'very' useful, and the remaining three as 'moderately' useful.



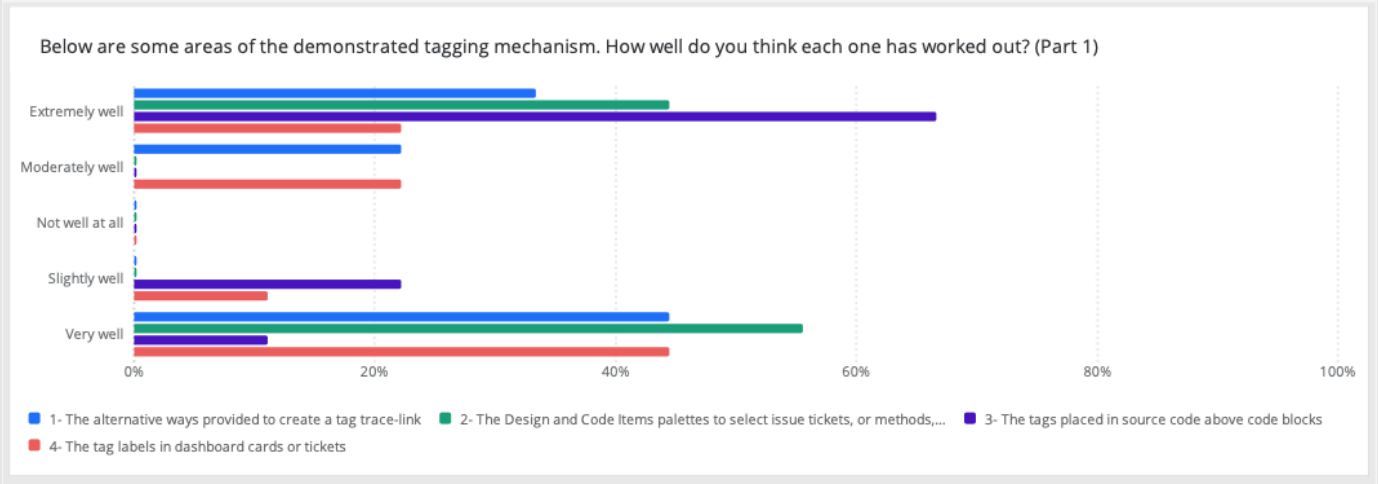
5. Aspects of the Tagging Mechanism (part 1) :

To better assess the tagging mechanism presented by AgileInsight, participants' feedback was sought about specific aspects of the process. For convenience, the question was broken into two parts, and pictures were provided where necessary. In part one, participants were asked about the following four areas:

1- The alternative ways provided to create a tag trace-link	
2- The Design and Code Items palettes to select issue tickets, or methods, etc.	
3- The tags placed in source code above code blocks	

4- The **tag labels appearing** in dashboard cards or tickets	

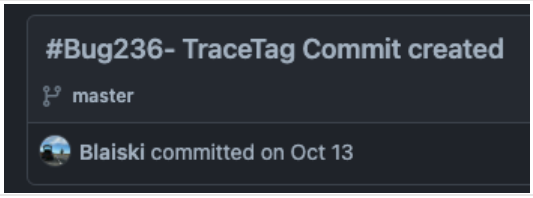
The majority of participants indicated that all four areas worked out either very well or extremely well for them. The lookup and select palette mechanism to choose an item to tag (Code or Design) stood out among the others, coming to be seen as working out 'very well' by five participants and 'extremely well' by four others. Attaching the design item tags to their related code items in source code (item 3 in the figure), followed next in the approval scale, and came to be described as working out 'extremely well' by six participants. On the other end, two participants expressed less liking to attaching the tags to their code items in the source code, and one participant expressed the same to displaying tag labels on dashboard cards, as they described both aspects to have worked out 'slightly well' only. The full response distribution is provided below.



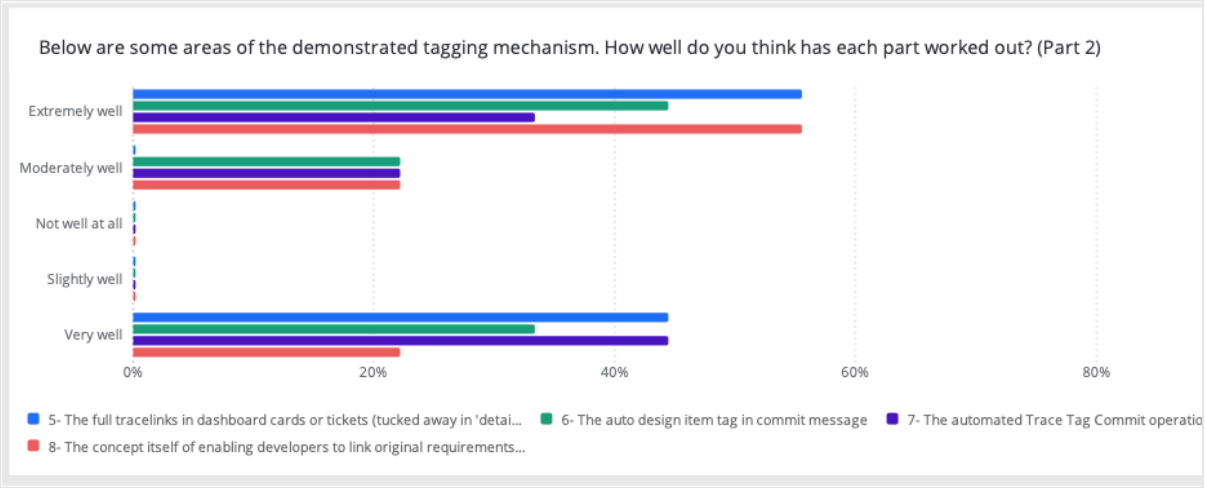
6. Aspects of the Tagging Mechanism (part 2) :

In continuation of the previous surveyed issue, participants were further asked to provide feedback on four other aspects of the tagging mechanism, as follows:

5- The **full tracelinks** in **dashboard** cards or tickets (*tucked away in 'detail' or other similar field*)	

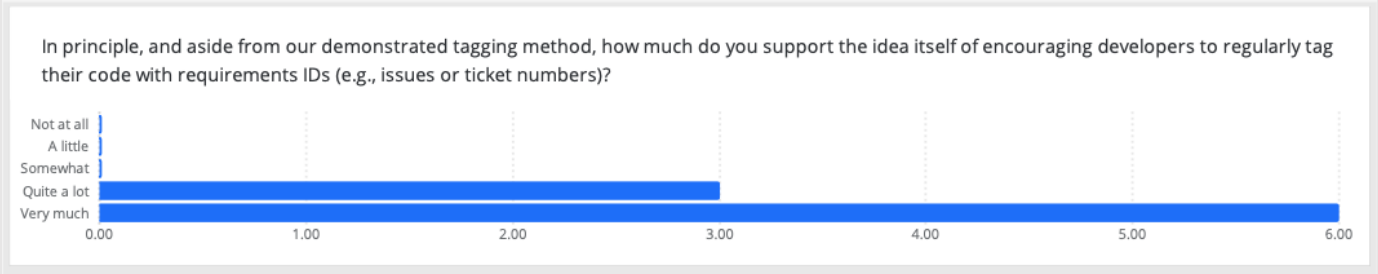
6- The auto **design item tag** in commit message	
7- The Automated **Three-legged Tracelink** operation as a whole	
8- The **concept itself** of enabling developers to link original requirements to their code implementation	

Five of the nine participants described the full tracelinks in cards, and the overall concept of linking code items with their corresponding original design, to have worked out ‘extremely well’, with the former aspect being further described as working out ‘very well’ by four other participants. The operation as a whole was described as working out ‘very well’ by four participants. On the other hand, two participants had an average liking of each of the operation, the concept, and the automatic tagging of commit messages, describing their working out as ‘moderate’ only.



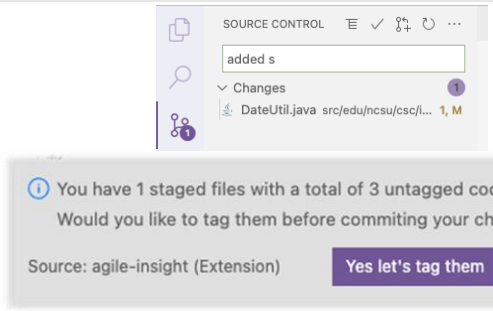
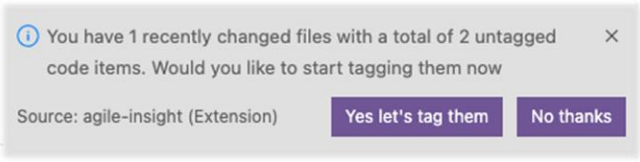
7. Support to Tagging Practice:

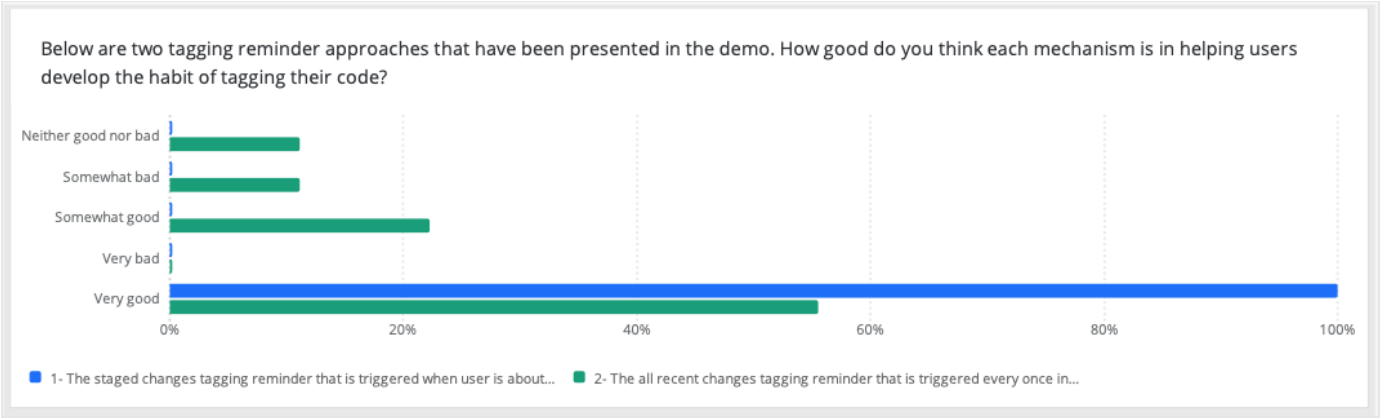
In addition to the perceived benefit of tagging presented earlier, participants were also explicitly asked if they would support the practice itself among developers. The response was unanimously strong approval between ‘quite a lot’ (3 participants) to ‘very much’ for the remaining six participants.



8. Tagging Reminders:

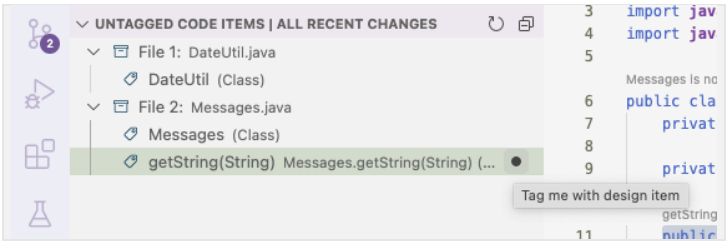
With regard to the two tagging reminders offered by AgileInsight, all nine participants expressed strong approval of the mechanism wherein the user is prompted to tag their staged changes upon attempting to commit them (triggered when they start typing a commit message), describing it as 'very good'. As for the other mechanism that is triggered periodically for recent changes, only five had strong approval of it while two found it 'somewhat' good. One participant found it 'somewhat' bad, while another was neutral.

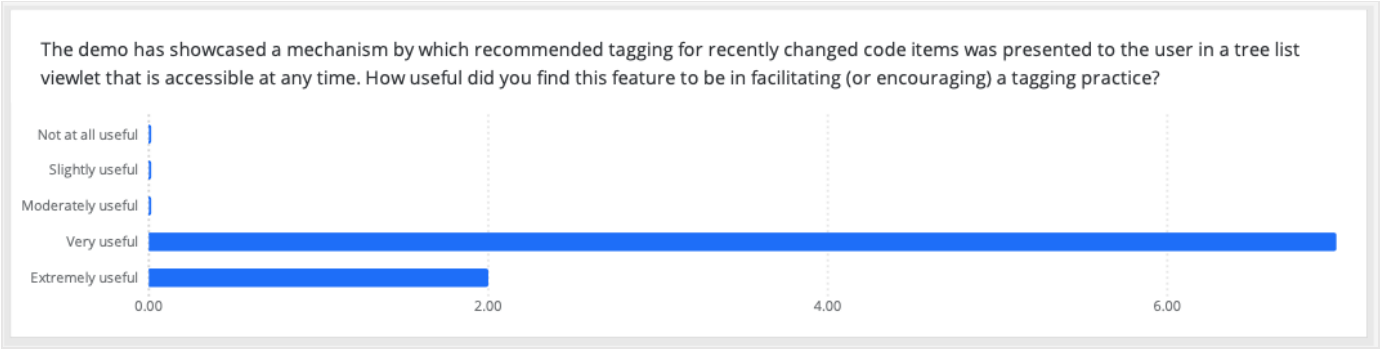
1- The **staged changes** tagging reminder that is triggered when user is **about to** commit their changes (starts typing in box).	
2- The **all recent changes** tagging reminder that is triggered every once in a while **as per configured timing** preference.	



9. Untagged Recent Changes Viewlet:

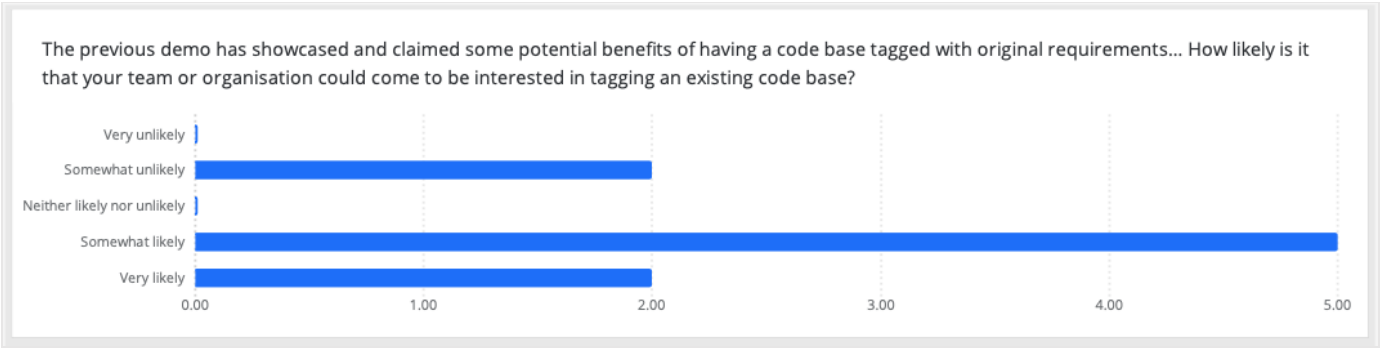
AgileInsight offered a list of all untagged recent changes (or working-tree changes in git system terminology) that is accessible to developers in a side viewlet at all times. All participants expressed strong approval of the mechanism, describing it as either 'very useful' (7 participants) or 'extremely useful' (2 participants).





10. Expressed Interest in Tagging Existing Codebase:

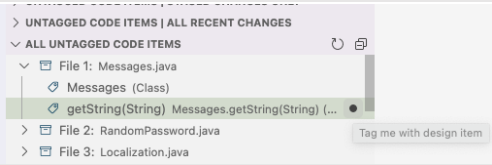
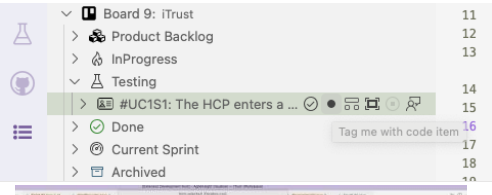
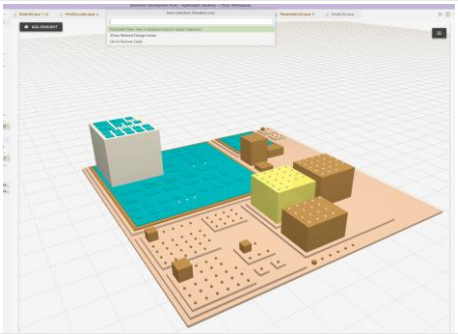
To assess the potential interest among practitioners in the idea of tagging an existing codebase, participants were asked about their perceived interest in such an activity by their team or organisation. Five thought it was 'somewhat likely', while two were confident to describe it as 'very likely'. The remaining two found it 'somewhat unlikely' that their organisation would be interested.

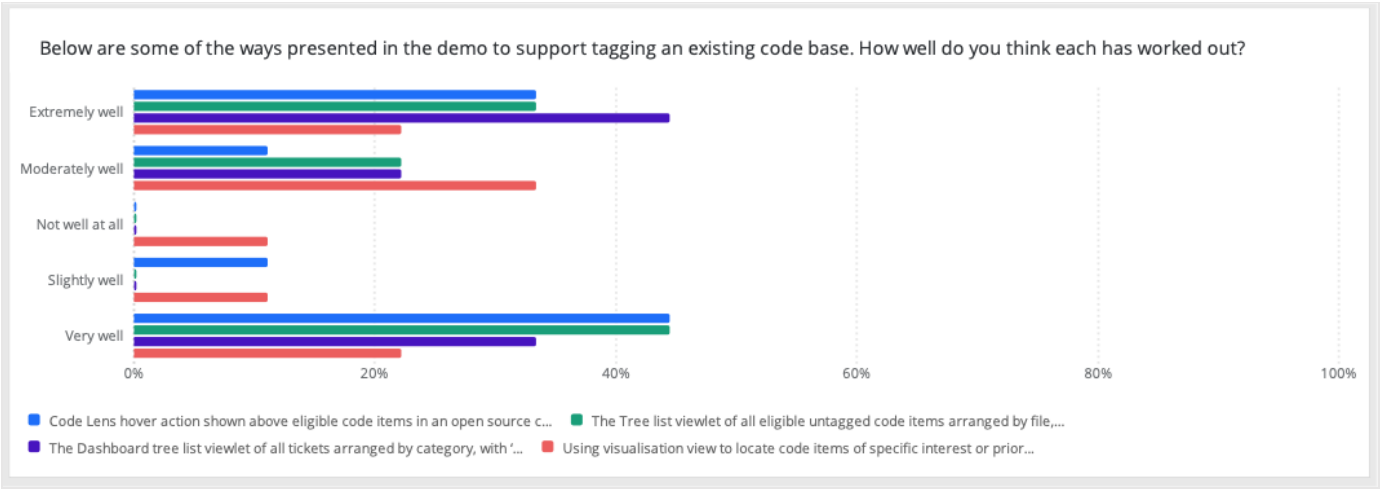


11. Offered Mechanisms for Tagging an Existing Codebase:

Participants' feedback was sought for the four methods offered to tag an already existing codebase as shown below. The first three methods were well-favoured with seven participants describing them as working out either 'extremely well' or 'very well'. Support for tagging from the visualisation scene was less decisive, with responses being spread across the band, and the 'moderately well' option receiving the highest number among them (3 responses). The chart below presents the full responses. It should be noted that a fifth method (not covered in the survey) that allowed users to identify untagged cards

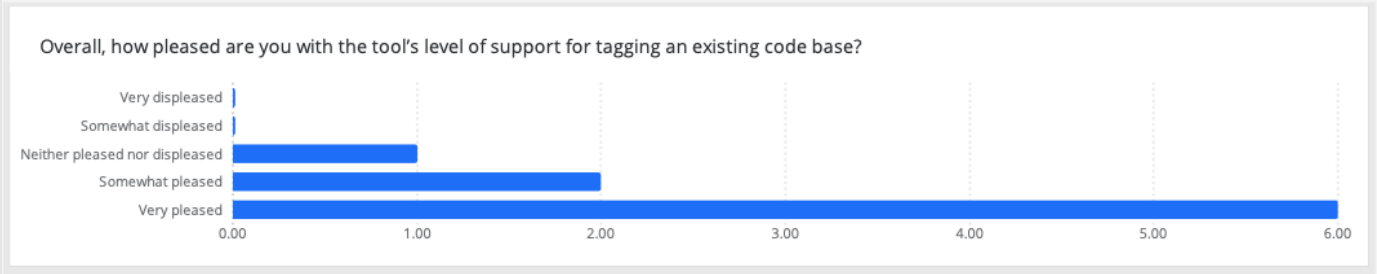
(i.e., design items) was demonstrated during the sessions and feedback on it was reported in the previous section.

Code Lens hover action shown above eligible code items in an open source code file	
The Tree list viewlet of all eligible untagged code items arranged by file, with 'Tag me' action presented on each item	
The Dashboard tree list viewlet of all tickets arranged by category, with 'Tag me' action provided on eligible cards.	
Using **visualisation** view to locate code items of specific interest or priority to tag	



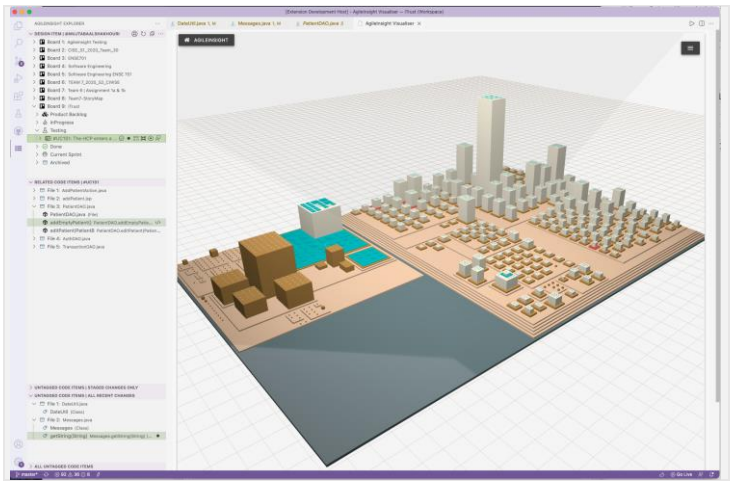
12. Overall Impression of Tagging Existing Codebase:

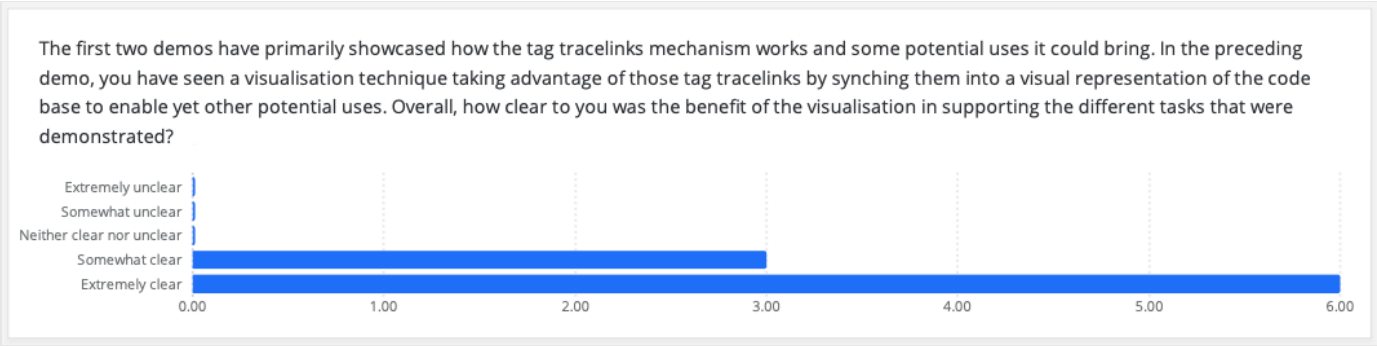
In regard to the overall satisfaction of the provided capabilities to tag an already existing codebase, six participants described themselves as 'very pleased', two as 'somewhat pleased', while one participant remained neutral.



13. Perceived Utility of the Visualisation:

Participants were asked about their perceived benefit of the visualisation as a medium to support development activities around traceability applications (in general), as demonstrated in the session. The majority (six participants) described the perceived benefit as 'extremely clear', while three described it as 'somewhat clear'.

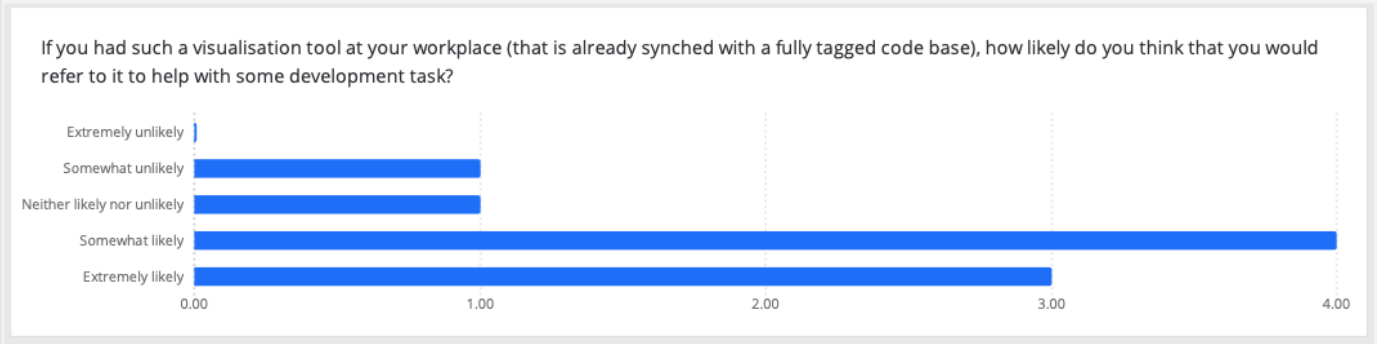




14. Personal Likelihood of Using the Visualisation (for traceability applications):

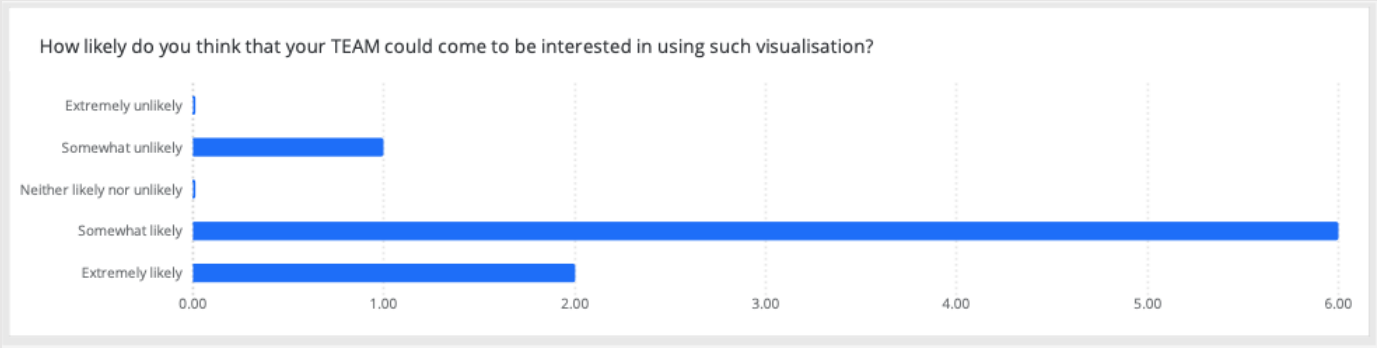
To assess the potential likelihood of developers utilising the visualisation tool to aid with their work around traceability tasks, participants were asked if they would use such a tool if it was available at their workplace. The majority of participants indicated that they would likely use it, with three

confidently labelling their use as ‘extremely likely’ and a further four describing it as ‘somewhat’ likely. One participant expressed they were ‘somewhat unlikely’ to use it, while another remained neutral.



15. Likelihood of Team using the Visualisation:

When it came to their teams, the expressed likelihood of using such a visualisation mechanism came out slightly stronger, with eight participants out of nine describing it as likely (compared to seven in the previous case). However, only two labelled it as ‘extremely’ likely this time, with the remaining six describing it as ‘somewhat likely’.



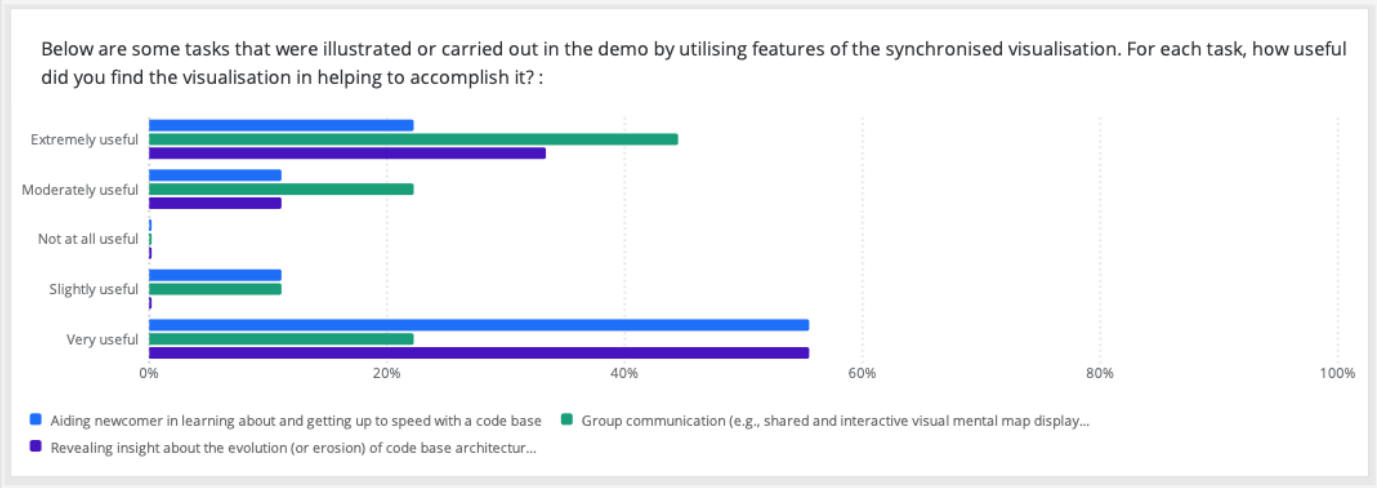
16. Visualisation Usefulness in Supporting Tasks:

To assess the perceived benefits of the visualisation, participants were explicitly asked about how useful they thought it was in supporting thee specific tasks (see below) after watching a few related demonstrations during the session.

Aiding newcomer in learning about and getting up to speed with a code base
Group communication (e.g., shared and interactive visual mental map displayed on large screen)
Revealing insight about the evolution (or erosion) of code base architecture over time

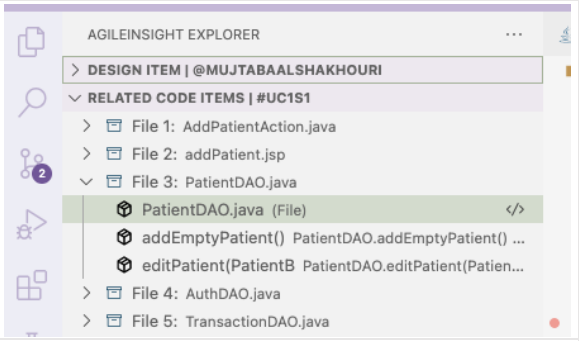
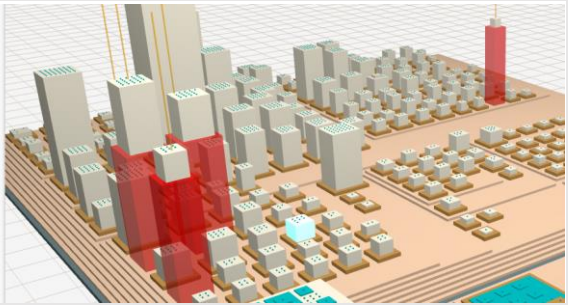
For all three tasks, the majority of participants described the visualisation as either ‘very useful’ or ‘extremely useful’ in supporting those tasks. Aiding newcomers and revealing insight about codebase

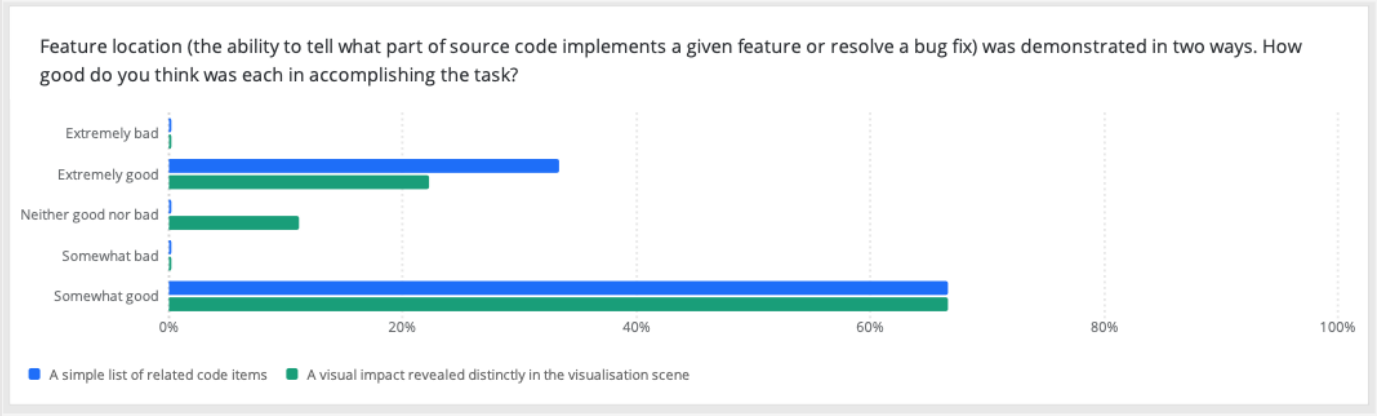
structure (including evolution/erosion over time) came out strongest, with five participants labelling them as very useful. On the other hand, group communication and aiding newcomers were described as ‘slightly useful’ by one participant each.



17. Support for Feature Location:

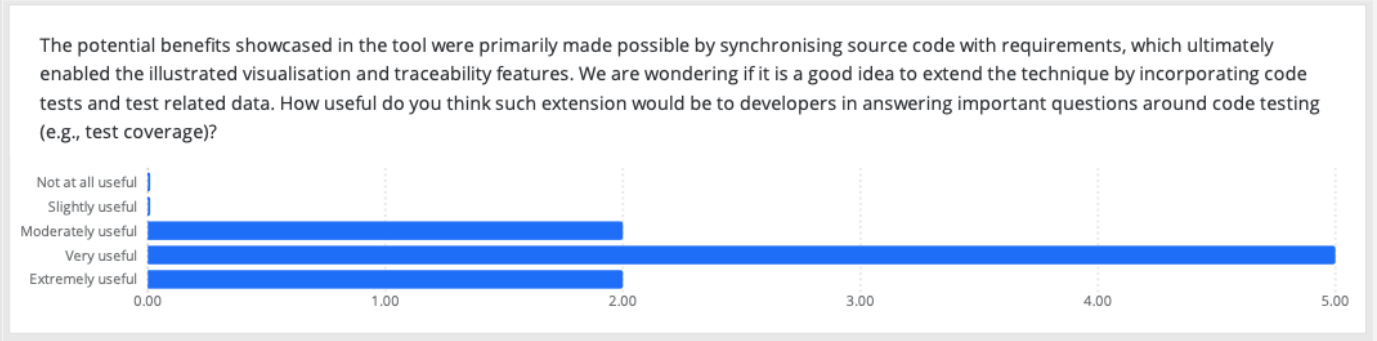
AgileInsight offered two primary mechanisms to support feature location—a hierarchical tree list and a visual impact revealed across the visualisation scene. Overall, both mechanisms were found to be good by the participants, with six describing both as ‘somewhat good’. The ‘extremely good’ description was also employed, with three such responses for the tree list view, and two for the visual impact method.

A simple list of related code items	
A visual impact revealed distinctly in the visualisation scene	



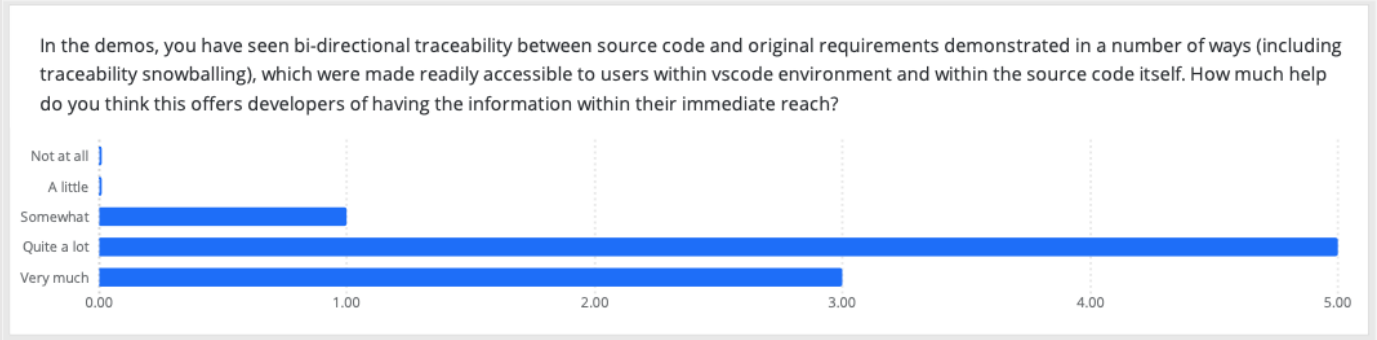
18. Potential for Code Testing Applications:

Participant feedback was sought regarding the potential expected benefit of extending the tool to support activities around code testing. Responses showed overall strong support for the proposal, with five participants describing it as likely to be 'very useful', and two each as 'extremely useful' and 'moderately useful'.



19. Bi-Directional Traceability Offered in-Context:

Participants were asked about their overall impression and the level of help offered by the various traceability functionalities made accessible within the Vscode environment, and inline within the source code itself. Almost all participants saw the functionalities as helpful, with five describing it as offering 'quite a lot' of help, and three as being of 'very much' help. One participant described it as 'somewhat' helpful.

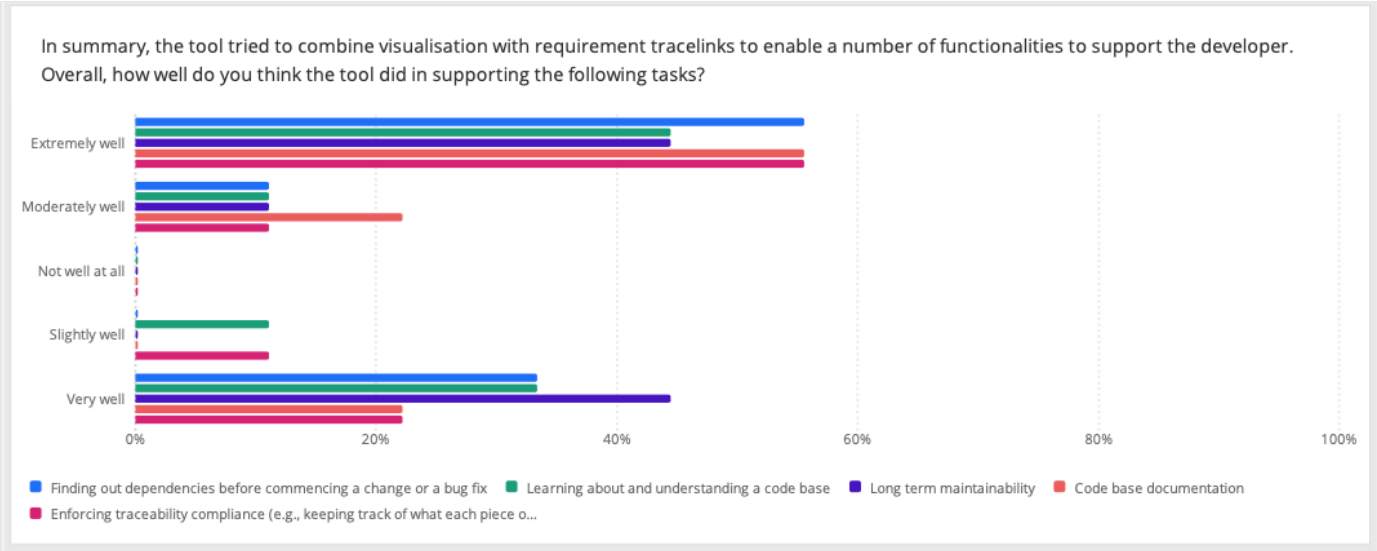


20. Offered Mechanisms for Tagging an Existing Codebase:

In concluding the survey, participants, were asked about their perceptions of the level of support that the tool had generally offered for five specific tasks—particularly as a result of combining visualisation with traceability features. The five tasks are listed below, and were elaborated during the demonstration.

Finding out dependencies before commencing a change or a bug fix
Learning about and understanding a code base
Long term maintainability
Code base documentation
Enforcing traceability compliance (e.g., keeping track of what each piece of code was exactly written for, in case of safety critical systems)

Responses displayed strong approval of the level of help offered by AgileInsight for all five tasks, with the majority describing it as being either ‘extremely’ or ‘very’ supportive. Five participants found it ‘extremely’ supportive for enforcing traceability, finding out dependencies, and for documentation. Four participants felt that it supported ‘extremely well’ developer learning about a codebase, and for long term maintainability. Support for each of the five tasks was described as ‘moderate’ by at least one participant, with documentation and learning about a codebase being also described as supporting ‘slightly’ well by one participant each.



8.7.3.3 Concluding Discussion

This section has presented the findings from the expert demonstration sessions and the results of the survey conducted in parallel. The verbal feedback from experts was analysed to identify key areas and themes expressed or raised. Those areas were then presented with a summary of participants' thoughts, concerns, suggestions, and impressions, and were supported by selected quotations. A sample selection of participants' quotations and remarks was then presented in a structure parallel to that reflected in the discussion—with the extended version appearing in Appendix IX.V. The survey sought to capture a quantitative picture of the usability and utility of the tool (and its underlying concept) as perceived by the experts. Results around each area covered by the survey were then summarised and presented with brief discussions.

A number of points are worthy of emphasis at this point. In terms of the overarching theme found in the results, it generally points towards a strong perception of utility and value, but also to interest in the tool and its concept. Nonetheless, there was also an expression of uncertainty by a small number of participants, which must not be neglected, and which is more visible in some areas than others (for example, tagging existing code). There is also a strong desire to improve certain areas of functionalities, which was better captured by the verbal discussions. Of particular interest is the fact that no single aspect or functionality received explicit disapproval by the participants—involving the developers in the design process since the research's early stage is certainly a factor to thank for this. Put very briefly, the overall results seem to indicate keen interest by the practitioners, accompanied by clear expressions of utility and value. The keen recommendations, and suggestions to improve certain areas, are also seen to reflect the interest of the developers to see such tool made readily accessible in the outside world—a desire that was explicitly expressed in the verbal discussions.

In regard to the nature of the evaluation and where it stands on the scale of rigour, both forms of assessment conducted do not go beyond a qualitative evaluation exercise. Caution is thus necessary around any attempt to generalise the findings, or to claim asserted benefits and advantages. As indicated earlier, this evaluation activity is rather seen as an attempt to assess the utility, the potential, and the practical usability of the research tool as perceived by experts from industry. This is in line with the research objective to bring the visualisation technology closer to the developer by putting it into applied use, and to present it in an accessible medium—the IDE. Arguments were also presented and supported by similar lines of thought in the literature for the suitability of this kind of evaluation for design science research, particularly in the early cycles of the research and development activities. In summary, the level of evaluation conducted is considered to have presented sufficient grounds to

establish the potential and utility of the research tool, and most importantly, to capture a reflection of the interest in the tool by practitioners in the outside world.

Threats to validity. A number of elements that could have potentially influenced the results must be brought to the surface at this point. There are at least three key areas that pose threats to the study's validity when considering the outcomes of the evaluation activity.

First, the sample size of nine participants is admittedly small. However, this size of sample is not uncommon in detailed qualitative evaluations, especially those based on one-to-one interview or demonstration sessions. Indeed, many studies featuring similar exercises report a comparable—and sometimes smaller—sample sizes (e.g., Heidmann et al. (2020); Kritzinger et al. (2019); Walny et al. (2018); Abuthawabeh et al. (2013)). More importantly, analysis of the verbal feedback seemed to point towards a saturation level around most areas, as the feedback started to repeat and converge (see discussion in section 8.7.3.1).

A second threat to validity is the potential influence imposed inadvertently on participants by the researcher through unintentional verbal cues and leads. Such subconscious cues are well-known to potentially affect the response and behaviour of participants, and are an unavoidable element in almost any qualitative evaluation that involves verbal or written interaction. Both the survey and the demonstration sessions are thus subject to this threat. In an attempt to reduce the impact of this threat, the sessions were scripted in advance and carefully reviewed to avoid such cues. The pilot run was also helpful in further refining the spoken script and the tone of the questions and conversation starters posed. Strict adherence to the script during an actual run is admittedly hard to achieve though, and total elimination of verbal or visual cues is thus practically infeasible. Similar care and review cycles were also expended to the survey questions to try to remove potential cues or leading questions. Despite all efforts, the participants themselves are also subject to the known acquiescence bias, where they consciously or subconsciously act amenably. However, the survey was run electronically with participants informed that their answers will only be reviewed after completion of all sessions—hence, reducing the influence of acquiescence. As discussed earlier in this section, item-specific Likert-style answers were also adopted in the survey in an attempt to reduce the effect of this threat.

A third and last remark to make is probably around the coverage of the visualisation's core aspects and benefits. Both the demonstration scenarios and the survey seemed to have skewed toward a larger focus on the traceability features of the tool as compared to the core benefits and potential of the visualisation itself. During the actual run of the sessions, this was realised early on and increased attention was given to the visualisation potential and its core benefits. This could be visibly seen in the

good number of verbal feedback comments received around this area (see Table 8.7 and Appendix IX.V). Unfortunately, the survey does appear to have less adequately covered the core benefits of the visualisation itself. While unintended, this was probably caused by the keen focus on evaluating the applied uses of the visualisation, which in this case was around traceability applications. Nonetheless, the key findings reported from the verbal discussions are considered to have captured sufficient feedback around the visualisation's core aspects. This area is also well-represented and supported by numerous earlier research studies, as elaborated in the previous chapters.

8.8 Conclusion

This chapter has presented three evaluation phases that were conducted in this research in order to assess and verify the design and development work undertaken. The three phases were carried out throughout the research activity since its early stages, forming a continuous feedback loop of research and development, assessment, and informing exercises—in close parallel to Hevner's methodology for Design Research (see Chapter 3). The evaluation focused particularly on accommodating the end user and their application environment into the research and development process.

The third phase of the evaluation involved 9 experts from industry, participating in a carefully designed set of demonstration-based sessions. The sessions consisted of scenarios and tasks aimed to assess the potential benefits and the usability of the tool, and were run using a real-world dataset that was carefully prepared and loaded. The sessions resulted in valuable feedback, insights, and suggestions, which were analysed, summarised and presented. Results of a user survey that was completed by the participants throughout their sessions were then presented with brief discussion, adding further support and insights to the earlier results. Also as part of the evaluation activities, a brief case study of the iTrust dataset was presented earlier to demonstrate the potential benefits and applications resulting from synchronising the original design intents with the source code. Similarly, a visual multi-language showcase involving 15 open-source systems was presented earlier, focusing specifically on the potential of the tool in exposing the complexity and the structure of source code, opening the possibility to a range of insights that could be learned about the code when using the visualisation. The showcase also served to demonstrate the language-agnostic nature of the research tool.

In summary, the three phases of evaluation allowed for close collaboration with intended end users from the start, and this has informed and guided the research in significant ways. The results and outcome of the work are hence a direct result of this close collaboration and of following Hevner's research model that guided this activity. Most importantly, the last phase has resulted in valuable insights that could yet seed future work.

Chapter 9
SUMMARY, CONTRIBUTIONS & FUTURE WORK

9.1 Summary

This work started with the motivation to bring the benefits of software visualisation technology closer to the real-world, in the form of practitioners in the development community. It saw particular potential in the concept of synchronizing software design and implementation artefacts to support outstanding software engineering issues. These issues were examined across a number of problem domains in the software engineering literature to inform and guide the research. To present a practical and operational approach, it was imperative to examine the current state of the art in the technological infrastructure, as well as to understand the targeted environment and practitioners' ways of working. The research approached the problem through a design science methodology with three cycles of feedback-design iterations that featured expert users as key players in informing and evaluating the research.

The research results point to benefits arising from practical applications in the realm of software traceability, and in supporting key development activities, including primarily: feature location, change impact analysis, and code refactoring. The research serves to show that such activities can benefit specifically from the use of modern metaphorical software visualisation. It further serves to show that software visualisation applications can be made practical and accessible to practitioners when well-integrated into their everyday toolsets and workflows.

On a paradigmatic level, this work demonstrates that incorporating the end user in the design and development cycle of research provides unique advantages, from inspiring and informing new ideas to an increased chance for achieving close relevance and connection to the intended users' actual needs and their working context. It also highlights the value of undertaking research based on recognized potential and well-justified new opportunities, as opposed to 'just' solving problems, as this expands the scope of a research workspace and encourages new innovations.

Lastly, this work saw considerable investment in exploring the space of technology stacks available in the public domain, used in industry by the practicing community—and to a lesser extent in exploring relevant practices in those domains. In doing so, it recognises the importance of including this landscape in academic research efforts, alongside the research literature.

9.2 Contributions

This work has greatly benefited from the efforts of prior research in the field of software visualisation, as well as from the work of researchers across other key domains including software traceability, requirements engineering, and information visualisation. It has built on and was inspired by many earlier works. Its contributions are thus perceived as an aggregation of and as a natural culmination of its predecessor efforts.

The key contributions of this research are identified as follows:

- A Simple Generalised Model of Software Artefacts: this work introduced a very simple, yet practical, concept to capturing software design and implementation artefacts across multi-level hierarchies in a flexible and unconstrained manner. The concept uses only a unique identifier and an optional parent/child pointer. It allows the user to capture, represent, and work with software artefacts in real-time, with minimal knowledge required about their nature or content. The concept can be implemented as a runtime ephemeral model, without requiring persistence in intermediate storage files or databases.

- Implementation of Agnostic Structural Parsing: the above concept was used to describe a mechanism and a detailed implementation for *structurally* parsing source code artefacts and design artefacts, in an agnostic and real-time manner. The mechanism was implemented in a popular development environment and demonstrated the ability to retrieve and work with those artefacts from their sources in real-time, and with no intermediaries. It demonstrated the potential of the generalised model to underpin the development of certain software tools that are not tied to a particular programming language or a specific development practice.

- Implementation of Agnostic Metaphorical Visualisation: the research leveraged the above capabilities to generate advanced metaphorical visualisations of software artefacts in an agnostic manner. It further implemented the approach to produce language-agnostic visualisations based on the 3D city metaphor.

- Live Agile Dashboard Integration: the above capabilities were also utilised to implement live integration of agile software development dashboards into a popular development environment, enabling users to seamlessly access their design artefacts within their development workspace.

- Execution to Address a Specific Application Context: the above benefits were then applied to support specific real-world engineering tasks faced by developers. For example, a developer can readily identify the code artefacts that they need to potentially modify to fulfil a particular change.

- Artefact Traceability for Agile Development: the work introduced a mechanism for capturing traceability links between design and code artefacts in real-time during development. The mechanism is semi-automatic in that it needs active input from the user. This input is minimal, however, with potential for full automation, which should appeal to agile practitioners. A fully automated operation *is* implemented to capture and record the tracelinks for individual source code artefacts, the versioning repository, and the agile management dashboard. The mechanism has the potential to pre-emptively prevent the loss of valuable traceability knowledge, and thus the need to rediscover it at later stages.

- Accessible to Practitioners: the above approaches and techniques were implemented in a popular environment that is widely accessible to the development community and software practitioners. We collaborated with expert users to ensure that the work addressed their needs, and to better integrate the introduced functionalities into their daily workflow and practices. In doing so, it should make software visualisation technology and its benefits more accessible to the development community.

- Expert User Evaluation: the research presented an evaluation process that spanned three phases of demonstration-based interviews with experts from industry, integrating their feedback iteratively into the design and development cycle. The last phase included a set of elaborately-designed usage scenarios, the use of a real-word dataset, and a feedback questionnaire. Detailed evaluation results were presented for each of the three phases, and showed largely positive results, with a number of users expressing their desire to use the tool.

- Showcase of 15 Open-source Systems: the work presented a visualisation showcase of 15 open-source systems implemented in multiple languages, illustrating its capability of language-agnostic software visualisation. Brief commentary highlights were also presented to demonstrate the advantages gained in exposing software structure across multiple languages, including within the same project. Detailed performance benchmarks were also presented.

9.3 Shortcomings & Challenges

Shortcomings. At this point of the research journey it is important to reflect on the work to identify things that could have been done better or differently. As the research progressed, and particularly during its final stages when the different components started to come together, revealing the big picture, a number of shortcomings became visible. Some of these shortcomings were inadvertent, while others were compromises made due to necessary limits on the research scope and other constraints. This section highlights some of these shortcomings.

AgileInsight Implementation. There are a number of features that were initially planned to be developed into the research tool in order to offer more complete user functionality during evaluation. This is especially true for some usability and interactivity features of the visualisation part of the tool. For example, some key features such as labelling of code artefacts and searching functionality were not implemented. Other experimental features were also planned to be incorporated to test their impact on a user's orientation inside the 3D world—examples include user drop-pins, special border and direction techniques, and custom user tagging of rendered buildings.

The current implementation also does not adequately address modification actions with regard to updating or deleting tags. Updating tags is only partially implemented, while deleting tags is not yet supported and the feature is purposely kept disabled. This does not, however, reflect a shortcoming in the approach itself—as removing or changing a tag has no effect on its working mechanism—but was solely dropped due to time and scope constraints.

Yet another important shortcoming involves an intricacy around the automatic commit action, which was also picked up by one expert user participant. Currently, if a file involves some changes unrelated to the code artefact being tagged, those changes would still be committed along with the modified code artefact. This was intentionally relaxed at this stage due to scope limitation. This could be fixed by utilising the already implemented feature of detecting the exact code artefact changed—which was used in implementing the reminder system—along with implementing a partial staging capability so that only the concerned code artefact is committed, and not other changes within the file. Nonetheless, this issue does not affect the on-commit tagging action, as in this case the developer would have staged the changes themselves, and thus other non-staged changes are correctly excluded from the operation.

Other shortcomings with respect to AgileInsight include a few usability features associated with selecting the recommended code artefacts for tagging. For example, when the user accepts the tagging action in the tagging reminder prompt, they are taken to the Untagged Items Viewlet where they

currently need to tag each item individually. A more practical implementation would need to include a multi-select capability and a "tag all" option. Admittedly though, implementation of the multi-select option was not possible due to limitations of Vscode's present APIs as of the time of development (unless a custom Webview viewlet was purposely developed). Similar other features that could have improved the user experience were also dropped –in some cases due to scope limitation, while in others they were necessarily delayed until Vscode APIs introduce the required flexibility.

Evaluation. Another set of shortcomings arise around the evaluation process. While substantial effort was expended to select and design the user tasks and scenarios, the process could have been more robust if the goals of the evaluation were explicitly and formally defined. This is, in fact, discussed and recommended in the Merino et al. work on software visualisation evaluation that was referred to earlier. The selection of user tasks and scenarios has been guided, however, by relevant research works as discussed earlier, and were carefully designed to capture the key functionalities of the tool. As such, the evaluation is still considered to hold sufficient strength. Moreover, given that the evaluation is primarily qualitative in nature, explicit definition of evaluation goals does not hold significant importance as there are no formal hypotheses to be tested against.

Two other characteristics of the evaluation must also be acknowledged here. The first is that interview-based evaluations are always susceptible to the threat of unintended visual and auditory cues that could inadvertently influence participants and skew the final results. However, as indicated earlier, the interviews—especially in that last phase—were carefully designed and elaborately scripted, and then subjected to a number of reviews and rehearsals. This is perceived to have helped in reducing those undesired cues. The second is the fact that many of the participants were solicited through the social and professional networks of one of the student's supervisors, and some are alumni of the same University. This could have led to acquiescence bias as discussed in previous chapter. Lastly, while the sample of participants (nine in this case) meets a common threshold when compared with other interview-based evaluations in the literature, a larger number secured from a range of sources would have certainly offered more robustness to the results, particularly with regard to generalisability.

Challenges. The work has also faced a number of challenges, particularly around the technological infrastructure and implementation. These were discussed in detail in Chapter 6, but the two key challenges are briefly revisited here.

The most prominent challenge was around implementation of OAuth functionality in the Vscode environment, where the API and its support was experimental at the time and in a very early stage. However, overcoming the challenge was crucial to the work as it allowed the seamless real-time

integration of agile dashboards within Vscode, which is key to realising the synchronisation concept as well as to presenting the tool in a practical fashion to the participants.

A second challenge was the implementation of the live source code parsing. As described earlier, since the current LSP API was not originally designed for such a purpose, our implementation of this mechanism required a number of improvisations and rather stretched workarounds to achieve the desired result. Nonetheless, as the LSP API is progressively developed, it is highly likely that new features will allow a more direct and efficient implementation of our live file parsing mechanism.

An important aspect of the challenges that must also be highlighted here is in regard to the availability of suitable and complete datasets. As substantially elaborated on in Chapter 8 and in Appendix IV, access to real-world and industrial datasets that are in complete shape remains a significant impediment facing software engineering researchers. We had to resort to sophisticated measures to combine two datasets in order to provision tracelinks at various levels of granularity. Efforts in this regard seem to be gaining traction though, as evident by some of the higher quality datasets that are being made available more recently. This is expected to better enable future research like this to be undertaken more often and more extensively.

9.4 Future Work & Recommendations

9.4.1 Future Work

A key objective of this work was to make the benefits of software visualisation technologies more accessible to the general development community. This is not to benefit the public user alone, but is also expected to benefit the software visualisation research community. There is at present hardly any data available regarding developers' use of software visualisation tools—especially those employing advanced metaphorical techniques. As users start to develop interest in and use such tools, this should result in the generation of valuable data that can be utilised for research and advancement of the field.

Making AgileInsight Available. It is hence our aspiration to refine the AgileInsight tool and release it to the public once it is in stronger operational shape. There are a number of refinements and improvements required before that is possible. This includes addressing the shortcomings referred to above, in addition to a few other desired improvements. For example, our intention is to make the tool suitable to the various ways that practitioners work in industry, and this certainly requires the ability to customise the tool to best fit each team's preferences. This subject was reflected on earlier in the light of feedback and comments received from the expert users. Moreover, supporting different ways of working has been a core goal since the early stages of the work, and so a number of flexible approaches have indeed been accommodated to date, such as the different ways of tagging, the different ways of exposing related artefacts, and the different computational functions of visualisation mapping. An example of further flexibility could include adding control to how artefacts are tagged. For example, in the current implementation, code artefacts are tagged by default in three places: the source code itself, in the versioning control system, and in the connected dashboard. The source code artefact is tagged by inserting a comment annotation on top of the artefact declaration. The operation can be easily customised, though, for users who do not wish to have tags inserted into their source code. In this case, they could choose a 'code lens' technique which would still give them the benefit of seeing the tracelinks in the context of the source code—on demand, whenever desired. However, they would lose the benefit of the contextual design item hover popups that offer quick details and quick actions (see Figure 6.24). Another crucial improvement is completing the Jira Dashboard Provider, which is essential as the platform currently enjoys high popularity among practitioners.

Empirical Evaluation. An important future aspiration is to utilise AgileInsight to conduct further empirical work involving a control group. This is of particular interest given its novel language-independent nature, and hence its extended reach. Given the largely positive results obtained from our expert interviews, it is now of particular interest to obtain rigorous experimental results to establish if

they align or deviate from those obtained earlier. Moreover, AgileInsight brings a novel concept of synchronised visualisation and applies it to address specific software engineering tasks, and thus obtaining experimental validation for its benefits should contribute to the advancement of the discipline given the outstanding call for more rigorous experimental work.

9.4.2 Recommendations & Potential New Extensions

At the end of this research journey, we would like to express that we perceive practicality and operationality to be an important aspect that needs to be given more attention in software visualisation research in order to promote its resultant artefacts among the software practitioners' community. We believe that this research has taken one small step towards this objective. In this regard, we reflect here on a number of potential directions that we believe could be of value to future researchers, as well as to potential extensions to AgileInsight.

Visual Studio Code (Vscode). We believe that Vscode, despite its recency and still-evolving extension APIs, presents advantageous potential to software visualisation researchers—as well as to those undertaking other SE research in general. This promising potential was discussed in a number of places across this work, particularly in Chapter 6 where its selection was justified based on the capabilities it presented, and later in Chapter 6 where we described how some of those features were utilised. We invite other researchers to capitalise on these advantages and its popularity to promote new software visualisation artefacts. A key advantage is the flexibilities it presents, and especially the potential to develop real-time functionalities—thanks to the lightweight and modern asynchronous features of JavaScript on which the environment is built.

Rendering Performance. While the performance of AgileInsight went beyond our original expectations with regard to its rendering potential, interacting with the scene has certainly become impractical when the number of files processed starts to grow exceedingly large—e.g., the Vscode case of 3280 files. It is worth noting here that this figure refers to the number of files only—the number of individual nodes rendered is thus increasingly multiple times larger. The benchmarks presented in the Chapter 7 were obtained on a 2015 laptop machine whose performance and graphics processing power are relatively outdated. We would like to investigate the threshold of acceptable performance on a more modern machine to give a better picture of what a typical developer would experience. Moreover, it would also be of interest to test the potential of AgileInsight's rendering performance on a more powerful machine with a specialised graphics engine card, as well as to seek independent

opinions from practitioners as to how fast is fast enough when it comes to system response times/delays.

Layout Stability. An outstanding issue in software visualisation research is the (in)stability of visualisation layouts across versions of source code, and its resilience to modifications such as the introduction of new code artefacts, or an increase in the number of lines of a certain code artefact. This is crucial in order to ensure that the overall layout does not change dramatically, and is hence disorienting for the user. This issue is well-known among SV researchers with a number of research efforts have borne techniques proposed in this regard. It is thus certainly of interest to see AgileInsight implementing a stable layout in the future.

Enhancing Visual and Aesthetic Qualities. A visualisation that looks aesthetically more appealing does not only have a positive effect in attracting users, there is also growing discussion in literature about its possible cognitive advantages. As briefly indicated previously, recent research also acknowledges the role of feelings and engagement in promoting user adoption and their subsequent productivity. It is only natural that when a tool is found to be more appealing, users are more inclined to use it—given that it supports their real needs. Given that 'visual presentation' is at the core of software visualisation, it is only logical that importance must be given to aesthetic elements. In this regard, it would be of interest to incorporate some aesthetic enhancements to AgileInsight's visualisation. In particular, we would like to learn from and adopt some of the techniques featured in recent works in this regard such as shading and progressive rendering techniques, which have been demonstrated to result in high quality scenes. Techniques to reduce visual complexity, such as varying level-of-details and aggregation mechanisms, are also expected to introduce significant practicality advantages if implemented in AgileInsight.

REFERENCES

Abad, Zahra Shakeri Hossein, Mohammad Noaeen, and Guenther Ruhe. 2016. “Requirements Engineering Visualization: A Systematic Literature Review.” *2016 IEEE 24th International Requirements Engineering Conference (RE)*, no. 1: 6–15. https://doi.org/10.1109/RE.2016.61.

Abid, Nahla J., Jonathan I. Maletic, and Bonita Sharif. 2019. “Using Developer Eye Movements to Externalize the Mental Model Used in Code Summarization Tasks.” In *Eye Tracking Research and Applications Symposium (ETRA)*. https://doi.org/10.1145/3314111.3319834.

Abuthawabeh, Ala, Fabian Beck, Dirk Zeckzer, and Stephan Diehl. 2013. “Finding Structures in Multi-Type Code Couplings with Node-Link and Matrix Visualizations.” *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*. https://doi.org/10.1109/VISSOFT.2013.6650530.

Agrawal, Ankit, Seyedehzahra Khoshmanesh, Michael Vierhauser, Mona Rahimi, Jane Cleland-Huang, and Robyn Lutz. 2019. “Leveraging Artifact Trees to Evolve and Reuse Safety Cases.” *Proceedings - International Conference on Software Engineering* 2019-May: 1222–33. https://doi.org/10.1109/ICSE.2019.00124.

Ahrens, Maike. 2020. “Towards Automatic Capturing of Traceability Links by Combining Eye Tracking and Interaction Data.” *Proceedings of the IEEE International Conference on Requirements Engineering* 2020-Augus: 434–39. https://doi.org/10.1109/RE48521.2020.00064.

Alan. R Hevner. 2015. “Designing Informing Systems: What Research Tells Us (Presentation).” Tampa, Florida, United States: Informing Science Institute.

Ali, Nasir, Zohreh Sharafi, Yann Gaël Guéhéneuc, and Giuliano Antoniol. 2015. “An Empirical Study on the Importance of Source Code Entities for Requirements Traceability.” *Empirical Software Engineering* 20 (2): 442–78. https://doi.org/10.1007/s10664-014-9315-y.

Aljawabrah, Nadera, Tamas Gergely, Sanjay Misra, and Luis Fernandez-Sanz. 2021. “Automated Recovery and Visualization of Test-to-Code Traceability (TCT) Links: An Evaluation.” *IEEE Access* 9: 40111–23. https://doi.org/10.1109/ACCESS.2021.3063158.

Alshakhouri, Mujtaba. 2013. “ScrumCity: Synchronised Visualisation of Software Process and Product Artefacts.” http://aut.researchgateway.ac.nz/handle/10292/5595.

Alshakhouri, Mujtaba, Jim Buchan, and Stephen G. MacDonell. 2018. “Synchronised Visualisation of Software Process and Product Artefacts: Concept, Design and Prototype Implementation.” *Information and Software Technology* 98 (June): 131–45. https://doi.org/10.1016/j.infsof.2018.01.008.

Andrews, Keith, Josef Wolte, and Michael Pichler. 1997. “Information Pyramids: A New Approach to Visualising Large Hierarchies.” *IEEE Visualization ’97: Proceedings of the 8th Conference on Visualization*, no. October: 49–52.

Ardigo, Susanna, Csaba Nagy, Roberto Minelli, and Michele Lanza. 2021. “Visualizing Data in Software Cities.” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 145–49. https://doi.org/10.1109/VISSOFT52517.2021.00028.

Aung, Thazin Win Win, Huan Huo, and Yulei Sui. 2019. “Interactive Traceability Links Visualization Using Hierarchical Trace Map.” *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, 367–69. https://doi.org/10.1109/ICSME.2019.00059.

Bacchelli, Alberto, Francesco Rigotti, Lile Hattori, and Michele Lanza. 2011. “Manhattan — 3D City Visualizations in Eclipse.” Milano.

Bacher, Ivan, Brian Mac Namee, and John D Kelleher. 2017. “The Code-Map Metaphor - A Review Of Its Use Within Software Visualisations.” *International Conference on Information Visualization Theory and Applications*, no. Visigrapp: 978–89. https://doi.org/10.5220/0006072300170028.

Balci, Faruk, Dilruba Sultan Haliloglu, Onur Sahin, Cankat Tilki, Mehmet Ata Yurtsever, and Eray Tuzun. 2021. “Augmenting Code Review Experience Through Visualization.” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 110–14. https://doi.org/10.1109/VISSOFT52517.2021.00021.

Balzer, Michael, and Oliver Deussen. 2007. “Level-of-Detail Visualization of Clustered Graph Layouts.” In *Asia-Pacific Symposium on Visualisation 2007, APVIS 2007, Proceedings*, 133–40. Ieee. https://doi.org/10.1109/APVIS.2007.329288.

Balzer, Michael, Andreas Noack, Oliver Deussen, and Claus Lewerentz. 2004. “Software Landscapes: Visualizing the Structure of Large Software Systems.” In *VisSym 2004, Symposium on Visualization*, edited by Eurographics Association. Konstanz, Germany: Bibliothek der Universität Konstanz.

Baum, David. 2015. “Introducing Aesthetics to Software Visualization,” no. JUNE.

Baum, David, Stefan Bechert, Ulrich Eisenecker, Isabelle Meichsner, and Richard Muller. 2020. “Identifying Usability Issues of Software Analytics Applications in Immersive Augmented Reality.” *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, 100–104. https://doi.org/10.1109/VISSOFT51673.2020.00015.

Baum, David, Jan Schilbach, Pascal Kovacs, Ulrich Eisenecker, and M Richard. 2017. “GETAVIZ : Generating Structural , Behavioral , and Evolutionary Views of Software Systems for Empirical Evaluation,” 114–18. https://doi.org/10.1109/VISSOFT.2017.12.

Baxter, Kathy, Catherine Courage, and Kelly Caine. 2015. “Surveys.” In *Understanding Your Users*, edited by Kathy Baxter, Catherine Courage, and Kelly B T - Understanding your Users (Second Edition) Caine, 264–301. Boston: Elsevier. https://doi.org/10.1016/B978-0-12-800232-2.00010-9.

Beck, Fabian, Michael Burch, and Stephan Diehl. 2013. “Matching Application Requirements with Dynamic Graph Visualization Profiles.” *Proceedings of the International Conference on Information Visualisation* 304: 11–18. https://doi.org/10.1109/IV.2013.2.

Bedu, Laure, Olivier Tinh, and Fabio Petrillo. 2019. “A Tertiary Systematic Literature Review on Software Visualization.” *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, 33–44. https://doi.org/10.1109/VISSOFT.2019.00013.

Bella, Emma Effa, Marie Pierre Gervais, Reda Bendraou, Laurent Wouters, and Ali Koudri. 2018. “Semi-Supervised Approach for Recovering Traceability Links in Complex Systems.” *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS* 2018-Decem: 193–96. https://doi.org/10.1109/ICECCS2018.2018.00030.

Ben-Ari, Mordechai, and Tzippora Yeshno. 2006. “Conceptual Models of Software Artifacts.” *Interacting with Computers* 18 (6): 1336–50. https://doi.org/10.1016/j.intcom.2006.03.005.

Bergel, Alexandre, Damien Cassou, Stéphane Ducasse, and Jannik Laval. 2012. *Deep into Pharo*.

Bohnet, Johannes, and Jürgen Döllner. 2011. “Monitoring Code Quality and Development Activity by Software Maps.” *Proceedings - International Conference on Software Engineering*, 9–16. https://doi.org/10.1145/1985362.1985365.

Brandt, Sebastian, Michael Striewe, Fabian Beck, and Michael Goedicke. 2017. “A Dashboard for Visualizing Software Engineering Processes Based on ESSENCE.” *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 134–38. https://doi.org/10.1109/VISSOFT.2017.14.

Brehmer, Matthew, and Tamara Munzner. 2013. “A Multi-Level Typology of Abstract Visualization Tasks.” *IEEE Transactions on Visualization and Computer Graphics* 19 (12): 2376–85. https://doi.org/10.1109/TVCG.2013.124.

Brito, Rodrigo, Aline Brito, Gleison Brito, and Marco Tulio Valente. 2019. “GoCity: Code City for Go.” *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, no. Dcc: 649–53. https://doi.org/10.1109/SANER.2019.8668008.

Burgess, Neil, Eleanor A. Maguire, and John O’Keefe. 2002. “The Human Hippocampus and Spatial and Episodic Memory.” *Neuron* 35 (4): 625–41. https://doi.org/10.1016/S0896-6273(02)00830-9.

Burkhard, R a. 2005. “Towards a Framework and a Model for Knowledge Visualization: Synergies between Information and Knowledge Visualization.” *Lecture Notes in Computer Science* 3426: 238–55. https://doi.org/10.1007/11510154_13.

Burstein, Frada. 2002. “System Development in Information Systems Research.” *Research Methods for Students, Academics and Professionals: Information Management and Systems* 2: 147–56.

Chen, Xiaofan, John Hosking, John Grundy, and Robert Amor. 2018. “DCTracVis: A System Retrieving and Visualizing Traceability Links between Source Code and Documentation.” *Automated Software Engineering* 25 (4): 703–41. https://doi.org/10.1007/s10515-018-0243-8.

Chotisarn, Noptanit, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. 2020. “A Systematic Literature Review of Modern Software Visualization.” *Journal of Visualization* 23 (4): 539–58. https://doi.org/10.1007/s12650-020-00647-w.

Cleland-Huang, Jane. 2015. “Towards Effective Software and Systems Traceability.”

Cleland-Huang, Jane, Ankit Agrawal, Michael Vierhauser, and Christoph Mayr-Dorn. 2021. “Visualizing Change in Agile Safety-Critical Systems.” *IEEE Software* 38 (3): 43–51. https://doi.org/10.1109/MS.2020.3000104.

Cleland-Huang, Jane, and Rafal Habrat. 2007. “Visual Support in Automated Tracing.” In *2nd International Workshop on Requirements Engineering Visualization, REV 2007*, 23–27. https://doi.org/10.1109/REV.2007.7.

Cleland-Huang, Jane, Mona Rahimi, and Patrick Mäder. 2014. “Achieving Lightweight Trustworthy Traceability.” *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, 849–52. https://doi.org/10.1145/2635868.2666612.

Cleland-Huang, Jane, and Michael Vierhauser. 2018. “Discovering, Analyzing, and Managing Safety Stories in Agile Projects.” *Proceedings - 2018 IEEE 26th International Requirements Engineering Conference, RE 2018*, 262–73. https://doi.org/10.1109/RE.2018.00034.

Cole, Robert, Sandeep Purao, Matti Rossi, and Maung K. Sein. 2005. “Being Proactive : Where Action Research Meets Design Research.” *Running Head: PROACTIVE RESEARCH APPROACHES.*, 1–21.

Cronholm, Stefan, Hannes Göbel, and Anders Hjalmarsson. 2016. “Empirical Evaluation of Action Design Research.” *Australasian Conference on Information Systems*, 1–12.

Cruz, Adriana, Camila Bastos, Paulo Afonso, and Heitor Costa. 2016. “Software Visualization Tools and Techniques: A Systematic Review of the Literature.” In *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*, 1–12. IEEE. https://doi.org/10.1109/SCCC.2016.7836048.

Curcio, Karina, Tiago Navarro, Andreia Malucelli, and Sheila Reinehr. 2018. “Requirements Engineering: A Systematic Mapping Study in Agile Software Development.” *Journal of Systems and Software* 139: 32–50. https://doi.org/10.1016/j.jss.2018.01.036.

Daeuble, Gerald, Michael Werner, and Markus Nuettgens. 2015. “Artifact-Centered Planning and Assessing of Large Design Science Research Projects – a Case Study.” *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9073: 343–57. https://doi.org/10.1007/978-3-319-18714-3_22.

Dashuber, Veronika, and Michael Philippsen. 2021. “Trace Visualization within the Software City Metaphor: A Controlled Experiment on Program Comprehension.” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 55–64. https://doi.org/10.1109/VISSOFT52517.2021.00015.

Delater, Alexander, and Barbara Paech. 2013a. “Analyzing the Tracing of Requirements and Source Code during Software Development.” In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7830 LNCS:308–14. https://doi.org/10.1007/978-3-642-37422-7_22.

———. 2013b. “Tracing Requirements and Source Code during Software Development: An Empirical Study.” *ESEM 2013: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, no. c: 25–34. https://doi.org/10.1109/ESEM.2013.16.

———. 2013c. “UNICASE Trace Client: (Semi-) Automatic Tracing of Requirements and Code During Development for Small and Medium Enterprises.” *Softwaretechnik-Trends* 33 (1): 27–28. https://doi.org/10.1007/BF03323543.

———. 2013d. “Tracing Requirements and Source Code during Software Development: An Empirical Study.” In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 25–34. IEEE. https://doi.org/10.1109/ESEM.2013.16.

Diehl, Stephan. 2007a. *Software Visualization*.

———. 2007b. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.

Ducasse, Stéphane, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. 2011. “MSE and FAMIX 3.0: An Interexchange Format and Source Code Model Family.”

Duru, Haci Ali, Murat Perit Çakir, and Veysi Işler. 2013. “How Does Software Visualization Contribute to Software Comprehension? A Grounded Theory Approach.” *International Journal of Human-Computer Interaction* 29 (11): 743–63. https://doi.org/10.1080/10447318.2013.773876.

Fabry, Johan, Andy Kellens, Simon Denier, and Stéphane Ducasse. 2014. “AspectMaps: Extending Moose to Visualize AOP Software.” *Science of Computer Programming* 79: 6–22. https://doi.org/10.1016/j.scico.2012.02.007.

Feijs, Loe, and Roel De Jong. 1998. “3D Visualization of Software Architectures.” *Communications of the ACM* 41 (12): 73–78. https://doi.org/10.1145/290133.290151.

Feist, Michael D., Eddie Antonio Santos, Ian Watts, and Abram Hindle. 2016. “Visualizing Project Evolution through Abstract Syntax Tree Analysis.” In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, 11–20. IEEE. https://doi.org/10.1109/VISSOFT.2016.6.

Fernandes, Sara, André Restivo, Hugo Sereno Ferreira, and Ademar Aguiar. 2020. “Helping Software Developers through Live Software Metrics Visualization.” *ACM International Conference Proceeding Series*, 209–10. https://doi.org/10.1145/3397537.3397539.

Fiechter, Aron, Roberto Minelli, Csaba Nagy, and Michele Lanza. 2021. “Visualizing GitHub Issues.” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, no. 3: 155–59. https://doi.org/10.1109/VISSOFT52517.2021.00030.

Fittkau, Florian, Alexander Krause, and Wilhelm Hasselbring. 2015. “Hierarchical Software Landscape Visualization for System Comprehension: A Controlled Experiment.” *2015 IEEE 3rd Working Conference on*

Software Visualization, VISSOFT 2015 - Proceedings, 36–45. https://doi.org/10.1109/VISSOFT.2015.7332413.

———. 2017. "Software Landscape and Application Visualization for System Comprehension with ExplorViz." *Information and Software Technology* 87: 259–77. https://doi.org/10.1016/j.infsof.2016.07.004.

García, Félix, Mª Ángeles Moraga, Manuel Serrano, and Mario Piattini. 2015. "Visualisation Environment for Global Software Development Management." *IET Software* 9 (2): 51–64. https://doi.org/10.1049/iet-sen.2013.0193.

Gauerhof, Lydia, Roman Gansch, Christian Heinzemann, Matthias Woehrle, and Andreas Heyl. 2022. "On the Necessity of Explicit Artifact Links in Safety Assurance Cases for Machine Learning," 23–28. https://doi.org/10.1109/issrew53611.2021.00069.

Genfer, Patric, Johann Grabner, Christina Zoffi, Mario Bernhart, and Thomas Grechenig. 2021. "Visualizing Metric Trends for Software Portfolio Quality Management." In *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 88–99. IEEE. https://doi.org/10.1109/VISSOFT52517.2021.00018.

Ghazi, Parisa, Norbert Seyff, and Martin Glinz. 2013. "FlexiView: A Magnet-Based Approach for Visualizing Requirements Artifacts" 7830: 262–69. https://doi.org/10.1007/978-3-642-37422-7.

Girba, Tudor, and Andrei Chis. 2015. "Pervasive Software Visualizations (Keynote)." In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, 1–5. IEEE. https://doi.org/10.1109/VISSOFT.2015.7332409.

Goodrum, Micayla, Jane Cleland-Huang, Robyn Lutz, Jinghui Cheng, and Ronald Metoyer. 2017. "What Requirements Knowledge Do Developers Need to Manage Change in Safety-Critical Systems?" In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, 90–99. IEEE. https://doi.org/10.1109/RE.2017.65.

Gotel, Orlena C.Z., Francis T. Marchese, and Stephen J. Morris. 2007. "On Requirements Visualization." *2nd International Workshop on Requirements Engineering Visualization, REV 2007*, no. Rev. https://doi.org/10.1109/REV.2007.4.

Graham, Hamish, Hong Yul Yang, and Rebecca Berrigan. 2004. "A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics." *Proceedings of the 2004 Australasian Symposium on Information Visualisation - Volume 35*, 163.

Gulden, Jens, and Hajo A. Reijers. 2015. "Toward Advanced Visualization Techniques for Conceptual Modeling." *CEUR Workshop Proceedings* 1367: 33–40.

Guo, Jin, Mona Rahimi, Jane Cleland-Huang, Alexander Rasin, Jane Huffman Hayes, and Michael Vierhauser. 2016. "Cold-Start Software Analytics." *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, 142–53. https://doi.org/10.1145/2901739.2901740.

Hammoudi, Mouna, Christoph Mayr-Dorn, Atif Mashkoor, and Alexander Egyed. 2021. "A Traceability Dataset for Open Source Systems." *Proceedings - 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR 2021*, 555–59. https://doi.org/10.1109/MSR52588.2021.00073.

Hasan, Mahmudul, Faramarz F. Samavati, and Christian Jacob. 2014. "Multilevel Focus+context Visualization Using Balanced Multiresolution." *Proceedings - 2014 International Conference on Cyberworlds, CW 2014*, 145–52. https://doi.org/10.1109/CW.2014.28.

Hayes, Jane Huffman, Wenbin Li, and Mona Rahimi. 2014. "Weka Meets TraceLab: Toward Convenient Classification: Machine Learning for Requirements Engineering Problems: A Position Paper." *2014 IEEE 1st International Workshop on Artificial Intelligence for Requirements Engineering, AIRE 2014 - Proceedings*, 9–12. https://doi.org/10.1109/AIRE.2014.6894850.

Heer, Jeffrey, and K Stuart Card. 2004. "DOITrees Revisited: Scalable, Space-Constrained Visualization of Hierarchical Data." *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI)*, 421–24. https://doi.org/10.1145/989863.989941.

Heidmann, Elke Franziska, Lynn von Kurnatowski, Annika Meinecke, and Andreas Schreiber. 2020. "Visualization of Evolution of Component-Based Software Architectures in Virtual Reality." *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, 12–21. https://doi.org/10.1109/VISSOFT51673.2020.00006.

Hess, Anne, Philipp Diebold, and Norbert Seyff. 2018. "Understanding Information Needs of Agile Teams to Improve Requirements Communication (Special Issue Edited by Nan Niu and Daniel Mendez)." *Journal of Industrial Information Integration*, no. April. https://doi.org/10.1016/j.jii.2018.04.002.

Hevner, Alan, and Samir Chatterjee. 2010. "Design Research in Information Systems" 22: 9–23. https://doi.org/10.1007/978-1-4419-5653-8.

Hevner, Alan R. 2007. “A Three Cycle View of Design Science Research.” *Scandinavian Journal of Information Systems © Scandinavian Journal of Information Systems* 19 (192): 87–92. https://doi.org/http://aisel.aisnet.org/sjis/vol19/iss2/4.

Hevner, Alan R, Salvatore T March, Jinsoo Park, and Sudha Ram. 2004. “Design Science in Information Systems Research.” *MIS Quarterly* 28 (1): 75–105. https://doi.org/10.2307/25148625.

Humayoun, Shah Rukh, Syed Moiz Hasan, Ragaad AlTarawneh, and Achim Ebert. 2018. “Visualizing Software Hierarchy and Metrics over Releases.” *Proceedings of the Workshop on Advanced Visual Interfaces AVI*. https://doi.org/10.1145/3206505.3206548.

Iivari, Juhani. 2007. “A Paradigmatic Analysis of Information Systems As a Design Science.” *Scandinavian Journal of Information Systems © Scandinavian Journal of Information Systems* 19 (2): 39–64. http://aisel.aisnet.org/sjis%0Ahttp://aisel.aisnet.org/sjis/vol19/iss2/5.

Jackson, Daniel. 2013. “Conceptual Design of Software: A Research Agenda.” *Massachusetts Inst. Technol.*

———. 2015. “Towards a Theory of Conceptual Design for Software.” *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 282–96. https://doi.org/10.1145/2814228.2814248.

Jeffery, Clinton L. 2019. “The City Metaphor in Software Visualization.” In *Computer Science Research Notes*, 2901:153–61. Západočeská univerzita. https://doi.org/10.24132/CSRN.2019.2901.1.18.

Johnson, Brian, and Ben Shneiderman. 1991. “Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures.” *Proceedings of the 2nd Conference on Visualization 1991, VIS 1991*, 284–91. https://doi.org/10.1109/visual.1991.175815.

Khaloo, Pooya, Mehran Maghoumi, Eugene Taranta, David Bettner, and Joseph Laviola. 2017. “Code Park: A New 3D Code Visualization Tool.” https://doi.org/10.1109/VISSOFT.2017.10.

Kiesow, Andreas, Novica Zarvić, and Oliver Thomas. 2015. “Design Science for Future Ais: Transferring Continuous Auditing Issues to a Gradual Methodology.” *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9073: 311–26. https://doi.org/10.1007/978-3-319-18714-3_20.

Kim, Youngtaek, Jaeyoung Kim, Hyeon Jeon, Young Ho Kim, Hyunjoo Song, Bohyoung Kim, and Jinwook Seo. 2021. “Githru: Visual Analytics for Understanding Software Development History through Git Metadata Analysis.” *IEEE Transactions on Visualization and Computer Graphics* 27 (2): 656–66. https://doi.org/10.1109/TVCG.2020.3030414.

Kleebaum, Anja, Barbara Paech, Jan Ole Johanssen, and Bernd Bruegge. 2021. “Continuous Rationale Visualization.” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 33–43. https://doi.org/10.1109/VISSOFT52517.2021.00013.

Knight, Claire, and Malcolm Munro. 1999. “Comprehension with[in] Virtual Environment Visualisations.” *Proceedings - 7th International Workshop on Program Comprehension, IWPC 1999*, 4–11. https://doi.org/10.1109/WPC.1999.777733.

Krause, Alexander, Malte Hansen, and Wilhelm Hasselbring. 2021. “Live Visualization of Dynamic Software Cities with Heat Map Overlays.” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 125–29. https://doi.org/10.1109/VISSOFT52517.2021.00024.

Krebs, Raffael. 2012. “Vera: An Extensible Eclipse Plug-In for Java Enterprise Application Analysis.” http://scg.unibe.ch/archive/masters/Kreb12a.pdf.

Kritzinger, Lisa Maria, Thomas Krismayer, Rick Rabiser, and Paul Grunbacher. 2019. “A User Study on the Usefulness of Visualization Support for Requirements Monitoring.” *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, no. i: 56–66. https://doi.org/10.1109/VISSOFT.2019.00015.

Kuhn, Adrian, David Erni, and O. Nierstrasz. 2010. “Embedding Spatial Software Visualization in the IDE: An Exploratory Study.” In *Proceedings of the 5th International Symposium on Software Visualization*, 113–122. ACM.

Lanza, Michele, Marco D’Ambros, Alberto Bacchelli, Lile Hattori, and Francesco Rigotti. 2013. “Manhattan: Supporting Real-Time Visual Team Activity Awareness.” *IEEE International Conference on Program Comprehension*, 207–10. https://doi.org/10.1109/ICPC.2013.6613849.

Lau, Andrea, and Andrew Vande Moere. 2007. “Towards a Model of Information Aesthetics in Information Visualization.” *Proceedings of the International Conference on Information Visualisation*, 87–92. https://doi.org/10.1109/IV.2007.114.

Lian, Xiaoli, Mona Rahimi, Jane Cleland-Huang, Li Zhang, Remo Ferrai, and Michael Smith. 2016. “Mining Requirements Knowledge from Collections of Domain Documents.” *Proceedings - 2016 IEEE 24th*

International Requirements Engineering Conference, RE 2016, 156–65. https://doi.org/10.1109/RE.2016.50.

Limberger, Daniel, Marcel Pursche, Jan Klimke, and Jürgen Döllner. 2017. "Progressive High-Quality Rendering for Interactive Information Cartography Using WebGL." In *Proceedings of the 22nd International Conference on 3D Web Technology*, 1–4. New York, NY, USA: ACM. https://doi.org/10.1145/3055624.3075951.

Limberger, Daniel, Willy Scheibel, Jürgen Döllner, and Matthias Trapp. 2019. "Advanced Visual Metaphors and Techniques for Software Maps." *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3356422.3356444.

Limberger, Daniel, Willy Scheibel, Sebastian Hahn, and Jurgen Dollner. 2017. "Reducing Visual Complexity in Software Maps Using Importance-Based Aggregation of Nodes." *VISIGRAPP 2017 - Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications* 3: 176–85. https://doi.org/10.5220/0006267501760185.

Limberger, Daniel, Willy Scheibel, Stefan Lemme, and Jürgen Döllner. 2016. "Dynamic 2.5D Treemaps Using Declarative 3D on the Web." In *Proceedings of the 21st International Conference on Web3D Technology*, 33–36. New York, NY, USA: ACM. https://doi.org/10.1145/2945292.2945313.

Limberger, Daniel, Benjamin Wasty, Jonas Trümper, and Jürgen Döllner. 2013. "Interactive Software Maps for Web-Based Source Code Analysis." *Proceedings of the 18th International Conference on 3D Web Technology - Web3D '13*, 91. https://doi.org/10.1145/2466533.2466550.

Liu, Mingnan, Sunghee Lee, and Frederick G. Conrad. 2015. "Comparing Extreme Response Styles between Agree-Disagree and Item-Specific Scales." *Public Opinion Quarterly* 79 (4): 952–75. https://doi.org/10.1093/poq/nfv034.

Lopez-Herrejon, Roberto Erick, Sheny Illescas, and Alexander Egyed. 2018. "A Systematic Mapping Study of Information Visualization for Software Product Line Engineering." *Journal of Software: Evolution and Process* 30 (2): 1–18. https://doi.org/10.1002/smr.1912.

Lucassen, Garm, Fabiano Dalpiaz, Jan Martijn E. M. van der Werf, and Sjaak Brinkkemper. 2016. "The Use and Effectiveness of User Stories in Practice." In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, edited by Maya Daneva and Oscar Pastor, 9619:205–22. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-30282-9_14.

Lucassen, Garm, Fabiano Dalpiaz, Jan Martijn E. M. van der Werf, and Sjaak Brinkkemper. 2016. "Visualizing User Story Requirements at Multiple Granularity Levels via Semantic Relatedness." In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9974 LNCS:463–78. https://doi.org/10.1007/978-3-319-46397-1_35.

Lucassen, Garm, Marcel Robeer, Fabiano Dalpiaz, Jan Martijn E.M. van der Werf, and Sjaak Brinkkemper. 2017. "Extracting Conceptual Models from User Stories with Visual Narrator." *Requirements Engineering* 22 (3): 339–58. https://doi.org/10.1007/s00766-017-0270-1.

Lüders, Clara Marie, Mikko Raatikainen, Joaquim Motger, and Walid Maalej. 2019. "OpenReq Issue Link Map: A Tool to Visualize Issue Links in Jira." *Proceedings of the IEEE International Conference on Requirements Engineering* 2019-Septe: 492–93. https://doi.org/10.1109/RE.2019.00070.

Luz, Saturnino, and Masood Masoodian. 2010. "Improving Focus and Context Awareness in Interactive Visualization of Time Lines." *Proceedings of the 24th BCS Interaction Specialist Group Conference*, 72–80. http://dl.acm.org/citation.cfm?id=2146303.2146314.

Maletic, J I, Jason Leigh, and Andrian Marcus. 2001. "Visualizing Software in an Immersive Virtual Reality Environment." *Proc. ICSE Work. Softw. Vis.*, 49–54.

Marcén, Ana C., Raúl Lapeña, Óscar Pastor, and Carlos Cetina. 2020. "Traceability Link Recovery between Requirements and Models Using an Evolutionary Algorithm Guided by a Learning to Rank Algorithm: Train Control and Management Case." *Journal of Systems and Software* 163. https://doi.org/10.1016/j.jss.2020.110519.

Maro, Salome, Jan-Philipp Steghöfer, Paolo Bozzelli, and Henry Muccini. 2022. "TracIMo: A Traceability Introduction Methodology and Its Evaluation in an Agile Development Team." *Requirements Engineering* 27 (1): 53–81. https://doi.org/10.1007/s00766-021-00361-5.

Mattila, A.-L., P. Ihantola, T. Kilamo, A. Luoto, M. Nurminen, and H. Väätäjä. 2016. "Software Visualization Today - Systematic Literature Review." *AcademicMindtrek 2016 - Proceedings of the 20th International Academic Mindtrek Conference*. https://doi.org/10.1145/2994310.2994327.

Mattila, Anna-Liisa, Kari Systä, Outi Sievi-Korte, Marko Leppänen, and Tommi Mikkonen. 2017. “Discovering Software Process Deviations Using Visualizations.” In *Lecture Notes in Business Information Processing*, 283:259–66. https://doi.org/10.1007/978-3-319-57633-6_18.

Mehra, Rohit, Vibhu Saujanya Sharma, Vikrant Kaulgud, Sanjay Podder, and Adam P. Burden. 2020. “Towards Immersive Comprehension of Software Systems Using Augmented Reality - An Empirical Evaluation.” *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, 1267–69. https://doi.org/10.1145/3324884.3418907.

Merino, L., M. Ghafari, C. Anslow, and O. Nierstrasz. 2018. “A Systematic Literature Review of Software Visualization Evaluation.” *Journal of Systems and Software* 144 (October 2017): 165–80. https://doi.org/10.1016/j.jss.2018.06.027.

Merino, Leonel, Alexandre Bergel, and Oscar Nierstrasz. 2018. “Overcoming Issues of 3D Software Visualization through Immersive Augmented Reality.” *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018*, 54–64. https://doi.org/10.1109/VISSOFT.2018.00014.

Merino, Leonel, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel A Keim. 2017. “On the Impact of the Medium in the Effectiveness of 3D Software Visualizations.” In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 11–21. IEEE. https://doi.org/10.1109/VISSOFT.2017.17.

Merino, Leonel, Mohammad Ghafari, and Oscar Nierstrasz. 2016. “Towards Actionable Visualisation in Software Development.” *Proceedings - 2016 IEEE Working Conference on Software Visualization, VISSOFT 2016*, 61–70. https://doi.org/10.1109/VISSOFT.2016.10.

———. 2018. “Towards Actionable Visualization for Software Developers.” *Journal of Software: Evolution and Process* 30 (2): e1923. https://doi.org/10.1002/smr.1923.

Merino, Leonel, Mohammad Ghafari, Oscar Nierstrasz, Alexandre Bergel, and Juraj Kubelka. 2016. “MetaVis: Exploring Actionable Visualization.” In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, 151–55. IEEE. https://doi.org/10.1109/VISSOFT.2016.19.

Merino, Leonel, Ekaterina Kozlova, Oscar Nierstrasz, and Daniel Weiskopf. 2019. “VISON: An Ontology-Based Approach for Software Visualization Tool Discoverability.” *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, 45–55. https://doi.org/10.1109/VISSOFT.2019.00014.

Meyer, Miriah, Michael Sedlmair, and Tamara Munzner. 2012. “The Four-Level Nested Model Revisited: Blocks and Guidelines.” *ACM International Conference Proceeding Series*, 1–6. https://doi.org/10.1145/2442576.2442587.

Mirakhorli, Mehdi, and Jane Cleland-Huang. 2016. “Detecting, Tracing, and Monitoring Architectural Tactics in Code.” *IEEE Transactions on Software Engineering* 42 (3): 206–21. https://doi.org/10.1109/TSE.2015.2479217.

Montaño, David, Jairo Aponte, and Andrian Marcus. 2009. “Sv3D Meets Eclipse.” *Proceedings of VISSOFT 2009 - 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 51–54. https://doi.org/10.1109/VISSOF.2009.5336413.

Moreno-Lumbreras, David, Roberto Minelli, Andrea Villaverde, Jesus M. Gonzalez-Barahona, and Michele Lanza. 2021. “CodeCity: On-Screen or in Virtual Reality?” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 12–22. https://doi.org/10.1109/VISSOFT52517.2021.00011.

Müller, Richard, Pascal Kovacs, Jan Schilbach, Ulrich Eisenecker, Dirk Zeckzer, and Gerik Scheuermann. 2014. “A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization.” *Proceedings of the 5th International Conference on Visualization Theory and Applications*, 204–9. https://doi.org/10.5220/0004835202040209.

Muller, Richard, Pascal Kovacs, Jan Schilbach, and Dirk Zeckzer. 2015. “How to Master Challenges in Experimental Evaluation of 2D versus 3D Software Visualizations.” *2014 IEEE VIS International Workshop on 3DVis, 3DVis 2014*, 33–36. https://doi.org/10.1109/3DVis.2014.7160097.

Munzner, Tamara. 2009. “A Nested Model for Visualization Design and Validation.” *IEEE Transactions on Visualization and Computer Graphics* 15 (6): 921–28. https://doi.org/10.1109/TVCG.2009.111.

Narayan, Nitesh, Jan Finis, Yang Li, and Alexander Delater. 2012. “Leveraging Traceability between Code and Tasks for Code Review and Release Management.” *Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA ´12)*, no. c: 8–14.

Nierstrasz, Oscar, Stéphane Ducasse, and Tudor Girba. 2005. “The Story of Moose: An Agile Reengineering Environment.” *SIGSOFT Softw. Eng. Notes* 30 (5): 1–10. https://doi.org/10.1145/1095430.1081707.

Novais, Renato L., Camila Nunes, Alessandro Garcia, and Manoel Mendonça. 2013. “SourceMiner Evolution: A Tool for Supporting Feature Evolution Comprehension.” *IEEE International Conference on Software Maintenance, ICSM*, 508–11. https://doi.org/10.1109/ICSM.2013.83.

Novais, Renato, José Amancio Santos, and Manoel Mendonça. 2017. “Experimentally Assessing the Combination of Multiple Visualization Strategies for Software Evolution Analysis.” *Journal of Systems and Software* 128: 56–71. https://doi.org/10.1016/j.jss.2017.03.006.

Nunamaker, Jay F. Jr., Minder Chen, and Titus D. M. Purdin. 1990. “Systems Development in Information Systems Research.” *Journal of Management Information Systems* 7 (3): 89–106.

Nunes, Rafael, Marcel Reboucas, Francisco Soares-Neto, and Fernando Castor. 2017. “Visualizing Swift Projects as Cities.” *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, 368–70. https://doi.org/10.1109/ICSE-C.2017.115.

Oppl, Stefan. 2017. *Supporting the Collaborative Construction of a Shared Understanding About Work with a Guided Conceptual Modeling Technique*. *Group Decision and Negotiation*. Vol. 26. https://doi.org/10.1007/s10726-016-9485-7.

Panas, Thomas, Rebecca Berrigan, and John Grundy. 2003. “A 3D Metaphor for Software Production Visualization.” *Proceedings of the International Conference on Information Visualisation* 2003-Janua: 314–19. https://doi.org/10.1109/IV.2003.1217996.

Paredes, Julia, Craig Anslow, and Frank Maurer. 2014. “Information Visualization for Agile Software Development.” *Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014*, 157–66. https://doi.org/10.1109/VISSOFT.2014.32.

Pawlak, Renaud, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. “<scp>SPOON</Scp> : A Library for Implementing Analyses and Transformations of Java Source Code.” *Software: Practice and Experience* 46 (9): 1155–79. https://doi.org/10.1002/spe.2346.

Petre, Marian. 2002. “Mental Imagery, Visualisation Tools and Team Work.” *Proceedings of the Second Program Visualization Workshop*, no. June: 2–13.

———. 2010. “Mental Imagery and Software Visualization in High-Performance Software Development Teams.” *Journal of Visual Languages and Computing* 21 (3): 171–83. https://doi.org/10.1016/j.jvlc.2009.11.001.

Petre, Marian, A Blackwell, and T. R. G. Green. 1998. *Cognitive Questions in Software Visualization*. *Software Visualization:* https://doi.org/citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.8820.

Pfahler, Federico, Roberto Minelli, Csaba Nagy, and Michele Lanza. 2020. “Visualizing Evolving Software Cities.” *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, 22–26. https://doi.org/10.1109/VISSOFT51673.2020.00007.

Pleuss, Andreas, Rick Rabiser, and Goetz Botterweck. 2011. “Visualization Techniques for Application in Interactive Product Configuration.” *ACM International Conference Proceeding Series*. https://doi.org/10.1145/2019136.2019161.

Rahimi, Mona, and Jane Cleland-Huang. 2017. “Artifact: Cassandra Source Code, Feature Descriptions across 27 Versions, with Starting and Ending Version Trace Matrices.” *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* 2: 612. https://doi.org/10.1109/ICSME.2016.42.

Rahimi, Mona, William Goss, and Jane Cleland-Huang. 2017. “Evolving Requirements-to-Code Trace Links across Versions of a Software System.” *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, 99–109. https://doi.org/10.1109/ICSME.2016.57.

Riegel, Norman, and Joerg Doerr. 2015. *A Systematic Literature Review of Requirements Prioritization Criteria*. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9013. https://doi.org/10.1007/978-3-319-16101-3_22.

Rilling, Juergen, and S. P. Mudur. 2002. “On the Use of Metaballs to Visually Map Source Code Structures and Analysis Results onto 3D Space.” *Proceedings - Working Conference on Reverse Engineering, WCRE* 2002-Janua: 299–308. https://doi.org/10.1109/WCRE.2002.1173087.

Robeer, Marcel, Garm Lucassen, Jan Martijn E.M. Van Der Werf, Fabiano Dalpiaz, and Sjaak Brinkkemper. 2016. “Automated Extraction of Conceptual Models from User Stories via NLP.” In *Proceedings - 2016 IEEE 24th International Requirements Engineering Conference, RE 2016*, 196–205. https://doi.org/10.1109/RE.2016.40.

Roberts, Jonathan C., Panagiotis D. Ritsos, James R. Jackson, and Christopher Headleand. 2018. “The Explanatory Visualization Framework: An Active Learning Framework for Teaching Creative Computing Using Explanatory Visualizations.” *IEEE Transactions on Visualization and Computer Graphics* 24 (1): 791–801. https://doi.org/10.1109/TVCG.2017.2745878.

Romano, Simone, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. 2019. “On the Use of Virtual Reality in Software Visualization: The Case of the City Metaphor.” *Information and Software Technology* 114 (June): 92–106. https://doi.org/10.1016/j.infsof.2019.06.007.

Rosso, Santiago Perez De, and Daniel Jackson. 2016. “Purposes, Concepts, Misfits, and a Redesign of Git.” *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, 292–310. https://doi.org/10.1145/2983990.2984018.

Rubasinghe, I. D., D. A. Meedeniya, and I. Perera. 2018. “Software Artefact Traceability Analyser: A Case-Study on POS System.” *ACM International Conference Proceeding Series*, 1–5. https://doi.org/10.1145/3193092.3193094.

Rubasinghe, Iresha, Dulani Meedeniya, and Indika Perera. 2018. “Traceability Management with Impact Analysis in DevOps Based Software Development.” *2018 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2018*, 1956–62. https://doi.org/10.1109/ICACCI.2018.8554399.

Ruf, Christian, and Andrea Back. 2015. “How Can We Design Products, Services, and Software That Reflect the Needs of Our Stakeholders? Towards a Canvas for Successful Requirements Engineering.” *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9073: 455–62. https://doi.org/10.1007/978-3-319-18714-3_38.

Saito, Shinobu, Yukako Iimura, Aaron K Massey, and Annie I Antón. 2018. “Discovering Undocumented Knowledge through Visualization of Agile Software Development Activities.” *Requirements Engineering*, no. 0123456789 (April). https://doi.org/10.1007/s00766-018-0291-4.

Salameh, Hani Bani, and Ashraf Aljammal. 2016. “Software Evolution Visualization Techniques and Methods - a Systematic Review.” *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, 1–6. https://doi.org/10.1109/CSIT.2016.7549475.

Saris, Willem E, Melanie Revilla, Jon A Krosnick, and Eric M Shaeffer. 2010. “Comparing Questions with Agree/Disagree Response Options to Questions with Item-Specific Response Options.” *Survey Research Methods*. Vol. 4. http://www.surveymethods.org.

Sauro, Jeff, and James R Lewis. 2016. “Standardized Usability Questionnaires.” In *Quantifying the User Experience*, edited by Jeff Sauro and James R B T - Quantifying the User Experience (Second Edition) Lewis, 185–248. Boston: Elsevier. https://doi.org/10.1016/B978-0-12-802308-2.00008-4.

Savio, Deepti, and Ashok Pancily Poothiyot. 2014. “Extended Support for Visualizing Requirements: Filtering and Tracing Requirements in ReBlock.” *2014 IEEE 5th International Workshop on Requirements Prioritization and Communication, RePriCo 2014 - Proceedings*, 11–14. https://doi.org/10.1109/RePriCo.2014.6895217.

Schreiber, Andreas, Lynn Von Kurnatowski, Annika Meinecke, and Claas De Boer. 2021. “An Interactive Dashboard for Visualizing the Provenance of Software Development Processes.” *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 100–104. https://doi.org/10.1109/VISSOFT52517.2021.00019.

Scott-Hill, Brandon, Craig Anslow, Jennifer Ferreira, Martin Kropp, Magdalena Mateescu, and Andreas Meier. 2020. “Visualizing Progress Tracking for Software Teams on Large Collaborative Touch Displays.” *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC* 2020-Augus. https://doi.org/10.1109/VL/HCC50065.2020.9127286.

Seider, Doreen, Andreas Schreiber, Tobias Marquardt, and Marlene Bruggemann. 2016. “Visualizing Modules and Dependencies of OSGi-Based Applications.” In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, 96–100. IEEE. https://doi.org/10.1109/VISSOFT.2016.20.

Seiler, Marcus, Paul Hubner, and Barbara Paech. 2019. “Comparing Traceability through Information Retrieval, Commits, Interaction Logs, and Tags.” *Proceedings - 2019 IEEE/ACM 10th International Workshop on Software and Systems Traceability, SST 2019*, 21–28. https://doi.org/10.1109/SST.2019.00015.

Seiler, Marcus, and Barbara Paech. 2017. “Using Tags to Support Feature Management across Issue Tracking Systems and Version Control Systems: A Research Preview.” In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, edited by Paul Grünbacher and Anna Perini, 10153 LNCS:174–80. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-54045-0_13.

Sein, Maung K., Ola Henfridsson, Matti Rossi, Ola Henfridsson, Sandeep Purao, Matti Rossi, and Rikard Lindgren. 2011. “ACTION DESIGN RESEARCH.” *MIS Quarterly* 35 (2): 1–20.

Seipel, Peter, Adrian Stock, Sivasurya Santhanam, Artur Baranowski, Nico Hochgeschwender, and Andreas Schreiber. 2019. “Speak to Your Software Visualization-Exploring Component-Based Software Architectures in Augmented Reality with a Conversational Interface.” *Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019*, 78–82. https://doi.org/10.1109/VISSOFT.2019.00017.

Sensalire, Mariam, Patrick Ogao, and Alexandru Telea. 2009. “Evaluation of Software Visualization Tools: Lessons Learned.” *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 19–26. https://doi.org/10.1109/VISSOF.2009.5336431.

Seriai, Abderrahmane, Omar Benomar, Benjamin Cerat, and Houari Sahraoui. 2014. “Validation of Software Visualization Tools: A Systematic Mapping Study.” *2014 Second IEEE Working Conference on Software Visualization*, 60–69. https://doi.org/10.1109/VISSOFT.2014.19.

Shahin, Mojtaba, Peng Liang, and Muhammad Ali Babar. 2014. “A Systematic Review of Software Architecture Visualization Techniques.” *Journal of Systems and Software* 94: 161–85. https://doi.org/10.1016/j.jss.2014.03.071.

Sharafi, Zohreh, Alessandro Marchetto, Angelo Susi, Giuliano Antoniol, and Yann Gaal Gueheneuc. 2013. “An Empirical Study on the Efficiency of Graphical vs. Textual Representations in Requirements Comprehension.” *IEEE International Conference on Program Comprehension* 15 (5): 33–42. https://doi.org/10.1109/ICPC.2013.6613831.

Sharif, Bonita, Grace Jetty, Jairo Aponte, and Esteban Parra. 2013. “An Empirical Study Assessing the Effect of SeeIT 3D on Comprehension.” *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*. https://doi.org/10.1109/VISSOFT.2013.6650519.

Shneiderman, Ben. 1992. “Tree Visualization with Tree-Maps.” *ACM Transactions on Graphics* 11 (1): 92–99.

Shrestha, Ayush, Ying Zhu, and Ben Miller. 2013. “Visualizing Time and Geography of Open Source Software with Storygraph.” *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*. https://doi.org/10.1109/VISSOFT.2013.6650532.

Siau, Keng, and Xin Tan. 2005. “Improving the Quality of Conceptual Modeling Using Cognitive Mapping Techniques.” *Data and Knowledge Engineering* 55 (3): 343–65. https://doi.org/10.1016/j.datak.2004.12.006.

Sinhabahu, Nadun, Prasad Wimalaratne, and Chaman Wijesiriwardana. 2020. “Secure Codecity with Evolution: Visualizing Security Vulnerability Evolution of Software Systems.” *20th International Conference on Advances in ICT for Emerging Regions, ICTer 2020 - Proceedings*, no. ICTer: 302–3. https://doi.org/10.1109/ICTer51097.2020.9325429.

Slob, Govert Jan, Fabiano Dalpiaz, Sjaak Brinkkemper, and Garm Lucassen. 2018. “The Interactive Narrator Tool: Effective Requirements Exploration and Discussion through Visualization.” *CEUR Workshop Proceedings* 2075.

“Sourcetrail.” 2020. 2020. https://sourcetrail.com/.

Steinbeck, Marcel, Rainer Koschke, and Marc O. Rudel. 2019. “Comparing the Evostreets Visualization Technique in Two-and Three-Dimensional Environments a Controlled Experiment.” *IEEE International Conference on Program Comprehension* 2019-May: 231–42. https://doi.org/10.1109/ICPC.2019.00042.

Steinbrückner, Frank. 2012. “Coherent Software Cities: Supporting Comprehension of Evolving Software Systems.”

Steinbrückner, Frank, and Claus Lewerentz. 2010. “Representing Development History in Software Cities.” In *Proceedings of the ACM Conference on Computer and Communications Security*, 193–202. ACM. https://doi.org/10.1145/1879211.1879239.

Storey, M.-A.D., F.D Fracchia, and H.A. Muller. 1999. “Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization.” In *Proceedings Fifth International Workshop on Program Comprehension. IWPC’97*, 44:17–28. IEEE Comput. Soc. Press. https://doi.org/10.1109/WPC.1997.601257.

Sülün, Emre, Eray Tüzün, and Uğur Doğrusöz. 2021. “RSTrace+: Reviewer Suggestion Using Software Artifact Traceability Graphs.” *Information and Software Technology* 130 (September 2020): 106455. https://doi.org/10.1016/j.infsof.2020.106455.

Tian, Fangchao, Tianlu Wang, Peng Liang, Chong Wang, Arif Ali Khan, and Muhammad Ali Babar. 2021. “The Impact of Traceability on Software Maintenance and Evolution: A Mapping Study.” *Journal of Software: Evolution and Process*, 1–38. https://doi.org/10.1002/smr.2374.

Tory, Melanie, and Torsten Möller. 2005. “Evaluating Visualizations: Do Expert Reviews Work?” *IEEE Computer Graphics and Applications* 25 (5): 8–11. https://doi.org/10.1109/MCG.2005.102.

Trapp, Matthias, Tassilo Glander, Henrik Buchholz, and Jürgen Döllner. 2008. “3D Generalization Lenses for Interactive Focus + Context Visualization of Virtual City Models.” *Proceedings of the International Conference on Information Visualisation*, 356–61. https://doi.org/10.1109/IV.2008.18.

Unterkalmsteiner, Michael. 2020. “Early Requirements Traceability with Domain-Specific Taxonomies-A Pilot Experiment.” *Proceedings of the IEEE International Conference on Requirements Engineering* 2020-Augus: 322–27. https://doi.org/10.1109/RE48521.2020.00042.

Viana, Marcos, Andre Hora, and Marco Tulio Valente. 2017. “CodeCity for (and by) JavaScript,” 1–9. http://arxiv.org/abs/1705.05476.

Vidya Sagar, Vidhu Bhala R., and S. Abirami. 2014. “Conceptual Modeling of Natural Language Functional Requirements.” *Journal of Systems and Software* 88 (1): 25–41. https://doi.org/10.1016/j.jss.2013.08.036.

Walny, Jagoda, Samuel Huron, Charles Perin, Tiffany Wun, Richard Pusch, and Sheelagh Carpendale. 2018. “Active Reading of Visualizations.” *IEEE Transactions on Visualization and Computer Graphics* 24 (1): 770–80. https://doi.org/10.1109/TVCG.2017.2745958.

Wang, Bangchao, Rong Peng, Yuanbang Li, Han Lai, and Zhuo Wang. 2018. “Requirements Traceability Technologies and Technology Transfer Decision Support: A Systematic Review.” *Journal of Systems and Software* 146: 59–79. https://doi.org/10.1016/j.jss.2018.09.001.

Weninger, Markus, Lukas Makor, and Hanspeter Mossenbock. 2020. “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor.” *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, 110–21. https://doi.org/10.1109/VISSOFT51673.2020.00017.

Wettel, Richard, and Michele Lanza. 2007a. “Program Comprehension through Software Habitability.” In *IEEE International Conference on Program Comprehension*, 231–40. IEEE. https://doi.org/10.1109/ICPC.2007.30.

———. 2007b. “Visualizing Software Systems as Cities.” In *VISSOFT 2007 - Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 92–99. https://doi.org/10.1109/VISSOF.2007.4290706.

———. 2008. “Visual Exploration of Large-Scale System Evolution.” *Proceedings - Working Conference on Reverse Engineering, WCRE*, 219–28. https://doi.org/10.1109/WCRE.2008.55.

Wettel, Richard, Michele Lanza, and Romain Robbes. 2011. “Software Systems as Cities: A Controlled Experiment.” *Proceeding of the 33rd International Conference on Software Engineering - ICSE ’11*, 551. https://doi.org/10.1145/1985793.1985868.

White, Robert, Jens Krinke, and Raymond Tan. 2020. “Establishing Multilevel Test-to-Code Traceability Links.” In *Proceedings - International Conference on Software Engineering*, 861–72. https://doi.org/10.1145/3377811.3380921.

Wijk, J.J. van. 2005. “The Value of Visualization.” In *VIS 05. IEEE Visualization, 2005.*, 79–86. IEEE. https://doi.org/10.1109/VISUAL.2005.1532781.

Wolfenstetter, Thomas, Mohammad R. Basirati, Markus Böhm, and Helmut Krcmar. 2018. “Introducing TRAILS: A Tool Supporting Traceability, Integration and Visualisation of Engineering Knowledge for Product Service Systems Development.” *Journal of Systems and Software* 144 (June 2017): 342–55. https://doi.org/10.1016/j.jss.2018.06.079.

Zogaan, Waleed, Palak Sharma, Mehdi Mirahkorli, and Venera Arnaoudova. 2017. “Datasets from Fifteen Years of Automated Requirements Traceability Research: Current State, Characteristics, and Quality.” In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, 110–21. IEEE. https://doi.org/10.1109/RE.2017.80.

Chapter 10
APPENDICES

Appendix I: Some Recent Software Visualisation Works of Interest

This appendix presents some of the recent software visualisation works that are found interesting but were not covered in our literature review It serves to highlight current research directions in the field, and to give credit to some of the works that inspired and informed this research.

Table 10.1: Recent Software Visualisation Works that are of interest but not covered in literature review of this work.

Software Comprehension	[Introduces Runtime Trace to City Metaphor] Dashuber, Veronika, and Michael Philippsen. 2021. "Trace Visualization within the Software City Metaphor: A Controlled Experiment on Program Comprehension." Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021, 55–64. https://doi.org/10.1109/VISSOFT52517.2021.00015. [Research Impact of ExplorViz—an influential work in SV] Hasselbring, Wilhelm, Alexander Krause, and Christian Zirkelbach. 2020. "ExplorViz: Research on Software Visualization, Comprehension and Collaboration." Software Impacts 6 (September): 100034. https://doi.org/10.1016/j.simpa.2020.100034. [Word-Sized Graphics—Sparklines, Iconification] Hoffswell, Jane, Arvind Satyanarayan, and Jeffrey Heer. 2018. "Augmenting Code with In Situ Visualizations to Aid Program Understanding." Conference on Human Factors in Computing Systems - CHI '18. https://doi.org/10.1145/3173574.3174106.
VR/AR/Gamifying	[Empirical Support]: Moreno-Lumbreras, David, Roberto Minelli, Andrea Villaverde, Jesus M. Gonzalez-Barahona, and Michele Lanza. 2021. "CodeCity: On-Screen or in Virtual Reality?" Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021, 12–22. https://doi.org/10.1109/VISSOFT52517.2021.00011. Mehra, Rohit, Vibhu Saujanya Sharma, Vikrant Kaulgud, Sanjay Podder, and Adam P. Burden. 2020. "Towards Immersive Comprehension of Software Systems Using Augmented Reality - An Empirical Evaluation." Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, 1267–69. https://doi.org/10.1145/3324884.3418907. Schaller, Meike, and Andreas Schreiber. 2019. Visualization of Component-Based Software Architectures: A Comparative Evaluation of the Usability in Virtual Reality and 2D. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 11569 LNCS. Springer International Publishing. https://doi.org/10.1007/978-3-030-22660-2_14. Romano, Simone, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. 2019. "On the Use of Virtual Reality in Software Visualization: The Case of the City Metaphor." Information and Software Technology 114 (June): 92–106. https://doi.org/10.1016/j.infsof.2019.06.007. [Impact on feelings, emotions, and thought process] Romano, Simone, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. 2019. "The City Metaphor in Software Visualization: Feelings, Emotions, and Thinking." *Multimedia Tools and Applications* 78 (23): 33113–49. https://doi.org/10.1007/s11042-019-07748-1. [EvoStreets—2D vs 3D] Steinbeck, Marcel, Rainer Koschke, and Marc O. Rudel. 2019. "Comparing the Evostreets Visualization Technique in Two-and Three-Dimensional Environments a Controlled Experiment." *IEEE International Conference on Program Comprehension* 2019-May: 231–42. https://doi.org/10.1109/ICPC.2019.00042.

Merino, Leonel, Alexandre Bergel, and Oscar Nierstrasz. 2018. "Overcoming Issues of 3D Software Visualization through Immersive Augmented Reality." *Proceedings - 6th IEEE Working Conference on Software Visualization, VISSOFT 2018*, 54–64. https://doi.org/10.1109/VISSOFT.2018.00014.

Merino, Leonel, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel A Keim. 2017. "On the Impact of the Medium in the Effectiveness of 3D Software Visualizations." In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 11–21. IEEE. https://doi.org/10.1109/VISSOFT.2017.17.

[Focus on regaining lost Design Knowledge (common grounds with our work)]
Hoff, Adrian, Michael Nieke, and Christoph Seidl. 2021. *Towards Immersive Software Archaeology: Regaining Legacy Systems' Design Knowledge via Interactive Exploration in Virtual Reality*. *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Vol. 1. Association for Computing Machinery. https://doi.org/10.1145/3468264.3473128.

[Investigating Usability Issues]
Baum, David, Stefan Bechert, Ulrich Eisenecker, Isabelle Meichsner, and Richard Muller. 2020. "Identifying Usability Issues of Software Analytics Applications in Immersive Augmented Reality." Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020, 100–104. https://doi.org/10.1109/VISSOFT51673.2020.00015.

[Cartographic]
Schreiber, Andreas, Lisa Nafeie, Artur Baranowski, Peter Seipel, and Martin Misiak. 2019. "Visualization of Software Architectures in Virtual Reality and Augmented Reality." *IEEE Aerospace Conference Proceedings* 2019-March. https://doi.org/10.1109/AERO.2019.8742198.

[Notable Overview of Metaphors in VR]
Averbukh, Vladimir, Natalya Averbukh, Pavel Vasev, Ilya Gvozdarev, Georgy Levchuk, Leonid Melkozerov, and Igor Mikhaylov. 2019. *Metaphors for Software Visualization Systems Based on Virtual Reality*. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11613 LNCS. Springer International Publishing. https://doi.org/10.1007/978-3-030-25965-5_6.

[Interaction by Speaking]
Seipel, Peter, Adrian Stock, Sivasurya Santhanam, Artur Baranowski, Nico Hochgeschwender, and Andreas Schreiber. 2019. "Speak to Your Software Visualization-Exploring Component-Based Software Architectures in Augmented Reality with a Conversational Interface." Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019, 78–82. https://doi.org/10.1109/VISSOFT.2019.00017.

[Code House Metaphor]
Hori, Akihiro, Masumi Kawakami, and Makoto Ichii. 2019. "Code House: VR Code Visualization Tool." Proceedings - 7th IEEE Working Conference on Software Visualization, VISSOFT 2019, 83–87. https://doi.org/10.1109/VISSOFT.2019.00018.

Mehra, Rohit, Vibhu Saujanya Sharma, Vikrant Kaulgud, and Sanjay Podder. 2019. "XRaSE: Towards Virtually Tangible Software Using Augmented Reality." *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 1194–97. https://doi.org/10.1109/ASE.2019.00135.

[Interesting Approach to address Software Analysis]
Vincur, Juraj, Pavol Navrat, and Ivan Polasek. 2017. "VR City: Software Analysis in Virtual Reality Environment." *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 509–16. https://doi.org/10.1109/QRS-C.2017.88.

[Gamifying]

	Merino, Leonel, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. 2017. "CityVR : Gameful Software Visualization," 1–5. https://doi.org/10.1109/ICSME.2017.70.
Metaphors	[Notable New Metaphors] Mortara, Johann, Philippe Collet, and Anne Marie Dery-Pinna. 2021. "Visualization of Object-Oriented Variability Implementations as Cities." *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 76–87. https://doi.org/10.1109/VISSOFT52517.2021.00017. Atzberger, Daniel, Willy Scheibel, Daniel Limberger, and Jürgen Döllner. 2021. "Software Galaxies: Displaying Coding Activitiesusing a Galaxy Metaphor." *ACM International Conference Proceeding Series*, 5–6. https://doi.org/10.1145/3481549.3481573. Heidmann, Elke Franziska, Lynn Von Kurnatowski, Annika Meinecke, and Andreas Schreiber. 2020. "Visualization of Evolution of Component-Based Software Architectures in Virtual Reality." *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, 12–21. https://doi.org/10.1109/VISSOFT51673.2020.00006. [Extending CodeCity] Ardigo, Susanna, Csaba Nagy, Roberto Minelli, and Michele Lanza. 2021. "Visualizing Data in Software Cities." *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 145–49. https://doi.org/10.1109/VISSOFT52517.2021.00028. [Applying City Metaphor to Visualise Memory] Weninger, Markus, Lukas Makor, and Hanspeter Mossenbock. 2020. "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor." *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, 110–21. https://doi.org/10.1109/VISSOFT51673.2020.00017. [Repurposing City Metaphor to Suit Visualisation of Software Evolution] Pfahler, Federico, Roberto Minelli, Csaba Nagy, and Michele Lanza. 2020. "Visualizing Evolving Software Cities." *Proceedings - 8th IEEE Working Conference on Software Visualization, VISSOFT 2020*, 22–26. https://doi.org/10.1109/VISSOFT51673.2020.00007. [Extending Software Cities to Visualise Runtime Trace] Krause, Alexander, Malte Hansen, and Wilhelm Hasselbring. 2021. "Live Visualization of Dynamic Software Cities with Heat Map Overlays." *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 125–29. https://doi.org/10.1109/VISSOFT52517.2021.00024. [Applying CodeCity to Expose Security Vulnerabilities] Sinhabahu, Nadun, Prasad Wimalaratne, and Chaman Wijesiriwardana. 2020. "Secure Codecity with Evolution: Visualizing Security Vulnerability Evolution of Software Systems." *20th International Conference on Advances in ICT for Emerging Regions, ICTer 2020 - Proceedings*, no. ICTer: 302–3. https://doi.org/10.1109/ICTer51097.2020.9325429.
Visual Enhancements	Limberger, Daniel, Willy Scheibel, Jürgen Döllner, and Matthias Trapp. 2019. "Advanced Visual Metaphors and Techniques for Software Maps." *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3356422.3356444. [Level of Detail—LoD, Degree of Interest—DoI, and Aggregation] Limberger, Daniel, Willy Scheibel, Sebastian Hahn, and Jurgen Dollner. 2017. "Reducing Visual Complexity in Software Maps Using Importance-Based Aggregation of Nodes." *VISIGRAPP 2017 - Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications* 3: 176–85. https://doi.org/10.5220/0006267501760185.
Stable Layouting	Tua, Davide Paolo, Roberto Minelli, and Michele Lanza. 2021. "Voronoi Evolving Treemaps." *Proceedings - 2021 Working Conference on Software Visualization, VISSOFT 2021*, 1–11. https://doi.org/10.1109/VISSOFT52517.2021.00032.

	Scheibel, Willy, Christopher Weyand, and Jürgen Döllner. 2018. “EvoCells - A Treemap Layout Algorithm for Evolving Tree Data.” *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications* 3 (Visigrapp): 273–80. https://doi.org/10.5220/0006617102730280. Hahn, Sebastian, and Jürgen Döllner. 2017. “Hybrid-Treemap Layouting.” *EuroVis ’17: Proceedings of the Eurographics/IEEE VGTC Conference on Visualization*. https://doi.org/10.2312/eurovisshort.20171137.

Appendix II: Other Secondary Methods to Initiate the Tagging

Below is a description of the secondary tagging mechanisms referred to in section 6.7.4

1- CodeLens Quick Action

This method utilises the CodeLens feature of Vscode to enable the developer to quickly spot the eligible code items (see Valid Document Symbols in Table 6.2) across an open source file that are not yet tagged. The feature can be enabled on demand through the command palette or a shortcut. Once enabled, the developer will see small CodeLens actions displayed on top of those code items for which tagging is recommended. By clicking on it, the process is initiated for that particular code item, and follows as described in the third method above.

2- List of Untagged Code Items in Recent Changes

This approach presents the developer with a quick list to go over the untagged code items that are found in their recent changes of the codebase. The method employs a specially developed technique to scan the recently changed parts of the source code across a user's opened workspace[136], and identifies all the code items that are not tagged, but are eligible to be so. It then presents those in a simple tree list view organised by file, which the user can review and take action on where desired.

3- List of Untagged Code Items in Staged Changes

In the same fashion as the mechanism applied to recent changes above, a similar approach is employed to identify the untagged code items in the source code changes that a developer has staged and is about to commit. This is in fact the same list of the untagged code items that was featured in the on-commit initiated method). Those items are then presented in a similar tree list view as recommendations for the user to take action on.

4- List of All Untagged Code Items

This last method presents a simple but similar list view to the two above, except that it is a comprehensive list: it scans the entire opened workspace (or project folder) and identifies all untagged code items for which tagging is recommended[137]. It then presents those in a collapsible tree list

[136] The workspace must have a git repository configured for this feature to work. All tagging functionalities require at least a local git repository to work.

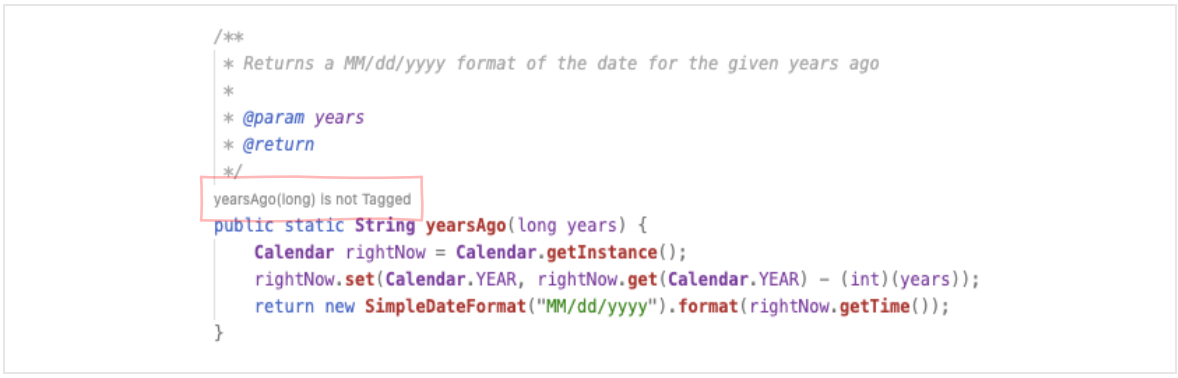
[137] At this stage, tagging recommendation is simply determined based on the code item type. In future work, advanced recommendation criteria can be developed.

organised per file. The user can then initiate a tagging action on any desired item, which will launch the same workflow described in the CI Initiated method.

While this particular list is unlikely to be useful for a regular developer's daily routine, it is intended for quality review sessions where the user is particularly interested in monitoring the untagged parts of their codebase. More discussion on this option appears in the Evaluation Chapter.

Figure 10.1 below illustrates all four secondary methods introduced above.

It is worth noting that the three list views described above—called viewlets in Vscode vocabulary—can be accessed at any time, and the user can have them docked anywhere in their Vscode editor to keep a monitoring eye on them. Their content is automatically refreshed upon detection of relevant changes. Of more interest to note is that detection of the untagged code items is in fact processed at the file parsing stage, which typically happens at the beginning of environment loading or workspace opening. As the whole source code model is always kept up to date (as described in Chapter 6), the untagged code items are consequently always kept live and up to date. Detection of those in recent or staged changes requires nonetheless a further computation, which is triggered upon such changes being detected on the git repository.



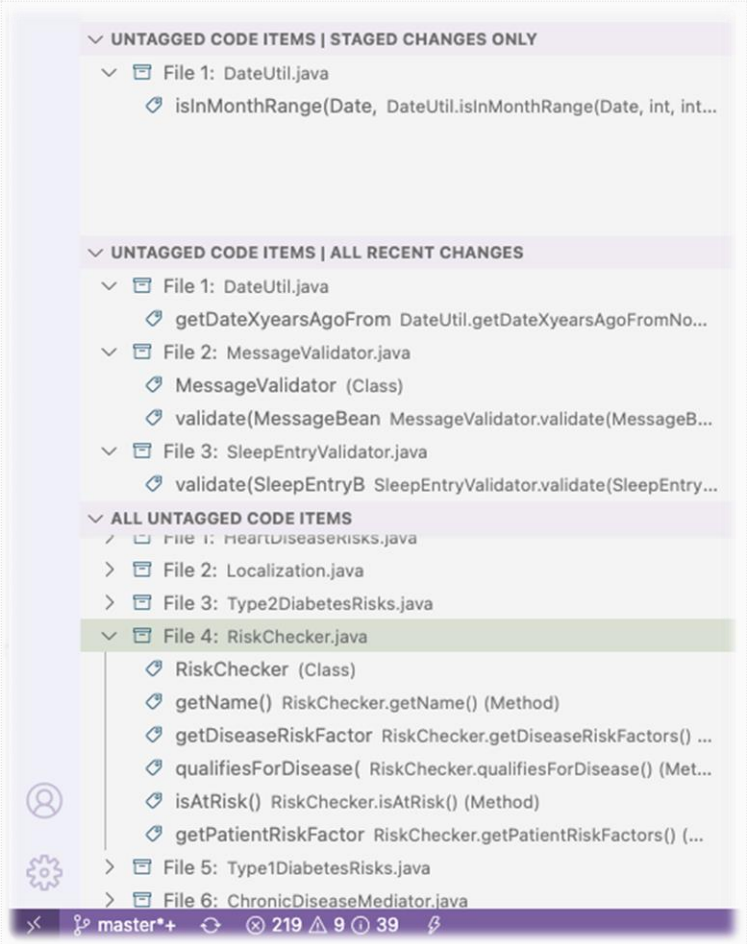
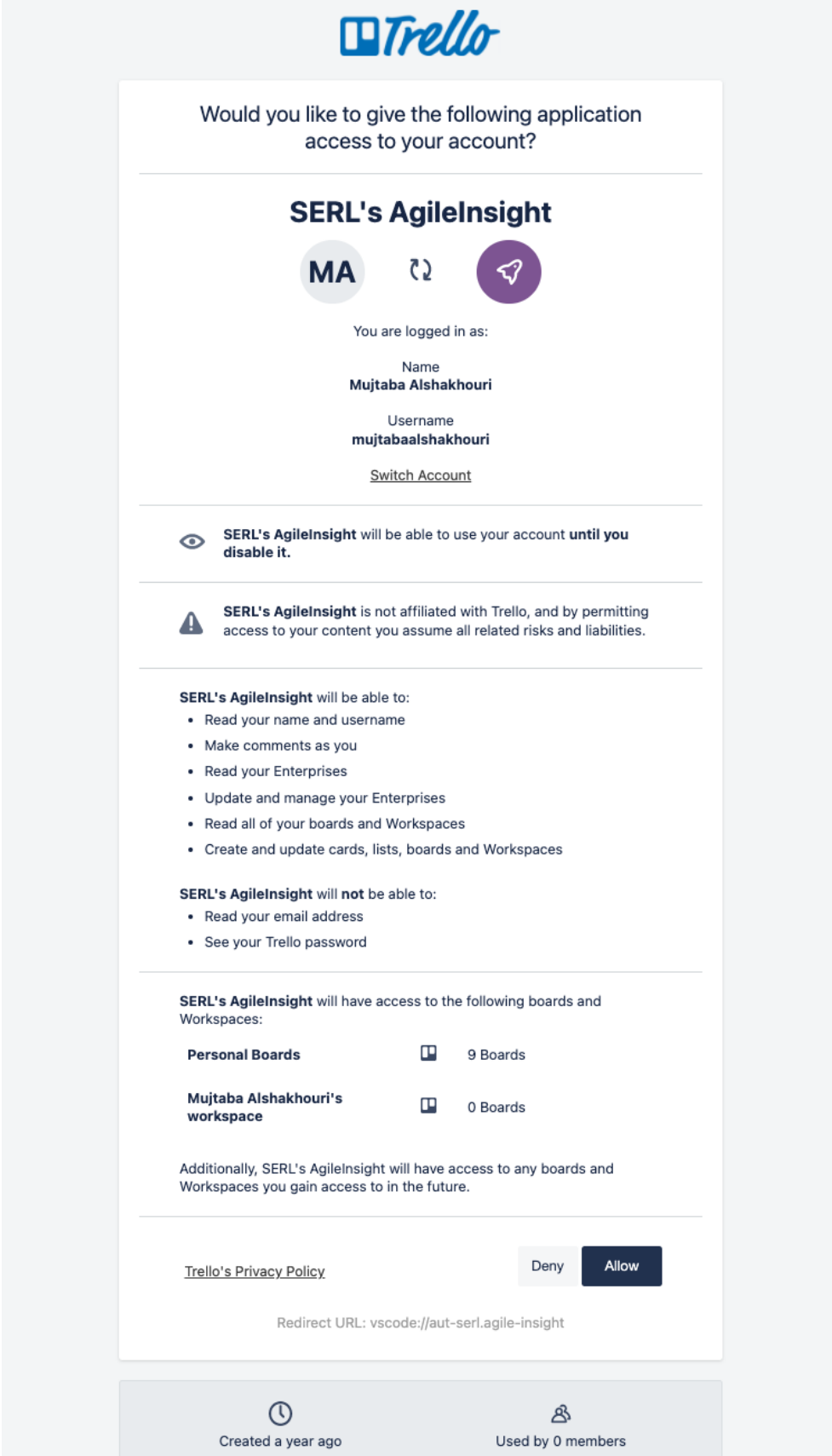
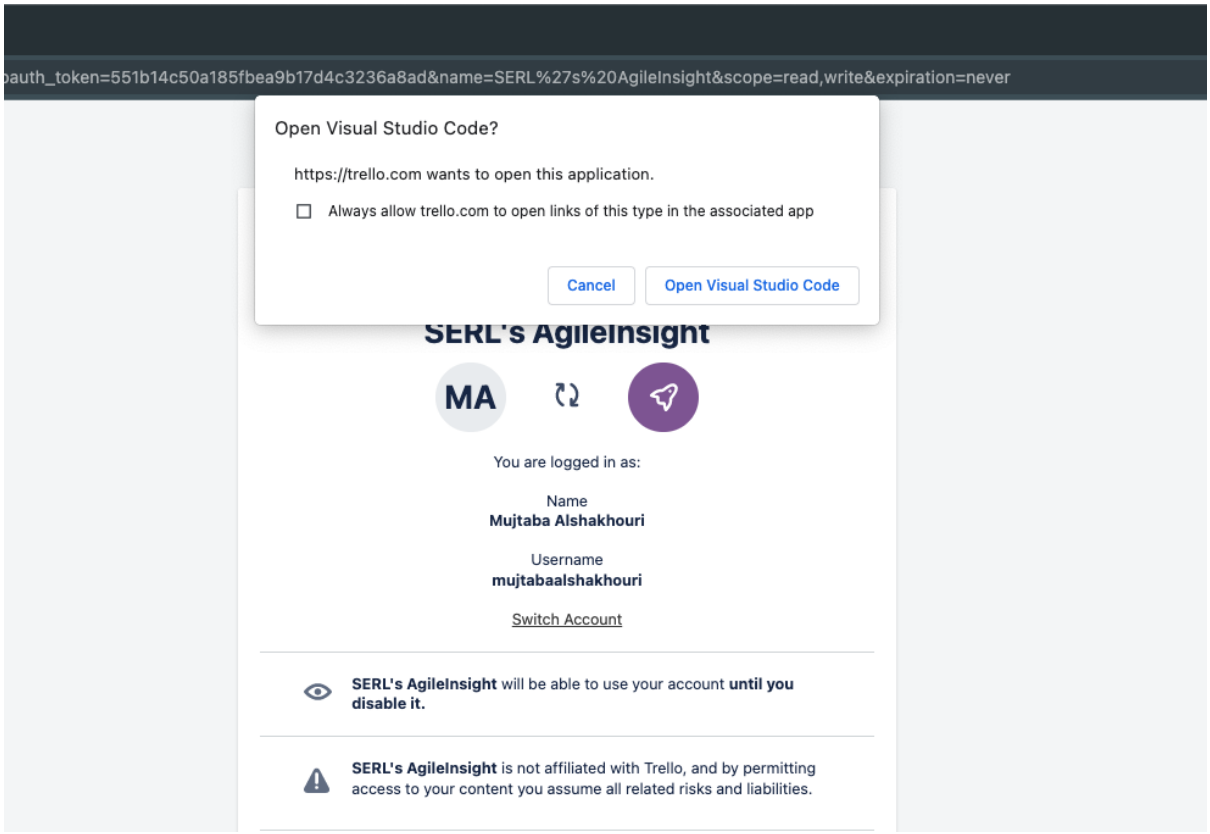
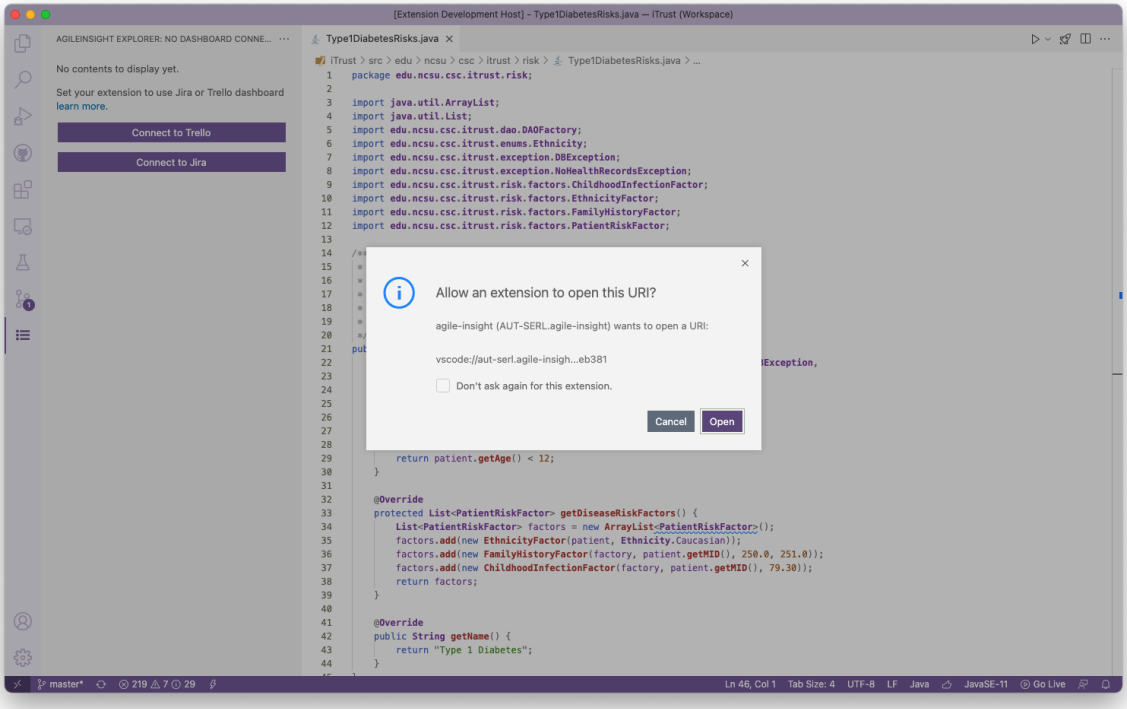


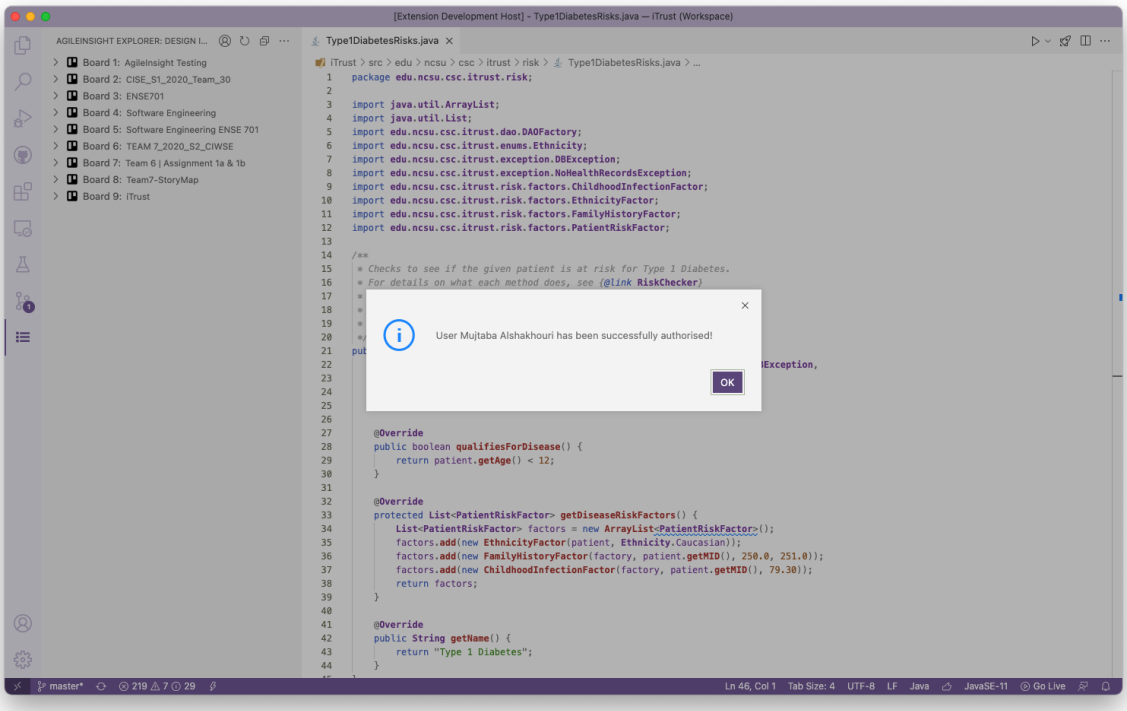
Figure 10.1: The top row illustrates the CodeLens tagging feature as it appearing in source code when it is enabled. The lower row illustrates the list views of the three untagged code items categories introduced above.

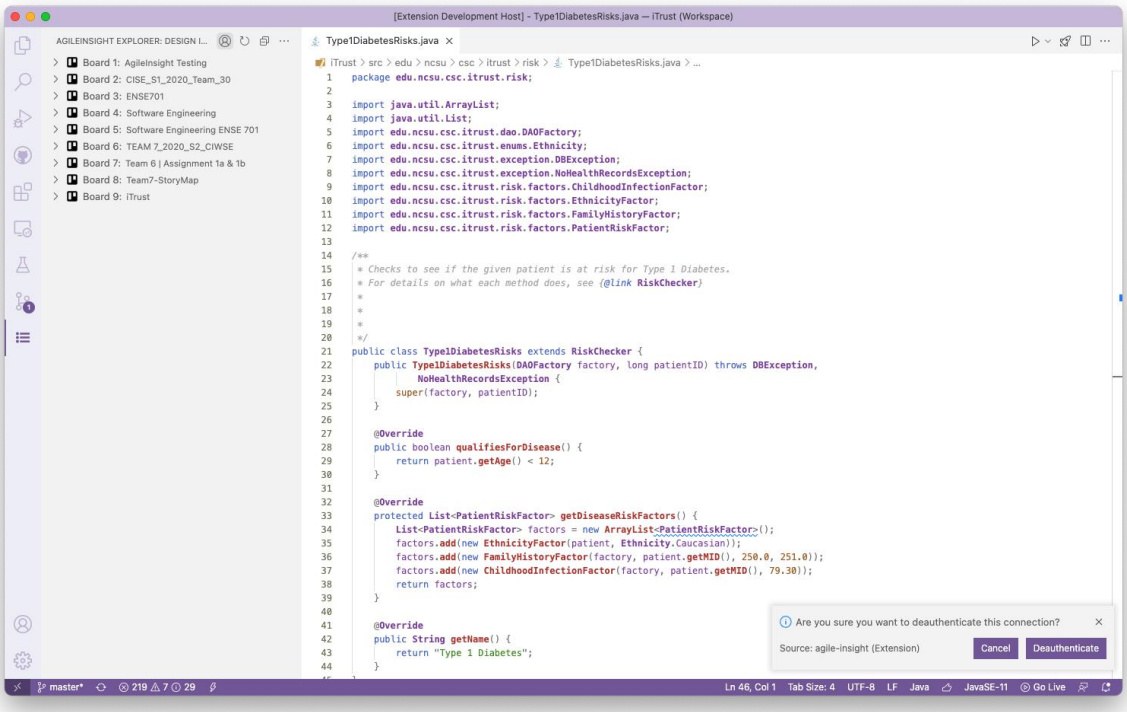
Appendix III: OAuth Workflow Demo




auth_token=551b14c50a185fbea9b17d4c3236a8ad&name=SERL%27s%20AgileInsight&scope=read,write&expiration=never
Open Visual Studio Code?
https://trello.com wants to open this application.
Always allow trello.com to open links of this type in the associated app
Cancel
Open Visual Studio Code
SERL's AgileInsight
MA
You are logged in as:
Name
Mujtaba Alshakhouri
Username
mujtabaalshakhouri
Switch Account
SERL's AgileInsight will be able to use your account until you disable it.
SERL's AgileInsight is not affiliated with Trello, and by permitting access to your content you assume all related risks and liabilities.


[Extension Development Host] - Type1DiabetesRisks.java — iTrust (Workspace)
AGILEINSIGHT EXPLORER: NO DASHBOARD CONNE...
No contents to display yet.
Set your extension to use Jira or Trello dashboard
learn more.
Connect to Trello
Connect to Jira
Type1DiabetesRisks.java
Allow an extension to open this URI?
agile-insight (AUT-SERL.agile-insight) wants to open a URI:
vscode://aut-serl.agile-insigh...eb381
Don't ask again for this extension.
Cancel
Open


[Extension Development Host] - Type1DiabetesRisks.java — iTrust (Workspace)
AGILEINSIGHT EXPLORER: DESIGN I...
Board 1: AgileInsight Testing
Board 2: CISE_S1_2020_Team_30
Board 3: ENSE701
Board 4: Software Engineering
Board 5: Software Engineering ENSE 701
Board 6: TEAM 7_2020_S2_CIWSE
Board 7: Team 6 | Assignment 1a & 1b
Board 8: Team7-StoryMap
Board 9: iTrust
Type1DiabetesRisks.java
User Mujtaba Alshakhouri has been successfully authorised!
OK
master*
Ln 46, Col 1
Tab Size: 4
UTF-8
LF
Java
JavaSE-11
Go Live


[Extension Development Host] - Type1DiabetesRisks.java — iTrust (Workspace)
AGILEINSIGHT EXPLORER: DESIGN I...
Board 1: AgileInsight Testing
Board 2: CISE_S1_2020_Team_30
Board 3: ENSE701
Board 4: Software Engineering
Board 5: Software Engineering ENSE 701
Board 6: TEAM 7_2020_S2_CIWSE
Board 7: Team 6 | Assignment 1a & 1b
Board 8: Team7-StoryMap
Board 9: iTrust
Type1DiabetesRisks.java
Are you sure you want to deauthenticate this connection?
Source: agile-insight (Extension)
Cancel
Deauthenticate
master*
Ln 46, Col 1
Tab Size: 4
UTF-8
LF
Java
JavaSE-11
Go Live

Appendix IV: Dataset Research and Selection

This section is dedicated to covering the effort and findings with regard to obtaining a suitable traceability dataset from current literature. It discusses examined literature, assessments and findings on a number of datasets that were evaluated for their suitability, and finally discusses the selected dataset (iTrust) and the approach used for preparing for AgileInsight evaluation.

Traceability Datasets—Availability in Literature

While studying the availability of datasets in traceability research is definitely not an objective or a focus of this research, a limited examination effort had to be conducted in order to find a proper dataset for the purpose of this evaluation phase. In this discourse, a total of four research works and dataset repositories were examined. As a convenient reminder, the core criteria for the suitability assessment is the availability of three datapoints:

- Original functional requirements in small format (e.g., user stories, features, etc.)
- Product Source Code
- Tracelinks between original requirement artefacts to source code artefacts, preferably at lower granularity level (e.g., method levels).

1. COEST's 15 Datasets—2017[138]

The Center of Excellence for Software and System Traceability (CoEST) is a well-known source of requirements traceability research and traceability datasets (Cleland-Huang et al.). Datasets from COEST have also appeared in a number of important works and literature (Bella et al. 2018; Hayes, Li, and Rahimi 2014). It was thus natural for this repository be the first to be considered. At the time of access, the repository at COEST[139] had a total of 15 datasets. All 15 datasets were examined to assess their suitability. The following is the result of the assessment:

[138] The year appearing in headings indicate either the paper's publishing year where that the datasets were made available, or the date of datasets were made available in a public repository.
[139] http://coest.org/

- Four datasets were excluded right away as they were designated as small and 'student-created' datasets. A real-world dataset from industry was naturally preferred with ideally medium-to-large size in order to represent an adequate level of complexity.

- The remaining eleven datasets were then downloaded and examined carefully

    - All have passed the condition of requirements and tracelinks availability

    - Seven were eliminated, however, as they did not offer the source code

    - Another two were further eliminated as they had their source code files in plain text files and with no readily identifiable hierarchical structure (folder/package)

    - One last dataset, Albergate, was also eliminated as it had a very small codebase (55 source files only), and the requirements were only made available in the Italian language.

    - iTrust was the only remaining dataset, and despite a few shortcomings[140], it nonetheless offered the best matching dataset. It consisted of 226 Java source files in their proper packaging structure. Additionally, it had 141 Jsp source files, and the tracelinks were available to both the Java and the Jsp files. Unfortunately, it only offered file-level tracelinks though. More discussion on this dataset follows in coming sections.

One particular finding that is worthy of highlighting here is that while the COEST repository has been instrumental to a number of traceability research, it was noticed that overall, the codebases were of relatively small size and fell short in comparison with real-world applications with regard to representing the complexity seen in the industry. This remark was also reported by Charalampidou et al. in their 2021 mapping study (Charalampidou et al. 2021).

2. Datasets From Fifteen Years—2017

In their recent paper titled 'Datasets from Fifteen Years of Automated Requirements Traceability Research: Current State, Characteristics, and Quality', the authors embarked on an extensive effort to evaluate datasets that were used in requirement traceability research over a 15 year period—since the paper's publishing date of 2017 (Zogaan et al. 2017). Their systematic literature review is claimed to be unique in literature in that it reviews the current status of those datasets and provides a framework of categorising and assessing their characteristics, their threats to validity, and the research context where

[140] For instance, it was still a 'student-created' dataset. However, it is comparatively well-maintained over a series of academic projects.

each could be suitable for. Their systematic literature presents an excellent review and assessment of 73 datasets, 44 of which are purportedly available online. Unfortunately, despite the classification effort, the study does not specify the size of each of those datasets (number of source code files, or both kind of source/target artefacts in general), nor the programming language of their source code—both elements would have been helpful indicators to other researchers in a process of selection. Nonetheless, both size and language are acknowledge among the set of factors impacting threats to validity. More importantly, the classification does not provide the status or availability of the tracelinks for each of those datasets. The dataset Easy-Clinic is discussed as an exemplary case to illustrate their introduced framework, however, the dataset has a very small size of source code files—37 classes only.

Given the above situation, only a selected number of the 73 datasets were examined to manually assess their suitability. A good number of the codebase were well-known from prior experience/work and could be excluded immediately. Out of the 18 industrial datasets (which would have offered a preferred option), only five were available, but unfortunately none have met the three conditions required—especially the tracelink datapoint. Out of the remaining set that were quickly examined (six in total), none have also met the three conditions, except for the dataset iTrust.

3. The Dronology Dataset—2017, On-going

Among the prominent and recent efforts in the traceability research comes the Dronology project (Cleland-Huang, Vierhauser, and Bayley 2018; Cleland-Huang and Vierhauser 2018) from the Systems and Requirements Engineering Center (SAREC) at University of Notre Dame. The project is aimed at delivering a safety-critical system for controlling drones. Taking advantage of its academic birth place, the project was designed to adopt a specific approach of development from its inception point, in that traceability links are strictly maintained between the code artefacts and their original requirement artefacts. The project also adopts an agile approach with special adaptations[141] to ensure it meets the safety requirements and mandated standards. In fact, there is a good similarity of the approach used to maintain the tracelinks in the Dronology project and this work—developers use identifiers of design definitions to manually link the definitions to their implementation source files. The design definition identifiers refer to Jira issues that are also used to tag Github commit messages. The approach relies on separate JSON files to maintain and keep track of the tracelink mappings.

[141] For example, user stories are adapted and are written in EARS (Easy Requirements Specification Format).

Considering those similarities, this has made the project to form an interesting potential opportunity to use in this evaluation phase, if a full dataset could be obtained. According to the project's site[142], the group is aiming to make such dataset available in the near future as a series of releases. At the time of writing, two versions of early datasets have indeed been made available, but they do not include the source code. A preliminary version where source code was made available appeared in (Goodrum et al. 2017), but given its early development stage at the time, the source code included only 39 Java files in its latest provided version (v0.6).

While the dataset would have formed a great opportunity, given the user story format of the requirements and the full traceability links maintained since its inception, the project only maintains file-level traceability though—which is an unfortunate disadvantage to the case of this evaluation, as lower-(and varying) level of tracelinks are a an important aspect of this work and its concept.

4. The SEOSS 33 Dataset—2019

This dataset published very recently by the Data in Brief Journal in 2019 represents probably one of the largest effort to date—to our best knowledge—with regard to amassing traceability datasets and making them available for researchers. The authors have undertaken a considerable effort to assimilate quality datasets of 33 open-source systems, and present them in well-organised and structured manner. The dataset is hosted by Harvards' Dataverse143, and is offered in a SQLite format. While Zogaan et al. (Zogaan et al. 2017) have produced earlier an excellent initiative by classifying a large number of datasets, Rath and Mäder have took the work one big leap ahead by scanning large corpus of software repositories to assimilate and make the datasets available (Rath and Mäder 2019).

In brief, all 33 datasets do match our 3 conditions, but only provide file-level mappings. The authors crawled Jira and Github repositories to extract and build the datasets, which provide Jira issues, Github commit changesets, and tracelinks that map issue identifiers to GitHub commits. They relied on what they called 'common practice' that developers would often tag their commit messages with issue identifiers, and thus built a tool to crawl those messages to extract and build the mappings. In other words, they create mappings from Jira issues to version control change blurbs. From the change blurbs (or changesets), a mapping is then offered to individual files that were modified in that commit action. While these are the key datapoints of interest to this work, their datasets include many other useful

[142] https://dronology.info/
[143] https://doi.org/10.7910/DVN/PDDZ4Q

information to other contexts and purposes. Their paper offer a helpful diagram to better understand their dataset object model and mappings.

While this datasets presents an exciting outlook to undertake an empirical evaluation of AgileInsight in the future, given that the mappings offered at this time are only file-level, and more importantly, that the mappings to individual files are indirect, it was decided not to use this dataset in this phase of the evaluation. It would require a non-trivial effort to prepare one of their datasets for use by AgileInsight, especially to extract the file-level mappings from the source control changesets. Nonetheless, it certainly offers an excellent opportunity for future work.

> Remark. While at the early stage of this work, it was not clear how common the practice is of developers tagging their commit messages with issue identifiers, the fact that the practice is gaining more adoption by the community is seen to add strength to the proposed concept underpinning this work. It also supports the findings from our expert interviews that developers do want to perform the tagging, but they are put back by having to do it manually, and by not seeing immediate value in return. Offering them a tool that takes that burden of their shoulders by automating the tagging, making it systematic, and by making the value immediately available to them in their IDE, is seen to promote the practice further and works towards standardising it. In fact, AgileInsight could fill in the current gap, where as such works illustrate, researchers have to exert considerable effort in order to extract and build those tracelinks in post manner, before the data could be put to use and benefit by the user.
>
> Nonetheless, the status of the practice of tagging the commit messages seems to remain contested by researchers, and our expert interviews have certainly revealed that not all teams adopt it, and among those who do, not all members would observe it regularly. In fact, in their 2020 paper defending their machine learning approach to build tracelinks in retrospective and post-development manner, the authors states that "…due to the time pressure, developers typically do not deal with the traceability links during their development as it does not bring short-term benefits… traceability links are often not updated and maintained in time and thus typically outdated or even missing. It is essential to recover the traceability links between software artifacts.." (Li et al. 2020).

5. International Conference on Mining Software Repositories (MSR)—2021

In our last attempt in obtaining a dataset with method-level traceability links, we examined the proceedings of the latest conference on Mining Software Repositories (MSR). Out of their 16 Data Showcase papers, only one was treating the subject of requirement traceability. Fortunately, this paper did offer the missing element that was being sought.

In their recent dataset paper titled "A Traceability Dataset for Open Source Systems", Hammoudi et al. have particularly focused on providing method-level tracelinks (Hammoudi et al. 2021). As far as our limited investigation has revealed, and indeed as far as the authors' paper seems to suggest, their dataset appears to be the only one available so far that makes tracelinks available at this level of granularity. Their motivation was that method-level tracelinks provide more accuracy in 'pinpointing' the parts of source code that implement a given requirement. Another distinguishing feature of their dataset is that they adopt a fully deterministic approach in creating their tracelinks. Almost all other datasets would provide the tracelink mappings as an R x C matrix (where R stands for requirement and C for code item) but they would only provide mapping relations where such a link is actually identified. The mapping does not say anything about code items or requirements where such a link was not identified. In their work, Hammoudi et al. have chosen to create instead a full matrix linking each single method to each requirement artefact—and deterministically labelling that link as either a Trace (T), a non-trace (N), or undefined (U). They claimed that this approach could provide more certainty to software engineers, enabling them to distinguish between when developers confirm no trace between two artefacts and when they were simply unsure. Their dataset offered this information for four open source systems in JSON format and a corresponding SQL script. One of those datasets was iTrust.

Interestingly, the authors had to resort to paying the original developers and a team of students in order to provision the method-level tracelinks for three of the systems. For iTrust, they indicated that the method-level tracelinks are readily available on the project's website[144].

Conclusion. Out of the four datasets offered by the authors, iTrust displayed the most complexity—having the highest number of classes and the highest number of tracelink mappings. Unfortunately, the fact that the dataset focused only on method-level tracelinks and not other granularities, meant that relying on this dataset alone would not allow us to evaluate and demonstrate the multi-granularity tracelink capability of AgileInsight. As explained earlier, this research introduces a concept of generalised actionable requirements (design items) and generalised code items. It consequently enables tracelinks between the two artefacts to be created at any granularity level (e.g., file-level, class-level, method-level, or function-level). It was hence important to demonstrate and evaluate this capability implemented by AgileInsight. As discussed in the next section, this has led us to adopt a

[144] Unfortunately, upon quick inspection, parts of the website were found to be not publicly accessible, including the Github repository containing the source code (only the one on sourceforge.net was available). Of the publicly accessible section of the site, only a single requirement page was found to actually contain information about the tracelinks. (At time of writing, the website was further found to be no longer accessible).

compromise approach by utilising both datasets of iTrust—the one from the CoEST repository by Cleland-Huang et al., and this one by Hammoudi et al.

Closing. In closing this section, an overarching finding that is of particular interest is that all datasets encountered were exclusively confined to the Java language[145]. While this could be a natural result due to the language's popularity, the latest indicators from development communities suggest that other languages are quickly taking over in popular use. Representing other languages in research datasets will certainly be of importance and interest to researchers in the near future. It would certainly be of particular interest to this work as well, as it would have allowed us to demonstrate and evaluate the traceability and the visualisation mechanisms of AgileInsight outside the Java language—which has strongly dominated software visualisation research, and apparently so for the late research in traceability[146]. Another remark to be highlighted, is that the scarcity of the method-level tracelinks in datasets—and the fact that researchers had to pay developers to manually reconstruct this knowledge—appears to suggest that it is impractical and infeasible for developers to declare tracelinks at this lower level of granularity using currently existing tools. This is an area where AgileInsight (through its IDE-integrated active collection mechanism) could possibly work towards enabling such functionality.

[145] The only uncertainty was around the list provided by Zogaan et al. 2017, as they did not include the language datapoint in their classification. But through the naming and prior experience, at least the majority could be confirmed as Java systems.

[146] The Dronology dataset appears to utilise both Java and Python languages, and it would make an interesting opportunity when the dataset becomes available with source code.

Appendix V: Full Script of Demonstration Scenarios

V.I Scenario One—Tasks and Script

On-Commit Tagging: A developer who is active in a development session, has just finished implementing a piece of requirement (or part of) and is about to commit their changes. They are shown instead AgileInsight's tagging functionality, which captures the tracelink knowledge from them and—as an added bonus—takes care of the commit operation for them as well.

> *"Let us say you had this UC card to implement [card shown on trello] and that you just completed writing this class with these two methods as part of your implementation [pointing to source code block in an open file]… you have also done your unit testing. Now when you are just ready to commit your changes… imagine you can do this [show, then click the hover tag action], right at your place, to tag what you have written of source code with the Id of the issue that you have just completed. Here… you now get this list (show Design Items Palette) where you can lookup and select your issue (UC, in this case)… You can also add your commit message here… and it is done (vscode shows tagging progress and displays feedback notifications). Now, your code item is permanently tagged in your repository with this issue you completed, which could be partially for now, that is fine. Your Jira (or Trello) ticket itself has also been tagged with this code item Id [show to participant]. Above here [pointing to source code block]... you see a tag has also been inserted right above the method name… and what is good is that your commit operation has been taken care of so you do not need to bother yourself about it. Here, let me show you what has just happened [participant is shown what has happened behind the scenes… the tags inserted automatically in 4 places].* **So, no need to switch tabs, do a stage, and then a commit. Just click this hover, enter your issue Id, and you are good to go…** *You have now created visible traceability in your source code, in your repository messages, and right in your ticket or card at the board. By doing this, we think you are also displaying transparency by letting your team know what parts of your code fulfils which requirement."*

Conversation Starter:

> *"We wanted this kind of tagging to be as low effort as we could so that developers can do it as part of their work routine without seeing it as an extra overhead… ideally we would expect the person who wrote the code to do the tagging as she or he are the origin of this information. However, we thought the tagging could still be done by a colleague doing a code review, or completing a unit test. What do you think?"*

A Variant Tagging Workflow: Participant is shown an alternative workflow to invoking the same tagging operation above, but this time, it is launched from the design items perspective—i.e., initiated from a dashboard card from within vscode.

> *"Ok, now let us say you do not want to do this hover thing for each code item you have written… then you can come here to the Dashboard Viewlet, go to your card, and click this action. You now get this list [showing Code Items Palette] that is similar to that one we have seen earlier, but instead you now get to select your code items… to get them tagged all at once. And of course your changes will be committed in the process as well, just like the in the previous case."*

Feature Location: Taking advantage of design item integration in IDE, a developer uses keywords to lookup original feature in code, and is able to locate the exact code items implementing it. Developer uses the information to identify areas of interest in the codebase that need to be modified as part of a new enhancement.

> *"Now, to finish off your issue implementation, let us say you realise you need to add a piece of code to a functionality developed by another team member. You know of a related keyword… but you are not sure of the exact feature details, nor which parts of the code that implements it. In this case, you could come to the dashboard viewlet over here, use the keywords to lookup the issue or feature you are after… [keywords are used to search the dashboard cards for feature of interest[147]] and ahh… here,… we found a few matching cards. Scanning the results, you identify the issue you are looking for. The tooltip gives you further convenience to read the full description on the spot. Now you can click on this quick action to bring up the related code items viewlet where you get to see and interact with the related code pieces as a list. You can navigate directly to the code of each if you want."*

Making Use of Visualisation to Reveal Feature Impact: A developer invokes a command on a selected feature resulting in revealing its impact on the code using AgileInsight's visualisation feature.

> *"Instead of that simple list view, we have this visualised mechanism as well that show you where that feature is actually implemented across the codebase. We can invoke it for any of the issues or features on the dashboard view through this Reveal Visual Impact action…[action is invoked] and you get to see this visual view exposing the locations of interest in your codebase… [Visualisation*

[147] The implementation enables a user to lookup a feature using its identifier or any keyword appearing in its title or description

view is automatically launched with buildings corresponding to code items of interest being visibly highlighted and animated]. So, this scene shows you where the feature is exactly implemented… like in which modules and in which files your code of interest is located. The sizes, and colors of the buildings give you an initial sense of what you are dealing with… I mean like level of complexity. OK, let us say you decide to inspect this module as it contains most of the implementation… we can then bring up a focused view of it separately, for easier inspection… we can go as deep as we need in the focused view. Now, let us say you decide you want to look at the code of this method, then we can also navigate to its code directly from the visualisation."

In-Context Change Impact Analysis: A developer who is about to modify a piece of code, carries out change impact analysis, identifying related design items and accessing key information on the spot without leaving the code.

"After examining the method's code here in the editor, imagine you decide to modify it… but looking here at the top of its declaration… you notice it is tagged with two other issues [showing the tags attached to the method name in the code editor]… Maybe you are now concerned your changes might affect them, in this case we could hover over the issue Id to reveal quick information about it… such as its title and a short excerpt from its description. The information could help us to rule out an a possible impact right away or if further look is needed. Or maybe, we can see that that its status is 'InProgress' so maybe we want to have a quick chat with the person currently working on it as they might still add changes later… we can find their name here at the bottom. You head to the second tag, and this time reading the quick details you are still not sure about impact. In this case, we could navigate to the full card on the board by clicking its title. This could offer more information to help with our analysis, providing more confidence to proceed with the changes."

Tagging Reminder: A developer modifies their code introducing new code items. Upon attempting to commit their changes, AgileInsight detects untagged eligible code items in their stagged part of the code, and they are prompted to tag them. The commit action is then automatically taken care of.

"Let us say you proceeded with your changes to that method, and you also ended up writing another new one. You have also done a quick unit test… and you think it is time to commit your changes. You head to the Git View (vscode's internal Git extension), hit the stage button, and start typing your commit message… and uh, you now get this alert reminding you that one of your stagged changes is untagged, prompting you to do so if you would like. The prompt would only be triggered for eligible code items in you staged changes, and only if they were not tagged. If you accept the prompt, it

takes you to this viewlet where all those identified code items are listed so you can choose which to tag [showing the stagged untagged code items viewlet]. You would not need to go again do the commit separately, the process will take care of that once you choose to tag any of those items. Assuming your changes belong to a single issue, you can in this case initiate the tagging from the card in the dashboard view, where you can select all code items of interest at once.

Do you think is helpful? Do you find it intrusive?"

In-Context Card Workflow: A developer completes their implementation of a particular issue, and pushes the card to the next stage on the spot without leaving their code.

"Good…, now after the TTO operation is complete, you see your issue Id is inserted above this method name. Let us say your changes completes the implementation of the issue, and you want to push your card down the workflow to the testing guy. We can hover over the issue Id here to bring the quick details tooltip, where we can perform some hot actions without leaving our spot. Here, you see that the card is in the 'InProgress' stage, so we can just push it forward down the workflow with this hot action here (the tooltip displays dynamic actions based on card's status)… now it i's with the testing guy."

Timed Tagging Reminder: Participant is shown an AgileInsight feature where a tagging reminder detects untagged eligible code items and displays a prompt according to a configured time period. Participant's thoughts are solicited on the feature's potential benefit for encouraging tagging practice.

Let us say that having seen some benefits of having your codebase tagged, you liked the idea and believe it could bring some good advantage to your team. As a team lead, you want to motivate your colleagues to start adopting the approach and develop the habit of regularly tagging their contributed code. The earlier on-commit reminder could be one way to encourage tagging practice. But imaging there is this developer who does not commit often… and they have already been coding for a while. In this case, we thought of this timed reminder that will pop up every once in a while (configurable) to help developers remember to tag their code. Ok... here it is… [timed recent untagged changes prompt is triggered], so it says 3 recently changed code items are found to be untagged. If the prompt is accepted, then the user is taken over here to this viewlet where all the recently contributed code items that need to be tagged are listed. They can then take action on any one, as desired. The viewlet is also accessible at all times… so a developer can come here anytime to check for recommended items to tag. Do you think this is useful?

V.II Scenario Two—Tasks and Script

Conversation Starter:

> *"After seen some benefits of having our code tagged with its original requirement, would you be happy if your team considered starting to tag your existing code on a slow gradual pace?... Ok, what do you think of tagging being done during a code review session? Or a by a quality engineer who is tasked with doing quality check?"*

Basic Post-Development Tagging: A developer wishing to tag previously contributed code is shown a basic mechanism where code items recommended for tagging are highlighted across a source file that is currently open in editor. Code Lens feature is utilised to enable in-context tagging action.

> *"So, to start… let us assume you have already checked out the branch of your repository that contains the source code files you want to tag… so you have the files accessible in your vscode workspace. At a basic level, there is this simple method that could be used if we wish to go through scanning the source code and tag whatever parts that one has contributed. See, once we enable this code lens feature [Code lens is activated through command palette] then it sort of offer a visual guide highlighting code items of interest as we go throughout a file."*

> *"Ok, here… imagine you spot this method that you have contributed few weeks ago. You read it quickly to refresh your mind of what it was for… and ok maybe you could remember a couple of keywords of the original issue or user story it belonged to. By hitting the code lens above here… we get this design items palette, which is basically a lookup tool that pulls up all your issues, bugs, user stories, etc., on any board on your Trello or Jira account… based on the keywords you are typing. As we start typing few keywords… we start seeing the relevant design items found… Here, this is the one we are looking for… We can select multiple items of course in case our method is related to more than one. Now, because part of the operation involves an automatic commit, you can edit the commit message if you like,…. but removing the tag is not permitted obviously. Now we just confirm, and that is it. We see the operation progressing… and, your method is now all linked up with your issue or issues… in the source code here, in your Github repo, and on the dashboard."*

An Alternative Post-Development Tagging: Participant is shown an alternative mechanism for tagging existing code, that offers faster accessibility.

> *"OK., going through scanning few files is probably not what we mostly want to do. Something more straightforward could probably more helpful here. So, we built this viewlet called where we can see*

all untagged code items listed in one place and organised per file. As long as you have checked out your relevant branch… you could come over here and you will see a simple tree list of all files that have code items eligible for tagging. Under each file, we now see only the specific items that are recommended for tagging. We can tag from here right away, or choose to navigate to the source code if we would like to read first."

Tagging Existing Code from the Design Perspective: Participant is shown two mechanisms for tagging existing code from the design perspective.

"So far we have seen two ways to perform post-development tagging starting from the source code… but we thought maybe in some occasions, it might be more convenient or relevant to perform the post development tagging from the design point of view... that is from the requirement perspective. For example, maybe you want to tag the code you contributed during the last release… or maybe just the last two issues, user stories, etc. In this case, we could come over here to the dashboard Viewlet, where we can find all our tickets or cards organised and categorised per their type… or in this case, per their workflow state. We can also just start typing to quickly lookup an item… the tooltip allows us to read more of the description if we need to be certain, and once found we can initiate the tagging on the specific issue or card desired. Similar to before, the process allows us to lookup code items by any keyword of their name or file, and then select those relevant."

"But probably more helpful is that we could just point to a particular board (or even specific release or sprint) and invoke this action that will reveal for us only the design items that are untagged so we can just focus on those."

Tagging Facilitated by Visualisation: A developer utilises the visualisation scene to locate and tag specific modules that are of particular interest.

"Ok, now we have seen some different ways to tag existing code base. One last possible scenario we thought of is what if we wanted to prioritise the tagging of specific modules… maybe based on some criteria, like their complexity or type… or maybe target some specific safety-critical components. We can of course go locate these manually scanning through files and folders… but we thought a better way could be to bring up the visualisation scene [AgileInsight Visualiser is launched]. Here, now we have like a birds' eye view of the entire codebase and its structure all at The layout… the colors, and buildings' sizes could probably help to spot areas of interest?… In this case, we can point to such item to tag it directly in the scene."

"Or we can locate and preselect items of interest in advance to work on them at later stage. For example, this building is exceptionally large so it is probably too complex and maybe we would want to prioritise it?… We can double click to select it… We can keep adding individual items or even an entire module. A lookup and filtering tool could also be used to locate specific code items of interest directly in the scene (not fully implemented yet). Once all the code items of interest are selected in the scene… we can save a screenshot for later reference or for team discussion. Or, could bring up a selection viewlet (not yet implemented) where all the selected code items are kept for later reference."

V.III Scenario Three—Tasks and Script

Exploring a New Codebase: A developer uses the visualisation to learn and explore a new codebase. The visualisation technique helps to expose their code structure, reveal aspects of its complexity, and helps build a shared mental map.

> *"During the previous demonstrations, we had brief exposure to the visualisation… We think the technique itself has potential in enabling us to learn about our code… it appears it could give us a good sense of the code complexity and also its structure… like where things are situated across the codebase. We are wondering if it could offer help in this aspect, especially for a new comer for example..."*
>
> *"Let me show you what I mean… see this is for example the iTrust codebase all visualised as we have seen earlier. You see, from the first glance, I could spot that there are two main modules… this one over here, and that one over there. By pointing over this module, I can now recognise that this is where all the Webroot files reside (Labels rendered directly on buildings were not implemented yet, so demonstration relied on basic labels revealed in status bar upon pointing). I can point here, and see right away that these are all CSS files… over there it's all html ones… and these over here are all JavaScript files. We are wondering if this could help in discovering the code to find out where things are located… Do you think the way these buildings are laid out reveal or communicate the structure of the code base? Do you think it could reveal insights about its architecture? Ideally, those buildings will be readily labelled, especially the outstanding ones… but we can of course actively explore around. For example, looking at all these buildings, we could tell by their whitish color, that they are all classes…[mouse pointed at one] ah, they are Java classes… and these ones on top.. are all methods! Those bluish ones over there are all functions… but why are they all nested up like this??… Ah, there is even a large class nested deep inside that big outer function! Do you think such activity is useful to identify irregularities about the code structure? Like to spot if a module is likely not conforming to the team's guidelines or coding practice."*

Shared Mental Map: Participant is engaged in active discussion to elicit their opinion, thoughts, and personal experiences with regard to the potential benefit of the visualisation in promoting a shared mental map amongst team members.

> *"OK., we saw how the visualisation could potentially be used learn about our code. But, we are thinking if it could also aid in team communication… like could this visual map work as a convenient mental reference when we come to discuss something with a colleague? Or maybe even between ourselves when we are working on or thinking about a specific problem? Because normally*

developers would each come to build their own mental map of any codebase they work on… So, we are wondering if bringing this outside to the surface, could create a shared and unified mental map among all team members. What do you think? Could such thing aid group discussions? What about using the visualisation displayed on a large screen in some team meetings? Could you see any use or benefit? Like pointing to spots of interests while talking… selecting items that are of concerns so they could be investigated later… etc."

Revealing Code Impact: Developers and other team members use AgileInsight's visualisation to examine impact of features on the codebase. The visualisation exposes the locality and the distribution of implementation, potentially revealing insights and aiding with some development activities.

"The visualisation could become more expressive when we start to augment it with other relevant information. See, as developers start to tag their code, the source code become sort of synched and unified with the design… I mean the original requirements. When we map this onto the visualisation, we think it could reveal additional insights that could help with the development"

"Let me show you something here… let us go to the dashboard viewlet. Imagine I am with my team in a sprint retro session. The code quality person might have particular interest in knowing the impact of each user story on the code base… the team lead and testing engineer might also be keen in finding what each user story has contributed to the code base… maybe they want to make sure implementations are well-contained, cohesive, and don't have tangled couplings everywhere. The visualisation maybe could help in such cases. We can pick any issue or card from the dashboard viewlet, and invoke the 'Reveal Code Impact' command. Let us invoke it on this issue for example. Now the visualisation is automatically launched, and it will highlight all the code items that associated with that issue. You see them now going up and down across the scene… the visualiser will animate them for few seconds to help spot them, in case they were too small or blocked from our angle. It also uses different colors to distinguish relations that are at file-level from those at inner level."

"Do you find exposing the exact location of where a user story or an issue is implemented in the code base is helpful? Do think being able to see how the implementation is distributed across the codebase to offer any benefit or insight in real life? Would it answer some questions that you come across in your work? What about refactoring? Does it reveal insights about couplings?"

"In this iTrust codebase, the tracelinks have been preloaded of course… but in real-world, as developers start to regularly tag their code, we think the tool could help to deliver this knowledge in useful ways. We can pick any design item, and instantly expose where it is implemented across our codebase. Developers can show their contribution visually to everyone on the table. The testing engineer and code quality person could inspect the distribution or impact of specific features on each release of the system. We are wondering if the visualisation could reveal insights about the evolution of a codebase (e.g., architecture erosion)… because as the system evolves, the locality of the implementation distribution could change and the tool could expose it."

"Now, just as we started with a design item in the dashboard view and revealed its implementation across the visualised code, we could also do the opposite… we could pick any building in the visualisation, and invoke the related design items command, which would reveal all the issues, bugs, or user stories that are associated to this code item"

Traversing the Traceability Chain: Taking advantage of AgileInsight's Synchronised code and design, a developer uses some bi-directional traceability functionalities to aid their development activities.

"Let us imagine we were trying to fix a bug in this feature that was developed in the past by someone else [pointing to one in dashboard viewlet]… or maybe add some enhancement. In a typical situation, we would most likely need to spend some time first familiarising ourself with what the user story is about. Maybe we would go to the dashboard portal to look it up, or maybe dig in Github issues and discussion threads. We would probably also invoke our experience, background, and knowledge of the codebase to guess what files we might need to check to find the part of code that is of interest… Or, if we are lucky and the team were used to tagging their commit messages, we might go to the repo site and do some searching and reading there."

Revealing Related Code Items—Two Ways

"AgileInsight, attempts to help developers in these situations. As we have seen earlier, we could use the reveal code impact visualisation functionality to not just instantly identify the files, but the exact methods or functions that are of direct relevance to this feature. But in such case, we could also invoke this 'Related Code Items' command on any design item… [command is invoked on an item in the dashboard viewlet]. Here… now we get this viewlet showing a simple interactive list of all code items related to the feature. If we click on any of those code items, it would be animated and highlighted in the visualisation too, so we get to see where exactly it is located. We can keep this viewlet open as a reference while we are completing our task."

Bi-Directional Traceability for Change Impact Analysis—without Leaving the IDE

"The good thing is that we could keep going on traversing the traceability chain… I mean OK, we have revealed where this feature is implemented and all its related code pieces… but what if we decide that we now need to modify this particular method?… Most likely we would like to ensure that our modification does not impact other features or break other parts of the code (and it is not just about code dependencies, we are dealing here with domain relationships, so it could be a conceptual/design impact). In this case, we could point to the method in the visualisation and invoke the 'Reveal Related Design Items' command and it will show all the design items in the dashboard viewlet that are impacted by it. We could then hover on the card title to reveal the full description… and do our change impact analysis right here without leaving vscode. Maybe from their title or description we could dismiss some of them as no impact, and for others we might want to do a deeper check by invoking the show code items command. So, we started with a design item, exposed its related code items, and then picked one particular code item to expose its related design items. We can keep doing this snowballing from design to code to design again and so on…"

"So we are wondering if this functionality to do bidirectional traceability could be helpful for developers to understand code change Impact during their development tasks. For example, other than bug fixing or new enhancements… do you think the technique we just illustrated could help with code refactoring?"

In-Editor Design Knowledge and Traceability: A developer uses AgileInsight in-editor features to aid with code comprehension while working in an open source file. Design Item tags attached to code items offer on the spot requirement knowledge and traceability mechanism.

"We have seen how having access to the traceability from the visualisation scene and from the dashboard viewlet could potentially aid some development tasks. But what about when a developer is having some file open… maybe trying to understand some part… or writing some code. We are wondering if having accessibility to that same level of traceability right in context within an open file… could be helpful as well. Let me show you what I mean… let us open this file, and now imagine we are trying to understand this part of the code… You could see these design item tags on top of some code items. By hovering over a tag, we get to see quick info about it like the title, its description, who worked on it, etc. Do you think having this information presented in context in the source code, is helpful for someone who is trying to gain some understanding? It is like having the requirement attached to the spot where it belongs to in the source code. We could actually hover over the method name and choose to display its related design items right from here in the source

code… Do you think this could be of any help to the way you work? Is there any occasion you could relate to from your experience where this could have helped?"

Appendix VI: Ethics Committee Approval Documents

VI.I Approval Letter

Auckland University of Technology Ethics Committee (AUTEC)

Auckland University of Technology
D-88, Private Bag 92006, Auckland 1142, NZ
T: +64 9 921 9999 ext. 8316
E: ethics@aut.ac.nz
www.aut.ac.nz/researchethics

21 May 2019

Stephen MacDonell
Faculty of Design and Creative Technologies

Dear Stephen

Re Ethics Application: **19/131 Software visualisation to support agile software processes**

Thank you for providing evidence as requested, which satisfies the points raised by the Auckland University of Technology Ethics Committee (AUTEC).

Your ethics application has been approved for three years until 21 May 2022.

Standard Conditions of Approval

1. A progress report is due annually on the anniversary of the approval date, using form EA2, which is available online through http://www.aut.ac.nz/research/researchethics.
2. A final report is due at the expiration of the approval period, or, upon completion of project, using form EA3, which is available online through http://www.aut.ac.nz/research/researchethics.
3. Any amendments to the project must be approved by AUTEC prior to being implemented. Amendments can be requested using the EA2 form: http://www.aut.ac.nz/research/researchethics.
4. Any serious or unexpected adverse events must be reported to AUTEC Secretariat as a matter of priority.
5. Any unforeseen events that might affect continued ethical acceptability of the project should also be reported to the AUTEC Secretariat as a matter of priority.

Please quote the application number and title on all future correspondence related to this project.

AUTEC grants ethical approval only. If you require management approval for access for your research from another institution or organisation then you are responsible for obtaining it. You are reminded that it is your responsibility to ensure that the spelling and grammar of documents being provided to participants or external organisations is of a high standard.

For any enquiries, please contact ethics@aut.ac.nz

Yours sincerely,

Kate O'Connor
Executive Manager
Auckland University of Technology Ethics Committee

Cc: , malshakh@aut.ac.nz; jbuchan@aut.ac.nz

Consent Form—Interview Individual Particiant

Project title: ***Software Visualisation to Support Agile Software Processes***

Project Supervisor: ***Stephen MacDonell, and Jim Buchan***

Researcher: ***Mujtaba Alshakhouri***

- O I have read and understood the information provided about this research project in the Information Sheet dated 12 May 2019.
- O I have had an opportunity to ask questions and to have them answered.
- O I understand that notes will be taken during the interviews and that they will also be audio recorded and transcribed.
- O I understand that taking part in this study is voluntary (my choice) and that I may withdraw from the study at any time without being disadvantaged in any way.
- O I understand that if I withdraw from the study then I will be offered the choice between having any data that is identifiable as belonging to me removed or allowing it to continue to be used. However, once the findings have been produced, removal of my data may not be possible.
- O I agree to take part in this research.
- O I wish to receive a summary of the research findings (please tick one): YesO NoO
- O I agree to have my interview audio recorded so researcher can refer to it when needed to double check information (please tick one):

  YesO NoO

Participant's signature: .....

Participant's name: .....

Participant's Contact Details (if appropriate):

.....

.....

.....

.....

Date:

Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

Note: The Participant should retain a copy of this form.

Participant Information Sheet

Date Information Sheet Produced:

12 May 2019

Project Title

Software Visualisation to Support Agile Software Processes

An Invitation

My name is Mujtaba Alshakhouri. I am a doctoral student in the School of Engineering, Computer and Mathematical Sciences at Auckland University of Technology, under the supervision of Prof Stephen McDonell and Jim Buchan. I am conducting research investigating the application of software visualisation techniques to support the agile software development process.

In this regard, I would like to invite you to participate in my research and be part of this promising study that could contribute to the improvement of agile practices. Please note that your participation in this research is voluntary in nature, and you may decline or withdraw your participation without any adverse consequences. None of the participants will be identified nor will the information gathered be used to hamper, hinder or harm your career.

The following questions and answers are intended to address the most common questions that a participant might ask about this research project. If you need further information, feel free to contact the researcher, Mujtaba Alshakhouri. My contact details can be found at the end of this document. It is recommended that you use e-mail to reach me.

What is the purpose of this research?

This research aims to investigate the benefits of applying software visualisation techniques to support agile development practice in industry. It employs a visual technique that unifies and synchronises software code entities with their original user stories and the associated development activities. This is expected to aid agile software developers in carrying out some of their tasks by simplifying them. This research is focused on identifying those tasks and applications that would most benefit from the use of this technique, and to verify them through observation and interview.

Software visualisation research utilises the naturally amplified capacity of humans in processing information that is presented to them visually (as in shapes, images, and graphs) rather than in textual or auditory forms. In this research, information about software code entities, their structure, relationships, dependencies, development activities, and their original user stories are encoded in a visual metaphor imitating city buildings. This metaphor, inspired by a natural and familiar environment, takes advantage of our readiness to quickly relate to this phenomenon to better communicate otherwise complex information.

This research contributes to the partial fulfilment of my doctoral degree. Findings and results obtained from this research could become incorporated (directly or indirectly) in academic publications and/or presentations that are also in partial fulfilment of my degree. Confidentiality of participants will be strictly enforced in all circumstances.

How is this research being conducted?

The research is structured in three stages where industry participants are being involved. The first stage (to which you are being invited) is aimed at understanding industry's practices and workflows around working with agile software development. The second and third stages involve participants engaging in experimental tool usage to assess its effectiveness. Invitees who choose to participate in this first stage will likely receive future invitations to participate in the two other stages. Participating in any stage remains completely voluntary, however, and independent of previous participation. To ensure complete confidentiality, no participant association will be made between the various stages and the results obtained from them.

Why am I being invited to participate in this research?

Your organisation has been identified and approached by the research supervisors' (or researcher) as an industry practicing team-based agile software development. As a member of this organisation and a practitioner of agile software development, your expertise, direct experiences, and field knowledge constitute a valuable resource to our research. By receiving this invitation, it means that your organisation has agreed to allow their employees to participate in our research on a voluntary basis.

Does my boss know I'm being approached?

Your manager (or the person to whom you report) will receive a request to grant us access to their employees. However, this request will be made in general and no particular employee will be identified or named. Employees will receive a general invitation message from their organisations to ask interested parties to participate. Understandably though, your manager might still come to know of your participation through other means. Nonetheless, the study will be taking special measures to provide you with sufficient privacy and confidentiality during and after the interview.

How do I agree to participate in this research?

To follow up on this invitation to participate in this research, please confirm your acceptance by signing the accompanying Participant Consent Form and returning it to me. You will also have a chance to reconsider and/or reconfirm your consent to participate just prior to your interview.

Your participation in this research is voluntary (it is your choice) and if you choose not to participate this will be of no disadvantage to you. Your manager (or the person from your organisation to whom you report) will also receive a consent form to grant us access to their employees in general. Your identity will be kept confidential and will not be shared with your organisation.

You are able to withdraw from the study at any time. If you choose to withdraw from the study, then you will be offered the choice of having any data collected from you removed, or allowing it to continue to be used. However, once the findings have been produced, removal of your data may not be possible.

What will happen in this research?

If you accept this invitation to participate you will be interviewed individually by the researcher. This will be a loosely structured interview where you will be asked some open-ended questions related to your project, your role in that project, and the tools you use to complete your day-to-day software development tasks. You will also be asked about workflows, routines, and ways of working that you specifically follow to complete those tasks.

A wireframe or interface prototype will then be demonstrated to you and explained. You will then be asked for feedback and opinions about this UI prototype, whether/how you think it might help you and your team in your daily work activities, and any desired functionalities that you would like to see in it for it to be more useful. You will also be given the opportunity to provide general suggestions/advice on how the UI could be improved.

The researcher will be taking notes, and will audio record the interview to enable later reference and analysis during the course of the research. The analysis will involve coding the data to identify trends and themes that provide insights to practitioners' needs, and typical workflows in the agile software industry. Note that the recording of the interview will not be transcribed in full, but quotes may be extracted as evidence of patterns identified. The data will be held confidentially and will have all references to organisations or individuals removed before analysis.

At the end of this research a report summarising the main results will be made available to you if requested on the Consent Form. Furthermore, it is expected that the data could be incorporated in papers published in academic journals and conferences where the main findings of this research project will be reported.

What are the discomforts and risks?

During the interview session, there is a possibility you may feel uncomfortable about sharing your point of view about your project, ways of working, and specific work processes/flows practised at your workplace.

Additionally, you may feel uncomfortable and unwilling to share information related to business and work processes or share your personal feelings.

You may feel uncomfortable that your line manager might come to know who is participating in the study and who has elected not to take up the invitation, and that this could affect their perception of you, and future prospects.

You may feel uncomfortable about having your interview recorded.

You may feel uncomfortable that your colleagues or line managers may overhear what you say during the interview, and that this could negatively affect their perception of you.

How will these discomforts and risks be alleviated?

In order to alleviate the first concern of possible discomforts, you will be reminded of our assurance of confidentiality of all interview data at the start of the interview process. You and your organisation will also be assured of the independent nature of this study and that no performance or capabilities of you or your organisation are being studied in any specific way. Rather, the study is focused on understanding the mainstream workflow of the agile software development practice in industry from an abstract and generic perspective.

You may choose not to answer specific questions, and you can also withdraw from participating in the interview at any stage. You can also request that your interview data be withdrawn from the study before the completion of data collection.

The second possible area of discomfort will be addressed by stressing the voluntary nature of participation to both you and your organisation. We understand the time pressures faced by you as an employee, and recognise that it is not always feasible or practical to participate in such studies. While your line manager may know you have been approached, participation or non-participation will not be specifically recorded or communicated apart from the need to organise a time and date for your interview.

You will be offered three options of venues to choose from for the interview to take place. These are, a) a closed meeting room at AUT premises, b) a meeting room at your own workplace, c) a neutral place away from work. If you choose option b) then a soundproof room will be requested. This shall provide you with comfort, ensure the confidentiality of your interview, and prevent anyone overhearing. If you choose option b) but no soundproof or sufficiently private room can be provisioned, then you will be asked to consider option a) or c) instead.

Recording of the interview is not a prerequisite of conducting the interview. Before the interview begins you will be asked for permission to record the interview. Even if consent to record is provided, you will be reminded that you can request that the recording be stopped or wiped at any stage of the interview.

What are the benefits?

Findings from this research will contribute to the body of knowledge in the software engineering discipline and could influence future practice in the area of agile software development. The insights gained from this study will be made available to yourself and your colleagues if you indicate your interest on your Consent Form. This could be valuable for your organisation in facilitating broader knowledge dissemination and potential future improvements.

Lastly, this research will be contributing to the partial fulfilment of my academic degree requirements.

How will my privacy be protected?

All of the materials related to participants' information (consent form, recording, and interview notes) will be stored at AUT in locked cupboards for at least 6 years. After that, the material will be destroyed.

No transcriber will be involved in transcribing the recorded interview. The researcher may, however, transcribe small parts of the recorded material to use as exemplars and evidence of trends and claims revealed in the analysis.

The data from the interviews will be de-identified and analysed for principles and insights that are independent of the interviewee's identity. Furthermore, demographic data will be coded and the data stored in a separate place so that the identity of each participant will be separated from their responses.

If you decide to withdraw from this research project for any reason before the completion of data collection, all of the materials relating to your interview will be destroyed as soon as practicable after your request.

In addition, your line manager will not hear or see the content of this research data. The only people who will have access to your data will be the researcher and the researcher's supervisors.

What are the costs of participating in this research?

Time is the only cost to you. We estimate that the interview will take around one hour of your time.

What opportunity do I have to consider this invitation?

Due to time restrictions in undertaking the laboratory work for the research, we would ideally like to have notice of your agreement within a week of you receiving this invitation.

Will I receive feedback on the results of this research?

If you would like a report summarising the results of this research, please tick the appropriate box on the Consent Form, provided at the interview. You will be emailed the summarising report once this research is completed.

What do I do if I have concerns about this research?

Any concerns regarding the nature of this project should be notified in the first instance to the Project Supervisors, *Stephen G. MacDonell, stephen.macdonell@aut.ac.nz, 09* 921 9999; or Jim Buchan, jbuchan@aut.ac.nz, 09 921 9999 ext 5455

Concerns regarding the conduct of the research should be notified to the Executive Secretary of AUTEC, Kate O'Connor, *ethics@aut.ac.nz*, 09 921 9999 ext 6038.

Whom do I contact for further information about this research?

Please keep this Information Sheet and a copy of the Consent Form for your future reference. You are also able to contact the research team as follows:

Researcher Contact Details:

Mujtaba Alshakhouri
Software Engineering Research Lab (SERL), School of Engineering, Computer and Mathematical Sciences
Auckland University of Technology, Private Bag 92006
Auckland 1142, NZ
Phone: + 64 9 921 9999 x 6376
Email: malshakh@aut.ac.nz

Project Supervisor Contact Details:

Stephen MacDonell
Software Engineering Research Lab (SERL), School of Engineering, Computer and Mathematical Sciences
Auckland University of Technology, Private Bag 92006
Auckland 1142, NZ
Phone: +64 9 9219999 ext 5811
Email: stephen.macdonell@aut.ac.nz

Jim Buchan
Software Engineering Research Lab (SERL), School of Engineering, Computer and Mathematical Sciences
Auckland University of Technology, Private Bag 92006
Auckland 1142, NZ
Phone: +64 9 9219999 ext 5455
Email: jbuchan@aut.ac.nz

Approved by the Auckland University of Technology Ethics Committee on *21 May 2019*, AUTEC Reference number *19/131*.

Please do not staple your application

AUT
TE WĀNANGA ARONUI O TĀMAKI MAKAU RAU

Auckland University of Technology Ethics Committee (AUTEC)

For AUTEC Secretariat Use only

_____/_____

EA1

APPLICATION FOR ETHICS APPROVAL BY AUTEC

Please print this application single sided in greyscale and do not staple. Once this application has been completed and signed, please read the notes at the end of the form for information about submission of the application for review.

NOTES ABOUT COMPLETION

- Ethics review is a community review of the ethical aspects of a research proposal. Responses should use clear everyday language with appropriate definitions being provided should the use of technical or academic jargon be necessary.
- The AUTEC Secretariat and your AUTEC Faculty Representative are able to provide you with assistance and guidance with the completion of this application which may help expedite the granting of ethics approval.
- The information in this application needs to be clearly stated and to contain sufficient details to enable AUTEC to make an informed decision about the ethical quality of the research. Responses that do not provide sufficient information may delay approval because further information will be sought. Overly long responses may also delay approval when unnecessary information hinders clarity. In general, each response should be around 100 words.
- AUTEC reserves the right not to consider applications that are incomplete or inadequate. Please do not alter the formatting or numbering of the form in any way or remove any of the help text.
- Comprehensive information about ethics approval and what may be required is available online at http://aut.ac.nz/researchethics
- The information provided in this application will be used for the purposes of granting ethics approval. It may also be provided to the Graduate Research School, the Research and Innovation Office, or the University's insurers for purposes relating to AUT's interests.
- The Form is focussed around AUTEC's ethical principles, which are in accordance with the *Guidelines for the approval of ethics committees* in New Zealand.

To respond to a question, please place your cursor in the space following the question and its notes and begin typing.

A. *Project Information*

A.1. What is the title of the research?

If you will be using a different title in documents to that being used as your working title, please provide both, clearly indicating which title will be used for what purpose.

Software Visualisation to Support Agile Software Processes

A.2. Is this application for research that is being undertaken in stages? ☒ Yes ☐ No

If the answer is 'Yes' please answer A.2.1 and the following sections, otherwise please answer A.3 and continue from there.

A.2.1. Does this application cover all the stages of the research? ☐ Yes ☒ No

If the answer is 'No' please provide details here of which stages are being covered by this application, otherwise please answer A.3 and continue from there.

The research is designed to incorporate three stages; 1) initial interviews to collect practitioners' impressions and feedback on a prototype version of the tool (qualitative research), 2) participation in a laboratory experiment to verify a number of the tool's capabilities and functionalities carried out on the first version of the tool, and finally, 3) a re-run of the previous experiment carried out on a second version of the tool where refined measurements are carried out. The second and third stages will involve qualitative and quantitative elements.

This application addresses the first of these three stages only. A new application will be made regarding stages 2 and 3.

A.3. Who is the applicant?

When the research is part of the requirements for a qualification at AUT, then the applicant is always the primary supervisor. Otherwise, the applicant is the researcher primarily responsible for the research, to whom all enquiries and correspondence relating to this application will be addressed.

Stephen MacDonell

A.4. Further information about the applicant.

A.4.1. In which faculty, directorate, or research centre is the applicant located?

Software Engineering Research Lab (SERL), School of Engineering, Computer and Mathematical Sciences

A.4.2. What are the applicant's qualifications?

PhD, Master of Commerce, Bachelor of Commerce (Hons)

A.4.3. What is the applicant's email address?

An email address at which the applicant can be contacted is essential.

smacdone@aut.ac.nz

A.4.4. At which telephone numbers can the applicant be contacted during the day?

+64-9-921 9999

A.5. Research Instruments

A.5.1. Which of the following does the research use:

☐ ***a written or electronic questionnaire or survey*** ☐ ***focus groups*** ☒ ***interviews***
☐ ***observation*** ☐ ***participant observation*** ☐ ***ethnography*** ☐ ***photographs***
☐ ***videos*** ☐ ***other visual recordings*** ☐ ***a creative, artistic, or design process***
☐ ***performance tests***
☐ ***some other research instrument (please specify)***

Please attach to this application form all the relevant research protocols. These may include: Indicative questions (for interviews or focus groups); a copy of the finalised questionnaire or survey in the format that it will be presented to participants (for a written or electronic questionnaire or survey); a protocol indicating how the data will be recorded (e.g. audiotape, videotape, note-taking) for focus groups or interviews (Note: when focus groups are being recorded, you will need to make sure there is provision for explicit consent on the Consent Form and attach to this Application Form examples of indicative questions or the full focus group schedule. Please note that there are specific confidentiality issues associated with focus groups that need to be addressed); a copy of the observation protocol that will be used (for observations); full information about the use of visual recordings of any sort, including appropriate protocols and consent processes; protocols for any creative, artistic, or design process; a copy of the protocols for the instruments and the instruments that will be used to record results if you will use some other research instrument.

A.5.2. Who will be transcribing or recording the data?

If someone other than the applicant or primary researcher will be transcribing the interview or focus group records or taking the notes, you will need to provide a confidentiality agreement with this Application Form.

The primary researcher will record and transcribe the data

A.6. Please provide a brief plain English summary of the research (300 words maximum).

This research aims to investigate the benefits of applying software visualisation techniques to support agile development practice in the software industry. It employs a visual technique that unifies and synchronises software code entities with their original user stories (i.e., requirements) and their development activities. This is expected to aid the agile software developer in carrying out some of their tasks and/or simplifying them. The research is focused on identifying those tasks and applications that would most benefit from the use of SV techniques, and to verify this empirically.

Software visualisation research utilises the naturally amplified capacity of humans in processing information that is presented to them visually (as in shapes, images, and graphs) rather than that presented in textual or auditory forms. In this research, information about software code entities, their structure, relationships, dependencies, development activities, and their associated user stories, are encoded in a visual metaphor imitating city buildings. This metaphor, which is inspired by a familiar natural environment, takes advantage of our readiness to readily relate to this phenomenon to better communicate what is otherwise complex information. The resulting research artefact (which will include a software tool) is expected to demonstrate and verify a number of advantages to the agile software practitioner community that range from supporting specific activities, such as feature traceability and feature location, to other broader benefits, such as change impact analysis, design reasoning, and aiding team communication.

Three cycles of design, evaluation, and laboratory refinement will be employed in the research. This consists of initial collection of experts' feedback through interviews, followed by two rounds of laboratory experiments conducted using the developed software tool. The interviews represent the first stage of the research for which this application is addressed.
Contribution to the discipline's body of knowledge is expected to result from the devised techniques employed by the tool, as well as from the empirical results obtained from the experiments.

A.7. Additional Research Information

A.7.1. Is this research an intervention study? ☐ Yes ☒ No

For research in general, what is the difference between intervention, interaction, and observation? Intervention includes both physical procedures by which data are gathered and manipulations of the participant or participant's environment that are performed for research purposes. Interaction includes communication or interpersonal contact between the investigator and participant that are performed for research purposes. Observation is neither an intervention nor an interaction. (cf https://www.gvsu.edu/hrrc/faq-definitions-35.htm).

Within health and disability research, 'intervention study' has the meaning given to it by the National Ethics Advisory Council's Ethical Guidelines for Intervention Studies; namely, a study in which the investigator controls and studies the intervention(s) provided to participants for the purpose of adding to knowledge of the health effects of the intervention(s). The term 'intervention study' is often used interchangeably with the terms 'experimental study' and 'clinical trial' (s.24 Standard Operating Procedures for Health and Disability Ethics Committees).

A.7.2. Is this Health and Disability Research? ☐ Yes ☒ No

Health and disability research is research that aims to generate knowledge for the purpose of improving health and independence outcomes (s.21 Standard Operating Procedures for Health and Disability Ethics Committees).

A.7.3. Does this research involve people in their capacity as consumers of health or disability support services, or in their capacity as relatives or caregivers of consumers of health or disability support services, or as volunteers in clinical trials (including, for the avoidance of doubt, bioequivalence and bioavailability studies)? ☐ Yes ☒ No

B. *The Ethical Principle of Research Adequacy*

AUTEC recognises that different research paradigms may inform the conception and design of projects. It adopts the following minimal criteria of adequacy: the project must have clear research goals; its design must make it possible to meet those goals; and the project should not be trivial but should potentially contribute to the advancement of knowledge to an extent that warrants any cost or risk to participants.

B.1. Is the applicant the person doing most of the research (the primary researcher)? ☐ Yes ☒ No

If the answer is 'No' please answer B.1.1 and the following sections, otherwise please answer B.2 and continue from there.

B.1.1. What is the name of the primary researcher if it is someone other than the applicant?

Mujtaba Alshakhouri

B.1.2. What are the primary researcher's completed qualifications?

Master's in Computer & Information Sciences (Hons), PG Dip. Computer Science, GDip. SIT, BSc. In Computer & Information Sciences.

B.1.3. What is the primary researcher's email address?

An email address at which the primary researcher can be contacted is essential.

malshakh@aut.ac.nz

B.1.4. At which telephone numbers can the primary researcher be contacted during the day?

+64 9 921 9999 ext. 6376, +64 21 529080

B.2. Is the primary researcher

☐ an AUT staff member ☒ an AUT student

If the primary researcher is an AUT staff member, please answer B.2.1 and the following sections, otherwise please answer B.3 and continue from there.

B.2.1. In which faculty, directorate, or research centre is the primary researcher employed?

If the response to this section is the same as that already given to section A.4.1 above, please skip this section and go to section B.2.2.

B.2.2. In which school or department is the primary researcher employed?

B.3. When the primary researcher is a student:

B.3.1. What is their Student ID Number?

1132793

B.3.2. In which faculty are they enrolled?

Faculty of Design and Creative Technologies

B.3.3. In which school, department, or Research Centre are they enrolled?

Software Engineering Research Lab (SERL), School of Engineering, Computer and Mathematical Sciences

B.4. What is the primary researcher's experience or expertise in this area of research?

Where the primary researcher is a student at AUT, please identify the applicant's experience or expertise in this area of research as well.

The researcher is experienced in working in software development using both traditional and agile methodologies. Moreover, he has a background in agile software methodologies as well as software visualisation techniques and completed his Master's degree at AUT in the same research area.

B.5. Who is in charge of data collection?

The primary researcher

B.6. Who will interact with the participants?

The primary researcher, applicant and secondary supervisor (Jim Buchan)

B.7. Is this research being undertaken as part of a qualification? ☒ Yes ☐ No

If the answer is 'Yes' please answer B.7.1 and the following sections, otherwise please answer B.8 and continue from there.

B.7.1. What is the name of the qualification?

Doctor of Philosophy, Computer and Information Sciences

B.7.2. In which institution will the qualification be undertaken?

AUT

B.8. Details of Other Researchers or Investigators

B.8.1. Will any other people be involved as researchers, co- investigators, or supervisors? ☒ Yes ☐ No

If the answer is 'Yes' please answer B.8.1.1 and the following sections, otherwise please answer B.8.2 and continue from there.

B.8.1.1 What are the names of any other people involved as researchers, investigators, or supervisors?

Jim Buchan

B.8.1.2 Where do they work?

Software Engineering Research Lab (SERL), School of Engineering, Computer and Mathematical Sciences

B.8.1.3 What will their roles be in the research?

Same as Main applicant

B.8.1.4 What are their completed qualifications?

BSc, MSc(Hons), GradDip(InfoSys), DipTchg

B.8.2. Will any research organisation or other organisation be involved in the research? ☐ Yes ☒ No

If the answer is 'Yes' please answer B.8.2.1 and the following sections, otherwise please answer B.9 and continue from there.

B.8.2.1 What are the names of the organisations?

B.8.2.2 Where are they located?

B.8.2.3 What will their roles be in the research?

B.9. Why are you doing this research and what is its aim and background?

Please provide the key outcomes or research questions and an academic rationale with sufficient information, including relevant references, to place the project in perspective and to allow the project's significance to be assessed.

This research endeavour is building on and expanding prior work by the student in the area of software visualisation. It aims to operationalise this relatively new technology and establish concrete application uses in the industry domain of agile software development. It seeks to achieve this through exploratory design and development activities that are informed by the latest literature in the field, and through empirical studies that investigate and verify the efficacy of those developed application techniques.

The main intent of this research can be summarised by the following question. "How can Software Visualisation Technology be better employed to achieve measurably effective value for the agile software development process—Scrum, in particular?" In our attempt to answer that question we are guided by three specific objectives.

Objective 1. To explore how SV technology can be utilised to establish a *Conceptual Model* of a system wherein the development process, its end products, and their relationships are visually captured in a unified and synchronised manner. Expected outcomes include a number of valuable benefits to the SE research and practice communities, such as artefact traceability, feature location, and change impact analysis.

Objective 2. To investigate whether the devised Conceptual Model brings advantage to users in terms of scaffolding their cognitive capacity when approaching the large number of code entities composing any non-trivial system. This primarily comes to working one's way around the complexities of the structure, relationships, dependencies, and the original user stories behind those code entities. Benefits are expected to manifest at the individual as well as the team level. The interactive visualised model unifying the three aspects mentioned earlier, is preconceived to create a shared communication medium for the whole team. Instead of leading to several varying mental pictures in everyone's minds, a single tangible visual model becomes commonly available to the team, which they can inspect together and through which they may explore the mental concepts behind the constituent code entities and their development process.

Objective 3. To verify the utility and efficacy of the proposed concept and its devised techniques through the conduct of rigorous empirical work that draws strength from a collaboration with the practicing community. This objective directly addresses a major gap in field, which is the lack of empirical evaluation of SV tools and techniques. It also contributes to establishing a solid empirical foundation upon which we may promote the technology among industry practitioners.

The significance of this research endeavour is realised and justified through three problem spaces, which we briefly discuss below.

a. Promoting adoption among practitioners.

The value of visualisation as a means of delivering and communicating information is well established in many scientific disciplines. Its ability to create a common basis of mental concepts among a working team is invaluable (Oppl, 2017; Roberts, Ritsos, Jackson, & Headleand, 2018; van Wijk, 2005). The software engineering discipline is no exception when it comes to considering the value that visualisation could present to it. Given the numerous intangible artefacts that software practitioners have to deal with, it becomes hard to question that visualising those artefacts and their interconnected relationships would bring benefit to such practitioners. The SV field has well-established empirical work supporting its value in this regard (Fittkau, Krause, & Hasselbring, 2017; Wettel, Lanza, & Robbes, 2011). It also enjoys solid theoretical foundation underpinning the cognitive advantage of its core concepts (Duru, Çakir, & Işler, 2013; Petre, 2010; Petre, Blackwell, & Green, 1998; Storey, Fracchia, & Muller, 1999). And yet, adoption of this technology by practitioners is lacking. Current research in the field is thus presently focussed on issues around practicality and applications of use, in an attempt to remove impediments that are hindering the adoption of this technology. Examples include devising new layout designs (Scheibel, Weyand, & Döllner, 2018), exploring compact layouts (Görtler, Schulz, Weiskopf, & Deussen, 2018), and investigating stable and change-accommodating layouts (Hahn, Trümper, Moritz, & Döllner, 2014; Sondag, Speckmann, & Verbeek, 2018; Tak & Cockburn, 2013). Yet other work attempts to enhance interactivity, agility, and responsiveness (Limberger, Scheibel, Trapp, & Dollner, 2017), while others pursue better integration with users' existing technology and processes (Lopez-Herrejon, Illescas, & Egyed, 2018; Paredes, Anslow, & Maurer, 2014). The proposed research is well-aligned with these overarching problems/opportunities as, in the process of addressing the three main objectives, it simultaneously incorporates efforts and experimentation to integrate the technology with the existing technology and processes of users, it incorporates elements for improving interactivity, and it incorporates experimentation intended to achieve acceptable layout stability.

b. Achieving artefact traceability in agile development environments.

While software requirement traceability is a well-researched and investigated problem space, it has received little attention with respect to the domain of agile development practice (Curcio, Navarro, Malucelli, & Reinehr, 2018; Hess, Diebold, & Seyff, 2018), and even less so for the Scrum methodology. However, interest in functional requirement traceability in agile contexts has started to gain momentum in the last few years (Curcio et al., 2018; Rivero, Grigera, Distante, Montero, & Rossi, 2017; Slob, Dalpiaz, Brinkkemper, & Lucassen, 2018).

In this line, the proposed research is expected to provide an opportunity to address the traceability problem in agile environments from a rather different angle—which could turn out to be more inclusive than 'requirements' per se. The fundamental goal of this research is to capture the original concepts and design intents behind the code entities, and to bring both to the surface in a visual and seamlessly integrated manner (Alshakhouri, 2013). When following a Scrum methodology, those original concepts and design intents are typically captured by user stories. Following up using this scenario, this novel concept could present a convenient way of achieving promising levels

of traceability between those user stories and the eventual code entities that are developed as part of fulfilling those user stories. Since the concept accommodates different aspects pertaining to the implementation process and its activities (and not just requirements), the term 'artefact traceability' is used here.

c. Addressing the empirical evaluation gap.

One issue hindering wider adoption of SV technology that is well-identified and strongly emphasised in the literature is the lack of rigorous empirical evaluation of the tools and techniques being proposed. For example, recent studies in this regard (Merino, Ghafari, Anslow, & Nierstrasz, 2018; Seriai, Benomar, Cerat, & Sahraoui, 2014) have concluded that except for very few cases, the majority of existing empirical work does not satisfy the required elements that would qualify them to be classified as rigorous studies. It was reported that empirical work carried out in the field is extremely rare, those that qualify as controlled experiments are also very few, and those that use random subject selection are very limited. The majority of the remaining studies fall in the category of case studies, simulations, and lab experiments.

This research is designed to include multiple cycles of empirical work informed by lessons learned from prior research in this regard (Müller et al., 2014; Muller, Kovacs, Schilbach, & Zeckzer, 2015). In doing so it is intended to contribute to addressing this overarching issue in the field.

B.10. What are the potential benefits of this research to the participants, the researcher, and the wider community?

The research outcomes will include a novel concept and techniques aimed at supporting the work of agile software developers by simplifying some challenges faced in their daily tasks. A software tool will be developed to implement and demonstrate these concepts.

Participants. By using and exploring the potential of this tool, participants will get the opportunity to discover new insights and learn about possible approaches to address some of their existing challenges.

Wider Community. The research will also contribute to the SE community through the introduction of these new techniques that could enrich the toolset available for the agile practitioner community. It will also contribute to the specific discipline of software visualisation through addressing the three areas of significance mentioned earlier, as well as serving to promote the SV technology through the introduced tool.

Researcher. Lastly, the successful completion of this research will lead in part to a PhD degree, plus co-authored publications, for the researcher.

B.11. What are the theoretical frameworks or methodological approaches being used?

This work is largely driven and motivated by the recognition of a new opportunity (potential) to deliver improvement to a particular application domain of the software practice through the introduction of a new concept, and the utilisation of an innovative technology to implement it. It is thus striving to contribute to, and help advance the field of software engineering through the process of artefact construction, and (most importantly) through the anticipated knowledge to be generated throughout and as a result of this process. Hence, the nature of this work places it intrinsically into the design science scheme of research. Design Science research, however, can be executed in various forms and methodologies, each focusing on different philosophical dispositions and motivations. Given the role of the application, environment, and organisational contexts in shaping and steering this research, it has been concluded that the Hevner et al. model (Hevner, March, Park, & Ram, 2004) is the one mostly suited to adopt.

A Hevner et al. design science research methodology is thus employed in this research, where iterative cycles of design, development and refinement of the visualisation approach is informed by prior experiences and expertise made available by researchers and practitioners (Background, Contact, & Bible, 2018; Hevner, March, Park, & Ram, 2004). A feedback loop from earlier results further lends strength to the research.

B.12. How will data be gathered and processed?

Qualitative data will be gathered by semi-structured interviews. The interviews will be audio recorded and supplemented by interview notes taken by the researcher. Relevant sections of the audio recording will be transcribed if required.

B.13. How will the data be analysed?

Please provide the statistical (for quantitative research) or methodological (for qualitative or other research) justification for analysing the data in this way.

The data will be analysed using thematic analysis involving categorising, coding and conceptualisation of themes. This is aimed at building a sound framework of better understanding the needs and actual workflow as practiced at industry in order to ensure the tool design is informed and driven by the application environment.

B.14. Has any peer review taken place? ☒ Yes ☐ No

If your answer is 'Yes', please specify and provide evidence e.g. a letter of confirmation.

☐ AUT Competitive Grant ***☐ External Competitive Research Grant***

☒ PGR1 ***☒ PGR2*** ***☒ PGR9*** ***☒ Independent Peer Review****

Optional exemplars for evidencing peer review are available from the Ministry of Health (HDEC) website (http://ethics.health.govt.nz/) or from the Forms section of the Research Ethics website (http://aut.ac.nz/researchethics)

C. General Project Details

C.1. Likely Research Output

C.1.1. What are the likely outputs of this research?

☒ a thesis ☒ a dissertation ☐ a research paper ☒ a journal article
☐ a book ☒ conference paper ☐ a documentary ☐ an exhibition
☐ a film ☐ some other artwork ☐ other academic publications or presentations
☒ Some other output, please specify: software tool/application

C.2. Research Location and Duration

C.2.1. In which countries and cities/localities will the data collection occur?

Auckland, New Zealand

C.2.1.1 Exactly where will any face to face data collection occur?

If face to face data collection will occur in participants' homes or similarly private spaces, then a Researcher Safety Protocol needs to be provided with this application.

Either at the place of work of the participant, at AUT premises (SERL), or at a business location convenient to the participant.

C.2.2. In which countries and cities/localities will the data analysis occur?

Auckland, New Zealand

C.2.3. When is the data collection scheduled to commence?

April to May 2019

C.3. Research Participants

C.3.1. Who are the participants?

The participants will be members of the software development industry who are undertaking software development using an agile methodology; relevant roles are likely to be scrum master, developer, tester, quality assurance, and product owner.

C.3.2. How many participants are being recruited for this research?

If you are unsure, please provide an indicative range.

15-20 participants

C.3.3. What criteria will be used to choose who to invite as participants?

Participants will be invited from selected organizations that are actively involved in software development process and agile software methodology has been adopted and implemented partially or in full.

C.3.3.1 How will you select participants from those recruited if more people than you need for the study agree to participate?

Based on participants' availability and appropriateness for the current research. For example, team members will be chosen based on their experience in working in agile software development. A variety of participants from different roles will be selected to hopefully provide a balanced representation.

C.3.4. Will any people be excluded from participating in the study? ☒ Yes ☐ No

Exclusion criteria apply only to potential participants who meet the inclusion criteria. An exclusion criterion is any characteristic that ought to disqualify any potential participant from recruitment into the study. Consider exclusion criteria when there are heightened risks due to power differences in the relationship, recent injury, or other characteristics that might place potential participants at unreasonable risk of harms.

If the answer to this question is 'Yes' please answer C.3.4.1 and the following sections, otherwise please answer C.3.5 and continue from there.

C.3.4.1 What criteria will be used to exclude people from the study?

The exclusion criterion is based on close and personal relationships. No personal friends or colleagues will be incorporated into the study.

C.3.4.2 Why is this exclusion necessary for this study?

This is to avoid conflict of interest influencing the results of the study.

C.3.5. Recruitment of participants.

Please describe in detail the recruitment processes that will be used. If you will be recruiting by advertisement or email, please attach a copy to this Application Form

C.3.5.1 How will the initial contact with potential participants occur?

Primarily through existing relationships of the researcher's supervisory team, but also possibly through networking with members of the Agile Auckland professional network or the Agile Auckland LinkedIn Group.

C.3.5.2 How will the contact details of potential participants be collected and by whom?

Some contact details are already available (via existing relationships). Some will be collected in person during networking opportunities at the meetings of Agile Auckland. Some will be contacted through LinkedIn. The combined resources of the primary researcher and the applicants will be utilised.

C.3.5.3 How will potential participants be invited to participate?

The primary researcher and applicants will contact them by email or phone and will follow up with interested parties via email to deliver the information sheets and consent forms.

C.3.5.4 How much time will potential participants have to consider the invitation?

Up to 2 weeks.

C.3.5.5 How will potential participants respond to the invitation?

Potential participants will respond by email.

C.3.5.6 How will potential participants give consent?

Potential participants will give consent by signing the Consent form and returning it by email or in person prior to the interview.

C.3.5.7 How and when will the inclusion criteria and exclusion criteria given in sections C.3.2 and C.3.3 be applied?

The inclusion criteria will be applied progressively, as responses to invitations are received.

C.3.5.8 Will there be any follow up invitations for potential participants?

No.

D. *Partnership, Participation and Protection*

D.1. How does the design and practice of this research implement the principle of Partnership in the interaction between the researcher and other participants?

How will your research design and practice encourage a mutual respect and benefit and participant autonomy and ownership? How will you ensure that participants and researchers will act honourably and with good faith towards each other? Are the outcomes designed to benefit the participants and/or their social or cultural group? How will the information and knowledge provided by the participants be acknowledged?

The research process itself is viewed strongly as a partnership between academia and industry where professional practice is both informing and being informed by a scholarly analysis of current practice, supporting tools, and latest research in the discipline. Industry partners are encouraged to participate in the understanding that this is a partnership with clear benefits and an opportunity to contribute, and with a clear undertaking to protect their values, beliefs and personal identity. The outcome of this research will benefit the practicing agile software development community in general, including the participants, and provides an opportunity for them to learn and explore new techniques that could enhance and simplify some of their work. The participants will be given the opportunity to have an executive summary of the research sent to them after the thesis is examined.

D.2. How does the design and practice of this research implement the principle of Participation in the interaction between the researcher and other participants?

What is the actual role of participants in your research project? Will participants be asked to inform or influence the nature of the research, its aims, or its methodology? Will participants be involved in conducting the research or is their principal involvement one of sharing information or data? Do participants have a formal role as stakeholders e.g. as the funders and/or beneficiaries of the research? What role will participants have in the research outputs (e.g. will they be asked to approve transcripts or drafts)?

The participants' involvement is mainly one of information sharing, including their individual perceptions and evaluations of the tool being evaluated and its potential benefit to their work. Participants will be given the opportunity to review and verify the transcript created during their interviews.

D.3. How does the design and practice of this research implement the principle of Protection in the interaction between the researcher and other participants?

How will you actively protect participants from deceit, harm and coercion through the design and practice of your research? How will the privacy of participants and researchers be protected? How will any power imbalances inherent in the relationships between the participants and researchers be managed? How will any cultural or other diversity be respected?

The interview will be designed to minimize risks to the participants. For example, the questions will be related to everyday work activities and will not be personal in nature. In addition, the interviews will be audio recorded and transcription (if applicable) will be de-identified. The interviews will take place in a place that is not within earshot of colleagues or managers in order to protect the participants from the risk of an overheard conversation being misconstrued or having a negative consequence.

Managers' approval will be candidly requested through an Organisation Consent Form to grant the researcher access to their employee(s). This consent form includes reassurance of the independent nature of the data being collected and that both participants and their organisations are not specifically being studied nor their performance being judged. Confidentiality and non-attribution are also both emphasised in the form. These elements are expected to create a good level of confidence between managers and the researcher, and thus lower the risk of any possible adverse impact to the participant (employee).

Once managers decide they are happy to grant the researcher access to their employees, the Organisation Consent Form will instruct them to send a general message to their employees asking those interested to volunteer by contacting the researcher directly. This should reduce the possibility of disclosing the identity of those who choose to participate.

To further lower the risk of participants being approached by their managers after the interviews are run, the managers will receive before the interview date a courtesy appreciation letter. The letter will first thank them for helping to advance practice by giving the researcher access to their employee(s), and will then include a courteous statement emphasising the ethical rights of participants and the importance of complying with and respecting those rights.

E. *Social and Cultural Sensitivity (including the obligations of the Treaty of Waitangi)*

E.1. What familiarity does the researcher have with the social and cultural context of the participants?

The researcher has worked as a specialized software practitioner in two multi-national organisations in Saudi Arabia. As the participants may be from a variety of cultures, mirroring the multi-cultural nature of the workforce in Auckland, the researcher's previous experience in those multi-national organisations will help him to manage this social and cultural context. The researcher has also undertaken teaching activities at Auckland University of Technology which has offered some exposure to the local social and cultural values. The researcher's supervisors are familiar with the social and cultural context of the likely participants, having worked in the industry and taught recent students from a similarly diverse cultural background.

E.2. What consultation has occurred?

Research procedures should be appropriate to the participants. Researchers have a responsibility to inform themselves of, and take the steps necessary to respect the values, practices, and beliefs of the cultures and social groups of all participants. This usually requires consultation or discussion with appropriate people or groups to ensure that the language and research approaches being used are relevant and effective. Consultation should begin as early as possible when designing the project and should continue throughout its duration.

All researchers are encouraged to make themselves familiar with Te Ara Tika: Guidelines for Maori Research Ethics: A framework for researchers and ethics committee members which is able to be accessed through the Research Ethics website. Researchers may also find Te Kaahui Maangai a directory of Iwi and Maaori organisations to be helpful. This may be accessed via the Te Puni Kookiri website (http://www.tkm.govt.nz/). As well as these documents, the Health Research Council has published Pacific Health Research Guidelines, and Guidelines on research involving children. (see http://www.hrc.govt.nz). There are also guidelines by various organisations about researching with other populations that researchers will find helpful.

E.2.1. With whom has the consultation occurred?

Please provide written evidence that the consultation has occurred.

No formal consultation has taken place

E.2.2. How has this consultation affected the design and practice of this research?

Not applicable.

E.3. Does this research target Māori participants? ☐ Yes ☒ No

All researchers are encouraged to make themselves familiar with Te Ara Tika: Guidelines for Maori Research Ethics: A framework for researchers and ethics committee members

If your answer is 'No', please go to section E.4 and continue from there. If you answered 'Yes', please answer the next question.

E.3.1. Which iwi or hapu are involved?

E.4. Does this research target participants of particular cultures or social groups? ☒ Yes ☐ No

AUTEC defines the phrase 'specific cultures or social groups' broadly. In section 2.5 of Applying for Ethics Approval: Guidelines and Procedures it uses the examples of Chinese mothers and paraplegics. This is to identify their distinctiveness, the first as a cultural group, the second as a social group. Other examples of cultural groups may be Korean students, Samoan husbands, Cook Islanders etc., while other examples of social groups may be nurse aides, accountants, rugby players, rough sleepers (homeless people who sleep in public places) etc. Please refer to Section 2.5 of AUTEC's Applying for Ethics Approval: Guidelines and Procedures (accessible in the Ethics Knowledge Base online via http://www.aut.ac.nz/about/ethics) and to the relevant Frequently Asked Questions section in the Ethics Knowledge Base.

If your answer is 'No', please go to section E.5 and continue from there. If you answered 'Yes', please answer the next question.

E.4.1. Which cultures or social groups are involved?

The agile software development community.

E.5. Does this research focus on an area of research that involves Treaty obligations? ☐ Yes ☒ No

All researchers are encouraged to make themselves familiar with Te Ara Tika: Guidelines for Maori Research Ethics: A framework for researchers and ethics committee members.

If your answer is 'No', please go to section E.6 and continue from there. If you answered 'Yes', please answer the next question.

E.5.1. Which treaty obligations are involved?

E.6. Will the findings of this study be of particular interest to specific cultures or social groups? ☒ Yes ☐ No

If the answer is 'Yes' please answer E.6.1 and the following sections, otherwise please answer F.1 and continue from there.

E.6.1. To which iwi, hapū, culture or social groups will the findings be of interest?

The agile software development community, and the software visualisation research community.

E.6.2. How will the findings be made available to these groups?

The findings of this research will be published in top ranking Journals and conferences. The doctoral thesis will also be held in the AUT library and will be accessible through publicly available links.

F. *Respect for the Vulnerability of Some Participants*

"Vulnerable persons are those who are relatively (or absolutely) incapable of protecting their own interests. More formally, they may have insufficient power, intelligence, education, resources, strength, or other needed attributes to protect their own interests. Individuals whose willingness to volunteer in a research study may be unduly influenced by the expectation, whether justified or not, of benefits associated with participation, or of a retaliatory response from senior members of a hierarchy in case of refusal to participate may also be considered vulnerable." (Standards and Operational Guidance for Ethics Review of Health-Related Research with Human Participants, World Health Organisation).

F.1. Will your research involve any of the following groups of participants? ☐ Yes ☒ No

If your research involves any of these groups of participants, please clearly indicate which ones and then answer F.2 and the following section, otherwise please answer G.1 and continue from there.

☐ *people unable to give informed consent?* ☐ *your (or your supervisor's) own students?*
☐ *preschool children?* ☐ *children aged between five and sixteen years?*
☐ *legal minors aged between sixteen and twenty years?*
☐ *People lacking the mental capacity for consent?*
☐ *people in a dependent situation (e.g. people with a disability, or residents of a hospital, nursing home or prison or patients highly dependent on medical care)?*
☐ *people who are vulnerable for some other reason (e.g. the elderly, persons who have suffered abuse, persons who are not competent in English, new immigrants)? – please specify*

F.2. How is respect for the vulnerability of these participants reflected in the design and practice of your research?

F.3. What consultation has occurred to ensure that this will be effective?

Please provide evidence of the consultation that has occurred.

G. *Informed and Voluntary Consent*

G.1. How will information about the project be given to potential participants?

A copy of all information that will be given to prospective participants is to be attached to this Application Form. If written information is to be provided to participants, you are advised to use the Information Sheet exemplar. The language in which the information is provided is to be appropriate to the potential participants and translations need to be provided when necessary.

A Participant Information Sheet will be sent to those who express and interest in participating.

G.2. How will the consent of participants be obtained and evidenced?

AUTEC requires consent to be obtained and usually evidenced in writing. A copy of the Consent Form which will be used is to be attached to this application. If this will not be the case, please provide a justification for the alternative approach and details of the alternative consent process. Please note that consent must be obtained from any participant aged 16 years or older. Participants under 16 years of age are unable to give consent, which needs to be given by their parent or legal guardian. AUTEC requires that participants under the age of 16 assent to their participation. When the nature of the research requires it, AUTEC may also require that consent be sought from parents or legal guardians for participants aged between 16 and twenty years. For further information please refer to AUTEC's Applying for Ethics Approval: Guidelines and Procedures.

Participants will sign and return a Participant Consent form that will be sent to them along with the information sheet prior to the interview date.

G.3. Will any of the participants have difficulty giving informed consent on their own behalf? ☐ Yes ☒ No

Please consider physical or mental condition, age, language, legal status, or other barriers.

If the answer is 'Yes' please answer G.3.1 and the following sections, otherwise please answer G.4 and continue from there.

G.3.1. If participants are not competent to give fully informed consent, who will consent on their behalf?

Researchers are advised that the circumstances in which consent is legally able to be given by a person on behalf of another are very constrained. Generally speaking, only parents or legal guardians may give consent on behalf of a legal minor and only a person with an enduring power of attorney may give consent on behalf of an adult who lacks capacity.

G.3.2. How will these participants be asked to provide assent to participation?

Whenever consent by another person is possible and legally acceptable, it is still necessary to take the wishes of the participant into account, taking into consideration any limitations they may have in understanding or communicating them.

G.4. Is there a need for translation or interpreting? ☐ Yes ☒ No

If your answer is 'Yes', please provide copies of any translations with this application and any Confidentiality Agreement required for translators or interpreters.

H. *Respect for Rights of Privacy and Confidentiality*

H.1. How will the researchers respect the privacy and confidentiality of participants?

Please note that anonymity and confidentiality are different. For AUTEC's purposes, 'Anonymity' means that the researcher is unable to identify who the participant is in any given case. If the participants will be anonymous, please state how, otherwise, if the researcher will know who the participants are, please describe how the participants' privacy issues and the confidentiality of their information will be managed.

- The interviews must be in a private, soundproof area.
- The forms must be collected and taken off site at regular intervals.
- The participants must be reassured that their performance is not being monitored. The research is about what kinds of information is important in the process.
- The participants must be reassured that the way they use the current practices is the right way, and that is what is being studied.
- The researcher must not talk about the interview results with participant's colleagues or employer.

- The participants must be clearly reassured that confidentiality of their data will be strictly enforced and respected.

H.2. Will any participants be identifiable in the research outputs or findings? ☐ Yes ☒ No

If your answer is 'Yes", please answer H.2.1, otherwise please answer H.3

H.2.1. What level of confidentiality is able to be offered to participants and how will this be managed?

If the research involves small or distinctive groups of participants or procedures such as interviews conducted at the worksite, or focus groups with peers, researchers should identify the level of participant confidentiality that can be offered in the Information Sheet. If participants or groups will be identified, please state why this is appropriate, how this will happen, and how the participants will give consent.

H.3. What information on the participants will be obtained from third parties?

This includes use of third parties, such as employers or professional organisations, in recruitment.

None

H.4. How will potential participants' contact details be obtained for the purposes of recruitment?

Some contact details are already available (existing relationships). Some will be collected in person during networking opportunities at the meetings of Agile Auckland. Some will be contacted through LinkedIn. The combined resources of the primary researcher and the applicant will be utilised

H.5. What identifiable information on the participants will be given to third parties?

None

H.6. Who will have access to the data during the data collection and analysis stages?

The primary researcher and supervisors

H.7. Who will have access to the data after the findings have been produced?

The primary researcher and supervisors

H.8. Are there any plans for the future use of the data beyond those already described? ☐ Yes ☒ No

The applicant's attention is drawn to the requirements of the Privacy Act 1993 (see Appendix I of AUTEC's Applying for Ethics Approval: Guidelines and Procedures). Information may only be used for the purpose for which it was collected so if there are plans for the future use of the data, then this needs to be explained in the Information Sheets for participants. If you have answered 'Yes' to this question, please answer section H.8.1.1 and continue from there. If you answered 'No' to this question, please go to section H.9 and proceed from there.

H.8.1.1 If data will be stored in a database, who will have access to that information, how will it be used, for what will it be used, and how have participants consented to this?

H.8.1.2 Will any contact details be stored for future use and if so, who will have access to them, how will they be used, for what will they be used, and how have participants consented to this?

H.9. Where will the data be stored once the analysis is complete?

Please provide the exact storage location. AUTEC normally requires that the data be stored securely on AUT premises in a location separate from the consent forms. Electronic data should be downloaded to an external storage device (e.g. an external hard drive, a memory stick etc.) and securely stored. If you are proposing an alternative arrangement, please explain why.

The data will be stored at AUT SECMS building (AUT Tower – WT704A). Paper documents will be stored in a locked filing drawer along with digital material stored on USB thumb drive.

H.9.1. For how long will the data be stored after completion of analysis?

AUTEC normally requires that the data be stored securely for a minimum of six years, or ten years for health data. If you are proposing an alternative arrangement, please explain why.

Six years.

H.9.2. How will the data be destroyed?

If the data will not be destroyed, please explain why, identify how it will be safely maintained, and provide appropriate informed consent protocols.

After the specified interval has expired, the material (tapes, papers, CD/DVD) will be destroyed by breaking and/or burning so the content cannot be accessed in any way.

H.10. Who will have access to the Consent Forms?

The primary researcher and the supervisors

H.11. Where will the completed Consent Forms be stored?

Please provide the exact storage location. AUTEC normally requires that the Consent Forms be stored securely on AUT premises in a location separate from the data. If you are proposing an alternative arrangement, please explain why.

The consent forms will be stored at AUT's WZ Science Building (Room WZ626).

H.11.1. For how long will the completed Consent Forms be stored?

AUTEC normally requires that the Consent Forms be stored securely for a minimum of six years, or ten years in the case of research involving health data. If you are proposing an alternative arrangement, please explain why.

Six years.

H.11.2. How will the Consent Forms be destroyed?

If the Consent Forms will not be destroyed, please explain why.

After the specified interval has expired, the consent forms will be shredded so the content cannot be accessed in any way.

H.12. Does your research involve the collection of personally identifiable and sensitive data? ☐ Yes ☒ No

Sensitive data can be used to identify an individual, object or location and has a risk of discrimination, harm or unwanted attention. Sensitive data potentially poses a substantial threat to those who are or who have been involved in it, especially if it is shared inappropriately, or if it falls into the wrong hands. If you have answered 'Yes' please identify what data is being collected and how it is sensitive and provide a Data Safety Management Protocol (see the Forms section of the Research Ethics website for a guide to drafting one). If the answer is 'No', please answer H.13 and continue from there.

H.13. Does your project involve the use of previously collected information or biological samples for which there was no explicit consent for this research? ☐ Yes ☒ No

If the answer is 'Yes' please answer H.13.1 and the following sections, otherwise please answer H.14 and continue from there.

H.13.1. What previously collected data will be involved?

H.13.2. Who collected the data originally?

H.13.2.1 Why was the information originally collected?

H.13.2.2 For what purposes was consent originally given when the information was collected?

H.13.3. How will the data be accessed?

H.14. Does your research involve the collection of information about organisational practices? ☒ Yes ☐ No

AUTEC applies a broad definition to the term 'organisations'. It could include for example, businesses, hospitals or clinics, schools, or sports clubs and teams If the answer is 'Yes' please answer H.14.1, otherwise please answer I.1 and continue from there.

H.14.1. How will the authorisation to access the organisation or its staff for research purposes be obtained?

Employer's will be contacted officially by the researcher or main applicant in order to obtain their approval/permission.

H.14.2. Could disclosure of this information potentially disadvantage the oganisation or the participants? ☒ Yes ☐ No

If your answer is 'Yes", please answer H.14.2.1, otherwise please answer H.14.3

H.14.2.1 How will the risks associated with potential disadvantages be managed?

The research collects information on the business process employed by the organisation. Disclosure of this information could potentially be used by competitors in a manner that might disadvantage the organisation for which the participants works for.

H.14.3. Will the particpants or anyone else in the oganisation be identified in this information? ☐ Yes ☒ No

If your answer is 'Yes", please answer H.14.3.1, otherwise please answer I.1 and continue from there.

H.14.3.1 How will the potential risks involved be managed?

If the research involves procedures such as interviews conducted at the worksite, or focus groups with peers, researchers should identify the level of participant confidentiality that can be offered in the Information Sheet.

The employer and participants will be reassured of the privacy and confidentiality of their collected information. No participant or organisation will be identified on any form of the recorded data. Information obtained from the interviews will also be kept in private locations as stated above. All results will also be completely anonymised before they are published.

I. *Minimisation of risk*

I.1. Risks to Participants

Please consider the possibility of moral, physical, psychological or emotional risks to participants, including issues of confidentiality and privacy, from the perspective of the participants, and not only from the perspective of someone familiar with the subject matter and research practices involved. Please clearly state what is likely to be an issue, how probable it is, and how this will be minimised or mitigated (e.g. participants do not need to answer a question that they find embarrassing, or they may terminate an interview, or there may be a qualified counsellor present in the interview, or the findings will be reported in a way that ensures that participants cannot be individually identified, etc.) Possible risks and their mitigation should be fully described in the Information Sheets for participants.

I.1.1. How much time will participants be required to give to the project?

This research will be based primarily on semi-structured interviews, therefore we estimate sixty minutes will be needed for the initial interview and thirty minutes for notes checking. However, if some of the participants willingly have more information to provide, the time may exceed 60 minutes but will not go beyond 2 hours.

I.1.2. What level of discomfort or embarrassment may participants be likely to experience?

- The participant may feel uncomfortable about having their interview recorded.
- The participant may feel uncomfortable that their colleagues or managers may overhear what they may say during the interview.

I.1.3. In what ways might participants be at risk in this research?

- The participant may express views of which the company may disapprove and this could have detrimental consequences
- The participant may accidentally divulge commercially sensitive information that has detrimental consequences

I.1.4. In what ways are the participants likely to experience risk or discomfort as a result of cultural, employment, financial or similar pressures?

- The participants may fear their opinions expressed in the interviews are not confidential.
- The participants might fear that by answering some questions they might be at risk of accidentally divulging their organisational practices that are deemed confidential
- The participants may feel their personal performance is being monitored.

This will be addressed by explicitly reassuring the participant at the beginning of their interview that their data are strictly confidential and that all results are anonymised. They will also be reminded that nothing will be disclosed to their colleagues or employers. The same is also clearly communicated in the information sheet.

I.1.5. Will your project involve processes that are potentially disadvantageous to a person or group, such as the collection of information, images etc. which may expose that person/group to discrimination, criticism, or loss of privacy? ☐ Yes ☒ No

If your answer is 'Yes', please detail how these risks will be managed and how participants will be informed about them.

I.1.6. Will your research involve collection of information about illegal behaviour(s) which could place the participants at current or future risk of criminal or civil liability or be damaging to their financial standing, employability, professional or personal relationships? ☐ Yes ☒ No

If your answer is 'Yes', please detail how these risks will be managed and how participants will be informed about them.

I.1.7. If the participants are likely to experience any significant discomfort, embarrassment, incapacity, or psychological disturbance, please state what consideration you have given to the provision of counselling or post-interview support, at no cost to the participants, should it be required.

Adult research participants in Auckland are able to utilise counselling support from the AUT Counselling Team, otherwise you may have to consider local providers for participants who are located nationwide, or in some particular geographical area or who are children. You may discuss the potential for participant psychological impact or harm with the Head of AUT Counselling, if you require. Please check the relevant Frequently Asked Question on the research ethics website as well and ensure the appropriate wording in included in the Information Sheet when counselling opportunities need to be offered.

Very low. Unlikely to need any counselling.

I.1.8. Will any use of human remains, tissue or body fluids which does not require submission to a Health and Disability Ethics Committee occur in the research? ☐ Yes ☒ No

e.g. finger pricks, urine samples, etc. (please refer to section 13 of AUTEC's Applying for Ethics Approval: Guidelines and Procedures). If your answer is yes, please provide full details of all arrangements, including details of agreements for treatment, how participants will be able to request return of their samples in accordance with right 7 (9) of the Code of Health and Disability Services Consumers' Rights, etc.

I.1.9. Will this research involve potentially hazardous substances? ☐ Yes ☒ No

e.g. radioactive material, biological substances (please refer to section 15 of AUTEC's Applying for Ethics Approval: Guidelines and Procedures and the Hazardous Substances and New Organisms Act 1996).

If the answer is 'Yes', please provide full details, including hazardous substance management plan.

I.2. Risks to Researchers

If this project will involve interviewing participants in private homes, undertaking research overseas, in unfamiliar cultural contexts, or going into similarly vulnerable situations, then a Researcher Safety protocol should be designed and appended to this application. This should identify simple and effective processes for keeping someone informed of the researcher's whereabouts and provide for appropriate levels of assistance.

I.2.1. Are the researchers likely to be at risk? ☐ Yes ☒ No

If the answer is 'Yes' please answer I.2.1.1 and then continue, otherwise please answer I.3 and continue from there.

I.2.1.1 In what ways might the researchers be at risk and how will this be managed?

I.3. Risks to AUT

I.3.1. Is AUT or its reputation likely to be at risk because of this research? ☐ Yes ☒ No

If the answer is 'Yes' please answer I.3.1.1 and then continue, otherwise please answer I.3.2 and continue from there.

I.3.1.1 In what ways might AUT be at risk in this research?

Please identify how and detail the processes that will be put in place to minimise any harm.

I.3.2. Are AUT staff and/or students likely to encounter physical hazards during this project? ☐ Yes ☐ No

If yes, please provide a hazard management protocol identifying how harm from these hazards will be eliminated or minimised.

J. *Truthfulness and limitation of deception*

J.1. How will feedback on or a summary of the research findings be disseminated to participants (individuals or groups)?

Please ensure that this information is included in the Information Sheet.

A summarised report will be distributed to interested participants, preferably via email.

J.2. Does your research include any deception of the participants, such as non-disclosure of aims or use of control groups, concealment, or covert observations? ☐ Yes ☒ No

Deception of participants in research may involve deception, concealment or covert observation. Deception of participants conflicts with the principle of informed consent, but in some areas of research it may sometimes be justified to withhold information about the purposes and procedures of the research. Researchers must make clear the precise nature and extent of any deception and why it is thought necessary. Emphasis on the need for consent does not mean that covert research can never be approved. Any departure from the standard of properly informed consent must be acceptable when measured against possible benefit to the participants and the importance of the knowledge to be gained as a result of the project or teaching session. This must be addressed in all applications. Please refer to Section 2.4 of AUTEC's Applying for Ethics Approval: Guidelines and Procedures when considering this question.

If the answer is 'Yes' please answer J.2.1 and the following sections, otherwise please answer J.3 and continue from there.

J.2.1. Is deception involved?

J.2.2. Why is this deception necessary?

J.2.3. How will disclosure and informed consent be managed?

J.3. Will this research involve use of a control group? ☐ Yes ☒ No

If the answer is 'Yes' please answer J.3.1 and the following sections, otherwise please answer K.1 and continue from there.

J.3.1. How will the Control Group be managed?

J.3.2. What percentage of participants will be involved in the control group?

J.3.3. What information about the use of a control group will be given to the participants and when?

K. *Avoidance of Conflict of Interest*

Researchers have a responsibility to ensure that any conflict between their responsibilities as a researcher and other duties or responsibilities they have towards participants or others is adequately managed. For example, academic staff members who propose to involve their students as participants in research need to ensure that no conflict arises between their roles as teacher and researcher, particularly in view of the dependent relationship between student and teacher, and of the need to preserve integrity in assessment processes. Likewise researchers have a responsibility to ensure that any conflict of interest between participants is adequately managed for example, managers participating in the same research as their staff.

K.1. What conflicts of interest are likely to arise as a consequence of the researchers' professional, social, financial, or cultural relationships?

None, because the study will not incorporate participants with close or personal relationship with the researcher.

K.2. What possibly coercive influences or power imbalances are there in the professional, social, financial, or cultural relationships between the researchers and the participants or between participants (e.g. dependent relationships such as teacher/student; parent/child; employer/employee; pastor/congregation etc.)?

There could possibly be influence exerted by employer on their participating employees with respect to openness and candour, by asking them for example to disguise some of their workplace practices, or to limit the extent of information they can express.

K.3. How will these conflicts of interest, coercive influences or power imbalances be managed through the research's design and practice and how will any adverse effects that may arise from them be mitigated?

This research is looking at current agile development processes as practiced at local organisations including the workflow, work dependencies, and tools—which are all independent elements of people and their capabilities. No performance or individual capabilities will be judged, so the outcomes will not affect the performance of any participant.

Moreover, the results will be presented based on workflow practices and processes as observed in the overall agile software development process practiced in industry without any identification of individual participants or organisations. All data will be carefully de-identified before it is used in analysis or reporting.

This information and measures aimed at protecting the confidentiality of participants and their organisations will be clearly communicated to participating organisations and their employees in order to create a level of confidence, comfort, and assurance, that none of their information will be divulged and that required precautions are in place to ensure this is fully enforced. For participants, this will be communicated in the Participant Information sheet, whereas for Organisations, this will be communicated in the Organisation Consent Form, and will be reinforced in a subsequent Appreciation Letter after receiving the signed consent form from each participant's manager (or person in charge).

Individual participants will also be reassured verbally before interview that none of their interview data will be disclosed to their colleagues or employers.

K.4. Does your project involve payments or other financial inducements (including koha, reasonable contribution towards travel expenses or time, or entry into a modest prize draw) to participants?

☐ Yes ☒ No

If the answer is 'Yes' please answer K.4.1 and the following sections, otherwise please answer K.5 and continue from there.

K.4.1. What form will the payment, inducement, or koha take?

K.4.2. Of what value will any payment, gift or koha be?

K.4.3. Will potential participants be informed about any payment, gift or koha as part of the recruitment process, and if so, why and how?

K.5. Have any applications for financial support for this project been (or will be) made to a source external to AUT?

☐ Yes ☒ No

If the answer is 'Yes' please answer K.5.1 and the following sections, otherwise please answer K.6 and continue from there.

K.5.1. What financial support for this project is being provided (or will be provided) by a source external to AUT?

K.5.2. Who is the external funder?

K.5.3. What is the amount of financial support involved?

K.5.4. How is/are the funder/s involved in the design and management of the research?

K.6. Have any applications been (or will be) submitted to an AUT Faculty Research Grants Committee or other AUT funding entity?

☐ Yes ☒ No

If the answer is 'Yes' please answer K.6.1 and the following sections, otherwise please answer K.7 and continue from there.

K.6.1. What financial support for this project is being provided (or will be provided) by an AUT Faculty Research Grants Committee or other AUT funding entity?

K.6.2. What is the amount of financial support involved?

K.6.3. How is/are the funder/s involved in the design and management of the research?

K.7. Is funding already available, or is it awaiting decision?

Already approved.

K.8. Do the applicant or the researchers, investigators or research organisations mentioned in Part B of this application have any financial interests in the outcome of this project? ☐ Yes ☒ No

If the response is 'Yes', please provide full details about the financial interests and how any conflicts of interest are being managed, otherwise, please respond to section K.9 and continue from there.

K.9. Are the participants expected to pay in any way for any services associated with this research? ☐ Yes ☒ No

If the response is 'Yes', please provide full details about the charges and describe how any benefits will balance the burdens involved as well as how any conflicts of interest are being managed. Otherwise please respond to section L.1 and continue from there.

L. *Respect for Property*

Researchers must ensure that processes do not violate or infringe legal or culturally determined property rights. These may include factors such as land and goods, works of art and craft, spiritual treasures and information.

L.1. Will this research impact upon property owned by someone other than the researcher? ☐ Yes ☒ No

If the answer is 'Yes' please answer L.1.1 and the following sections, otherwise please answer L.2 and continue from there.

L.1.1. How will this be managed?

L.2. How do contexts to which copyright or Intellectual Property apply (e.g. research instruments, social media, virtual worlds etc.) affect this research and how will this be managed?

Particular attention should be paid to the legal and ethical dimensions of intellectual property. Care must be taken to acknowledge and reference the ideas of all contributors and others and to obtain any necessary permissions to use the intellectual property of others. Teachers and researchers are referred to AUT's Intellectual Property Policy for further guidance.

M. *References*

Please include any references relating to your responses in this application in the standard format used in your discipline.

Alshakhouri, M. A. J. (2013). *ScrumCity: synchronised visualisation of software process and product artefacts*. Retrieved from http://aut.researchgateway.ac.nz/handle/10292/5595

Background, P., Contact, P., & Bible, R. V. (2018). Hevner et al 2004: Design Science in Information Systems Research Hevner, 1–3.

Curcio, K., Navarro, T., Malucelli, A., & Reinehr, S. (2018). Requirements engineering: A systematic mapping study in agile software development. *Journal of Systems and Software*, *139*, 32–50. https://doi.org/10.1016/j.jss.2018.01.036

Duru, H. A., Çakir, M. P., & Işler, V. (2013). How Does Software Visualization Contribute to Software Comprehension? A Grounded Theory Approach. *International Journal of Human-Computer Interaction*, *29*(11), 743–763. https://doi.org/10.1080/10447318.2013.773876

Fittkau, F., Krause, A., & Hasselbring, W. (2017). Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology*, *87*, 259–277. https://doi.org/10.1016/j.infsof.2016.07.004

Görtler, J., Schulz, C., Weiskopf, D., & Deussen, O. (2018). Bubble Treemaps for Uncertainty Visualization. *IEEE Transactions on Visualization and Computer Graphics*, *24*(1), 719–728. https://doi.org/10.1109/TVCG.2017.2743959

Hahn, S., Trümper, J., Moritz, D., & Döllner, J. (2014). Visualization of varying hierarchies by stable layout of Voronoi treemaps. *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*, 50–58.

Hess, A., Diebold, P., & Seyff, N. (2018). Understanding information needs of agile teams to improve requirements communication (Special issue edited by Nan Niu and Daniel Mendez). *Journal of Industrial Information Integration*, (April). https://doi.org/10.1016/j.jii.2018.04.002

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105. https://doi.org/10.2307/25148625

Limberger, D., Scheibel, W., Trapp, M., & Dollner, J. (2017). Mixed-Projection Treemaps: A Novel Approach Mixing 2D and 2.5D Treemaps. In *2017 21st International Conference Information Visualisation (IV)* (pp. 164–169). IEEE. https://doi.org/10.1109/iV.2017.67

Lopez-Herrejon, R. E., Illescas, S., & Egyed, A. (2018). A systematic mapping study of information visualization for software product line engineering. *Journal of Software: Evolution and Process*, *30*(2), 1–18. https://doi.org/10.1002/smr.1912

Merino, L., Ghafari, M., Anslow, C., & Nierstrasz, O. (2018). A systematic literature review of software visualization evaluation. *Journal of Systems and Software*, *144*(October 2017), 165–180. https://doi.org/10.1016/j.jss.2018.06.027

Müller, R., Kovacs, P., Schilbach, J., Eisenecker, U., Zeckzer, D., & Scheuermann, G. (2014). A Structured Approach for Conducting a Series of Controlled Experiments in Software Visualization. *Proceedings of the 5th International Conference on Visualization Theory and Applications*, 204–209. https://doi.org/10.5220/0004835202040209

Muller, R., Kovacs, P., Schilbach, J., & Zeckzer, D. (2015). How to master challenges in experimental evaluation of 2D versus 3D software visualizations. *2014 IEEE VIS International Workshop on 3DVis, 3DVis 2014*, 33–36. https://doi.org/10.1109/3DVis.2014.7160097

Oppl, S. (2017). *Supporting the Collaborative Construction of a Shared Understanding About Work with a Guided Conceptual Modeling Technique*. *Group Decision and Negotiation* (Vol. 26). https://doi.org/10.1007/s10726-016-9485-7

Paredes, J., Anslow, C., & Maurer, F. (2014). Information visualization for agile software development. *Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014*, 157–166. https://doi.org/10.1109/VISSOFT.2014.32

Petre, M. (2010). Mental imagery and software visualization in high-performance software development teams. *Journal of Visual Languages and Computing*, *21*(3), 171–183. https://doi.org/10.1016/j.jvlc.2009.11.001

Petre, M., Blackwell, A., & Green, T. R. G. (1998). *Cognitive questions in software visualization*. *Software visualization:* https://doi.org/citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.8820

Rivero, J. M., Grigera, J., Distante, D., Montero, F., & Rossi, G. (2017). DataMock: An Agile Approach for Building Data Models from User Interface Mockups. *Software and Systems Modeling*, 1–28. https://doi.org/10.1007/s10270-017-0586-9

Roberts, J. C., Ritsos, P. D., Jackson, J. R., & Headleand, C. (2018). The Explanatory Visualization Framework: An Active Learning Framework for Teaching Creative Computing Using Explanatory Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, *24*(1), 791–801. https://doi.org/10.1109/TVCG.2017.2745878

Scheibel, W., Weyand, C., & Döllner, J. (2018). EvoCells - A Treemap Layout Algorithm for Evolving Tree Data. *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, *3*(Visigrapp), 273–280. https://doi.org/10.5220/0006617102730280

Seriai, A., Benomar, O., Cerat, B., & Sahraoui, H. (2014). Validation of Software Visualization Tools: A Systematic Mapping Study. *2014 Second IEEE Working Conference on Software Visualization*, 60–69. https://doi.org/10.1109/VISSOFT.2014.19

Slob, G.-J., Dalpiaz, F., Brinkkemper, S., & Lucassen, G. (2018). The interactive narrator tool: Effective requirements exploration and discussion through visualization. *CEUR Workshop Proceedings*, *2075*.

Sondag, M., Speckmann, B., & Verbeek, K. (2018). Stable Treemaps via Local Moves. *IEEE Transactions on Visualization and Computer Graphics*, *24*(1), 729–738. https://doi.org/10.1109/TVCG.2017.2745140

Storey, M.-A. D., Fracchia, F. ., & Muller, H. A. (1999). Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97* (Vol. 44, pp. 17–28). IEEE Comput. Soc. Press. https://doi.org/10.1109/WPC.1997.601257

Tak, S., & Cockburn, A. (2013). Enhanced spatial stability with hilbert and moore treemaps. *IEEE Transactions on Visualization and Computer Graphics*, *19*(1), 141–148. https://doi.org/10.1109/TVCG.2012.108

van Wijk, J. J. (2005). The Value of Visualization. In *VIS 05. IEEE Visualization, 2005.* (pp. 79–86). IEEE. https://doi.org/10.1109/VISUAL.2005.1532781

Wettel, R., Lanza, M., & Robbes, R. (2011). Software Systems as Cities: A Controlled Experiment. *Proceeding of the 33rd International Conference on Software Engineering - ICSE '11*, 551. https://doi.org/10.1145/1985793.1985868

N. *Checklist*

Please ensure all applicable sections of this form have been completed and all appropriate documentation is attached as incomplete applications will not be considered by AUTEC.

Have you discussed this application with your AUTEC Faculty Representative, the Executive Secretary, or the Ethics Coordinator?	☒ *Yes* ☐ *No*
Is this application related to an earlier ethics application? If yes, please provide the application number of the earlier application.	☐ *Yes* ☒ *No*
Are you seeking ethics approval from another ethics committee for this research? If yes, please identify the other committee.	☐ *Yes* ☒ *No*

Section A	*Project information provided*	☒
Section B	*Research Adequacy information provided*	☒
Section C	*Project details provided*	☒
Section D	*Three Principles information provided*	☒
Section E	*Social and Cultural Sensitivity information provided*	☒
Section F	*Vulnerability information provided*	☒
Section G	*Consent information provided*	☒
Section H	*Privacy information provided*	☒
Section I	*Risk information provided*	☒
Section J	*Truthfulness information provided*	☒
Section K	*Conflict of Interest information provided*	☒
Section L	*Respect for Property information provided*	☒
Section M	*References provided*	☒
Section N	*Checklists completed*	☒
Section O.1 and 2	*Applicant and student declarations signed and dated*	☒
Section O.3	*Authorising signature provided*	☒

Spelling and Grammar Check (please note that a high standard of spelling and grammar is required in documents that are issued with AUTEC approval)

Attached Documents (where applicable)

Participant Information Sheet(s)	☒
Consent Form(s)	☒
Questionnaire(s)	☐
Indicative Questions for Interviews or Focus Groups	☒
Observation Protocols	☐
Recording Protocols for Tests	☐
Advertisement(s)	☐
Researcher Safety Protocol	☐
Hazardous Substance Management Plan	☐
Any Confidentiality Agreement(s)	☐
Any translations that are needed	☐
Other Documentation	☐

O. *Declarations*

O.1. Declaration by Applicant

Please tick the boxes below.

☒ ***The information in this application is complete and accurate to the best of my knowledge and belief. I take full responsibility for it.***

☒ ***In conducting this study, I agree to abide by all applicable laws and regulations, and established ethical standards contained in AUTEC's Applying for Ethics Approval: Guidelines and Procedures and internationally recognised codes of ethics.***

☒ ***I will continue to comply with AUTEC's Applying for Ethics Approval: Guidelines and Procedures, including its requirements for the submission of annual progress reports, amendments to the research protocols before they are used, and completion reports.***

☒ ***I understand that brief details of this application may be made publicly available and may also be provided to the Graduate Research School, the Research and Innovation Office, or the University's insurers for purposes relating to AUT's interests.***

5 April, 2019

Signature ***Date***

O.2. Declaration by Student Researcher

Please tick the boxes below.

☒ ***The information in this application is complete and accurate to the best of my knowledge and belief.***

☒ ***In conducting this study, I agree to abide by all applicable laws and regulations, and established ethical standards contained in AUTEC's Applying for Ethics Approval: Guidelines and Procedures and internationally recognised codes of ethics.***

☒ ***I will continue to comply with AUTEC's Applying for Ethics Approval: Guidelines and Procedures, including its requirements for the submission of annual progress reports, amendments to the research protocols before they are used, and completion reports.***

☒ ***I understand that brief details of this application may be made publicly available and may also be provided to the Graduate Research School, the Research and Innovation Office, or the University's insurers for purposes relating to AUT's interests.***

5 April, 2019

Signature ***Date***

O.3. Authorisation by Head of Faculty/School/Programme/Centre

Please tick the boxes below.

☐ ***The information in this application is complete and accurate to the best of my knowledge and belief.***

☐ ***In authorising this study, I declare that the applicant is adequately qualified to undertake or supervise this research and that to the best of my knowledge and belief adequate resources are available for this research and all appropriate local research governance issues have been addressed.***

☐ ***I understand that brief details of this application may be made publicly available and may also be provided to the Graduate Research School, the Research and Innovation Office, or the University's insurers for purposes relating to AUT's interests.***

d April, 2019

Signature ***Date***

Notes for submitting the completed application for review by AUTEC

- Please ensure that you are using the current version of this form before submitting your application.
- Please ensure that all questions on the form have been answered and that no part of the form has been deleted.
- Please provide ***one*** printed, single sided, A4, and signed copy of the application and all related documents.
- Please deliver or post to the AUTEC Secretariat, room WU406, fourth floor, WU Building, City Campus. The internal mail code is D-88. The courier address is 46 Wakefield Street, Auckland 1010. Alternatively, please hand the application to the Research Ethics Advisor in person at one of the Drop In sessions at any of the four campuses (http://www.aut.ac.nz/researchethics/resources/workshops-and-drop-inns).
- Applications should be submitted once they have been finalised. For a particular meeting it needs to have been received in the AUTEC Secretariat by midday on the relevant agenda closing day [AUTEC's meeting dates are listed in the website at http://www.aut.ac.nz/researchethics]
- If sending applications by internal mail, please post them at least two days earlier to allow for any delay that may occur.
- Late applications will be placed on the agenda for the following meeting.

MINIMAL RISK CHECKLIST

Your application may be appropriate for an expedited review if it poses no more than minimal risk of harm to participants. To assist AUTEC's Secretariat to screen the application for assignment to the correct review pathway, please complete the following checklist:

Does the research involve any of the following?

ANONYMOUS SURVEY ASSESSMENT

		Yes	No
1	The collection of anonymous and non-sensitive survey/questionnaire data only. *(If YES is checked, the application may receive an expedited review if the data is from adults and poses no foreseeable risks to participants OR where any foreseeable risk is no more than inconvenience – no further questions on this checklist need be answered.)*	☐	☒

MINIMAL RISK ASSESSMENT[1]

		Yes	No
2	Participants who are unable to give informed consent (including children under 16 years old), or who are particularly vulnerable or in a dependent situation, (e.g. people with learning difficulties, over-researched groups, people in care facilities, or patients highly dependent on medical care)?	☐	☒
3	A reasonable expectation of causing participants physical pain beyond mild discomfort, or that experienced by the participants on an every-day basis, or any emotional discomfort, embarrassment, or psychological or spiritual harm, (e.g. asking participants to recall upsetting events)?	☐	☒
4	Research processes which may elicit information about any participant's involvement in illegal activities, or activities that represent a risk to themselves or others, (e.g. drug use or professional misconduct)?	☐	☒
5	Collection of any human tissue, blood or other samples, or invasive or intrusive physical examination or testing?	☐	☒
6	The administration of any drugs, medicines, supplements, placebo or non-food substances?	☐	☒
7	An intervention of any form of exercise, or other physical regime that is different to the participants' normal activities (e.g. dietary, sleep)?	☐	☒
8	Participants who are being asked to give information of a personal nature about their colleagues, employers, teachers, or coaches (or any other person who is in a power relationship with them), and where the identity of participants or their organisation may be inferred?	☐	☒
9	Any situation which may put the researcher at risk of harm? (E.g. gathering data in private homes)?	☐	☒
10	The use of previously collected biological samples or identifiable personal information for which there was no explicit consent for this research?	☐	☒
11	Any matters of commercially sensitive information?	☒	☐
12	Any financial interest in the outcome of the research by any member(s) of the research team?	☐	☒
13	People who are not giving consent to be part of the study, or the use of any deception, concealment or covert observations in non-public places, including social media?	☐	☒
14	Participants who are in a dependent or unequal relationship with any member(s) of the research team (e.g. where the researcher is a lecturer/ teacher/ health care provider/ coach/ employer/ manager/ or relative etc.) of any of the participants?	☐	☒

[1] If "No" is checked to all items 2-14, the application's status as Minimal Risk will be checked by the Secretariat, and may be forwarded to expedited review. Applications with more than Minimal Risk (any one "yes" to questions 2-14 above), and applications where the checklist is not completed will appear on AUTEC's next agenda.

Appendix VII: Phase One Material

This appendix includes the Evaluation Sheet & Problem set from Phase One. Six problems were originally designed based on the Cassandra dataset. Given the time limitations, the problems were reduced to 4 and were conducted by the student running the study rather than the participants.

VII.I Evaluation Problem Set

ScrumCity Preliminary Evaluation

Introduction

Thank you for being willing to participate in this preliminary evaluation of our ScrumCity tool. This sheet is intended to provide you with the information so that you are able to complete the evaluation successfully. Kindly read these opening sections before commencing your evaluation.

ScrumCity

ScrumCity is a proof of concept tool that was built to demonstrate a new software visualisation approach that aims to support software practitioners in exploratory and reasoning activities. In particular, it aims to support bidirectional traceability between software requirements and their end product implementation. ScrumCity is an early experimental prototype implementation that will help us to investigate the feasibility of this approach.

Experiment Subject (Cassandra)

In this evaluation exercise a Java-based open source project called Cassandra was selected as the subject of this initial experiment. Cassandra is a scalable distributed key-value store. Version 2.2.0 is specifically used here. A set of 48 requirement features were identified (*from the project documentation*) and trace links were then created connecting each feature to the classes that implementing it.

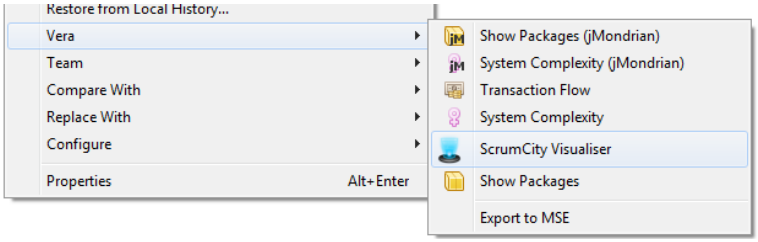
Purpose of this Evaluation

The purpose of this evaluation session is summarised as follows:

A. Evaluate any perceived overall practical benefit of the proposed approach
B. Evaluate its potential feasibility in real world scenarios
C. Investigate if the tool could provide added value in software reasoning and exploratory activities
D. Investigate if the tool could provide added value in software maintenance and refactoring activities by supporting tasks such as feature location and identification of change impact.

Experiment Setup

The session will be run on the Windows platform with an Eclipse IDE installed and a large screen to display the visualisation. ScrumCity is provided as a plugin for Eclipse, with Cassandra source code preloaded to the workspace. The user starts the experiment by simply selecting the Cassandra project and then choosing the ScrumCity visualiser from the right-click drop-down menu (see below). Interaction with the visualisation is made possible via keyboard shortcuts and a mouse. A sheet will be handed to you that visually presents all keyboard shortcuts and their functionalities.



You will be given a brief demo of the tool before commencing on the problem set below. Guidance will also be provided throughout the experiment as the tool is still in an experimental state.

Evaluation Problem Set

Problem 1

The **NodeProbe.java** class is identified as a potentially complex class that might benefit from refactoring. Using ScrumCity, locate this class and then try to asses if it might indeed benefit from some refactoring. Investigate which other classes might be potentially impacted by this refactoring.

Problem 2

The ***Built-in Row Caching*** functionality needs to be revamped completely. Kindly try to:

a) Identify the classes that need to be reworked
b) Assess the effort that might be required
c) Identify any potential impact on the other functions of the system.
d) Compare this finding with the same exercise for feature **F4**.

Problem 3

Quickly explore features **F36** up to **F42** and then very briefly report your findings with any comments or feedback/advice you might provide.

Problem 4

The ***Marshalling*** functionality (**F25**) requires some improvement. Try to quickly assess the effort that is likely to be required to do this. Compare that to the effort that would be required to add some improvements to **F16** and **F29**.

ScrumCity Questionnaire

Overview

This questionnaire is intended to collect feedback and comments after you have completed the evaluation of the ScrumCity tool. Most answers are based on a 5-point rating scale where 1 designates lowest agreement and 5 designates highest agreement.

Estimated time to complete this questionnaire is 10-15 minutes.

Question1

How many years of industrial software development experience do you have?

Answer:

Question 2

Does the above experience period included the use of Scrum development methodology (or any variant of agile methodology)? If yes, kindly specify the methodology and the number of years you have practiced it.

Answer:

Question 3:

ScrumCity was helpful in developing quick understanding of the system's structure that is being visualised:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 4:

The way that ScrumCity has presented the feature requirements in context with the visualised system structure is found generally helpful for developers:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 5:

Making the feature requirements accessible to developers in context with system's physical structure has an added value in supporting some development or maintenance activities:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 6:

ScrumCity was helpful in identifying interdependencies between the various system artefacts (classes or modules against original design requirements):

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 7:

ScrumCity was helpful in tracing original functional requirement to their code implementations.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 8:

The bidirectional tracing mechanism as presented by ScrumCity (functional requirements to code artefacts and vice versa) is beneficial for software maintenance and development activities:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 9:

The concept of bidirectional tracing mechanism (functional requirements to code artefacts and vice versa) is beneficial for software maintenance and development activities:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 10:

ScrumCity is helpful for design reasoning and exploratory activities of a system:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 11:

ScrumCity can be helpful for newcomers to gain quick understanding of a system:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 12:

ScrumCity can help developers in their feature location activities:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

Question 13:

ScrumCity can help developers to understand the distribution of implementation of a particular feature:

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

General Comments:

Kindly provide any general feedback or comments about the concept presented in ScrumCity. In particular, please mention any specific suggestions for improvements:

Appendix VIII: Phase Two Material

VIII.I Interview Protocol

Interview Protocol

Team member

Project title: **Software Visualisation to Support Agile Software Processes**

Project Supervisor: **Stephen MacDonell, Jim Buchan**

Researcher: **Mujtaba Alshakhouri**

This protocol is designed to support the conduct of interviews with participants from local industry who are practicing agile software development. The interview is focused around a demonstration of a tool prototype and aimed at collecting preliminary feedback.

---

1. **Participant Briefing**

   At the beginning of each interview session, each participant will be briefed about the research and reminded of key protocols of how it will be conducted, in accordance to the details found in the information sheet. This will include:

   - Purpose of the research
   - Explaining the process & and expected time of the session
   - Getting their consent forms signed and reminding them that they have the right to stop at any stage
   - Notes will be taken on a structured form
   - Interview will be audio recorded if consent is given
   - Receiving and answering any queries they might have before actual interview starts

2. **Background information**

   - What is your role in your current project?
   - How many years of software development experience do you have?
   - What is your educational background?
   - What are your collective experiences with agile development and practices?

3. **Understanding agile workflow based on participant's industry experience**

   - Typical work routine in agile software development environment. How does this compare to other developers? Testers?
   - Main toolsets, products, and technologies used? How?
   - Work Break down and allocation
   - Typical lifecycle of a user story (or a feature, if applicable)?
   - Maintenance of an existing feature. Change impact analysis? Supporting tools? Ways of doing it?
   - How is information required to accomplish above tasks (maintenance) obtained?
   - What are the main challenges faced in an agile software development environment (with respect to feature implementation/maintenance)?

Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

4. **Prototype Demo**
    - Is there any specific way you see the tool being relevant to your work? Any challenges it addresses?
    - Would you be interested to try out the fully developed tool? Can you recognise other colleagues/roles who you think might benefit from it?
    - What functionalities of those demonstrated would you rather get rid of? What others would you rather have?
    - Impression of user story tagging. Do you like the idea? What is your reaction if required to do it regularly? Would your reaction change if you see immediate value? How would you prefer to do it?

5. **General Recommendations**
    - What general recommendations or advice would you like to offer to improve the effectiveness of the tool?
    - Anything else to add?

Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

VIII.II Interview Guide (informal)

This is a guide to the structure of the interview showing the sequence of questions to be asked, cues to opening keywords, and points to be discussed.

Opening [5 mins]
1. Thank participant for coming and willing to participate 2. Introduce yourself, supervisors. Jim to be taking notes 3. Explain what will happen in this session, two parts, 1 hour. [*a. understanding your daily workflow & way of working, b. feedback on muck-up*] 4. Consent form signing, recording permission. In cordial & casual manner. 5. Questions?
Introduce Problem [5 mins]
6. Use the slide with user story diagram 7. Based on talking to developers, it seems like understanding the impact of a code change on features to be challenging problem… 8. Similarly, given an existing feature that needs to be enhanced, it seems also not so easy to identify which piece of code to modify, or what parts of code might be impacted… 9. Let me illustrate better… (diagram is now helpful. Highlight key questions developers normally ask) 10. We are building a tool that tracks which part of code relates to which user story. We think it will allow these and similar questions to be answered easier and quicker. 11. There are two parts that make this tool possible. 12. Now I explained this to give you the right context of the problem we are working on… but before we continue I'd love now to know a bit about you, about your daily workflow… and how you go about completing some activities.
Demography [4 mins]
13. Let interview introduce themselves and give a bit of background 14. If not already covered, ask: years of experience practicing agile, experience in general, responsibilities at work, etc.
Practiced Workflow [20 mins]
15. I'd love to know briefly how work is broken down among your team members and how task allocation is done [*ensure # of team members, tool used are covered*] 16. Now, what about a new user story… could you please describe the typical workflow and processes involved from picking up this user story card, until it is finally completed. [*ensure they cover: tools involved, what happens to the card at the end*] 17. What about doing some changes to an existing feature? How would such task be normally approached? [*ensure they cover: how they identify parts of code, how they access and obtain required information, any tools involved in the process*]

18. OK, now let's say there's a new bug discovered. Could you please tell us how handling this task would be any different?
19. If not already covered…, ask: what happens to user story cards after they are completed. [*ensure they cover tools, and cover if there were situation where they found themselves wanting to go back to these cards*]
20. What are some key challenges you face in the way you currently work with respect to implementing new features or modifying existing ones?

Feedback on Mock-up [26 mins**]**

21. Thank you very much, now let's go back to the tool or rather the mock-up… I'd love to show you some ideas of features and scenarios so you can hopefully give us feedback, and advice on how to improve it
22. So we were saying that the tool has two parts that make it possible…
23. First, there's the problem of how to make the link between user stories and their related piece of code. We wanted a mechanism that has low footprint but also very accurate and reliable.
24. Then, there's the problem of how to present these links in a nice and natural way that makes it easy to answer the kind of questions and scenarios we talked about, and support a convenience decision making process.

    [*use a diagram when explaining the above*]
25. For the first part, let me show you the mechanism we've been thinking of… [*now use the tool to demonstrate the tagging process]*

    [*Do not give any cues to any improvements just as yet, record original reaction to mechanism demonstrated. Then cue interviewee in sharing his/her thoughts about it. Now, ask for ideas to improve. If not already brought up, seek their opinion of received suggestions: tagging as part of commit operations, tags automatically done by tool then user manually confirms or overrides, verify correctness as part of pull requests*]
26. Now let me show you how we present these links to the user…
27. Show some user stories on Trello
28. Now show the 3D code city visualisation… describe how a user story is now represented/revealed on the scene
29. Record original reaction, then seek interviewee feedback and opinion
30. If not brought up, explain how you think this representation is helpful from various perspectives: shared mental map, cognitive benefits, as information radiator, visual shared context for discussions, support various scenarios, etc.
31. Let me show some scenarios we think this tool could be useful for…
32. Now use the mock up to demonstrate a couple of the scenarios implemented. Maintain passiveness and avoid cues.
33. With each scenario, record original reaction then seek explicit feedback, ideas for improvements, etc.
34. Cover some examples of how some other code attributes could also be integrated into this representation and how they could be useful.
35. Seek feedback and opinion, enhancements, ideas, etc.

Interview Guiding Scenarios

Team member

Project title: **Software Visualisation to Support Agile Software Processes**

Project Supervisor: **Stephen MacDonell, Jim Buchan**

Researcher: **Mujtaba Alshakhouri**

These are usage scenarios to invoke during participant interview sessions when demonstrating the tool's wireframe. They are intended to help draw practical and real-world context to the potential uses of the prospective tool. They are expected to guide and prime the conversation into how the tool could be used to solve actual problems faced by the participants in their work environment, and to elicit further use case scenarios from the participants. The goal is to collect information directly from practitioners to better inform the development of the tool.

---

1. **Do a tagging scenario but only after running through one of the below functionality scenarios**

2. **Change Impact analysis 1 (Developer)**

   Context: Developer needs to modify a known code artefact. They would like to identify any impact on other code artefacts as well as possibly related system features. **Action:** They use the tool's "Browse Code Entities" feature to lookup the target code artefact by name or through the IDE's package explorer. **Result**: the tool generates a 3D scene that visually highlights all impacted code artefacts. The tool displays a list of related agile artefacts (user stories, issues, bug fix, etc.). User can interact with both sets of information in bi-directional way and can navigate to original agile artefact on its source (e.g., Trello Board).

3. **Change Impact analysis 2 (Exposing Dependency & Coupling /QA Engineer)**

   Context: A prospective release is planned that requires changes to a number of existing system features. A quality engineer is tasked with analysing the robustness of existing codebase, assessing the extent of potential impact, and identifying key code components and system functionalities that are going to be most affected. **Action**: QA personnel decides to use the tool's 'Change Impact Analysis' functionality to inspect and study existing dependencies and couplings in the codebase. Using the goals and objectives set for the planned release, QA person identifies and come up with a list of related system features (or concepts) in the form of keywords and named functionalities. They then use these keywords to perform lookup operation under the tool's 'Change Impact Analysis' section. **Result**: With each lookup operation, the tool displays a list of matching user stories and issues in the dropdown search results menu. QA person selects an item of interest and the tools then displays the visualised city landscape with a couple of buildings now distinctly highlighted and clearly designated by a floating textual label showing the named user story or concept that was selected. Performing other lookup operations, the QA person identifies and adds more designated buildings with labels floating overhead. She ends up identifying 9 buildings

Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

corresponding to 9 classes and interfaces that she believe will be most impacted by the planned release. She prints off a report and produce a snapshot of the visualised 3D codebase to further discuss with the developers.

4. Feature Location (or Concept Location)

Context: Developer is tasked with enhancing and modifying a certain system feature (or user story, or issue, etc). Developer has no information regarding the exact code artefacts that are implementing this feature. **Action**: Developer decides to use the tool's "Browse Original Design Concepts" functionality to try to locate related code artefacts. They use keywords of the system feature in question to lookup the original agile artefacts. **Result**: Developer finds two related agile artefacts. Selecting each one transports the developer to a region of the 3D city visualisation where related code artefacts are visibly highlighted. Developer can conveniently interact back and forth with each of the identified code artefacts, inspect their source code in place, and find out about any other code artefacts that could possibly be impacted too. Out of the related code artefacts identified, developer has now recognised two entities that they need modify to implement the enhancement.

5. Refactoring

Context: Developer is tasked with refactoring several code artefacts. Many of those are developed by other team members. Developer would like to understand the original purpose behind each of those code artefacts before initiating any refactoring. Developer finds only minimal comments on source code. **Action**: They decide to use the "Navigate Concepts" feature of the tool to identify and locate the original agile artefacts related to a given code artefact. Using the name of the code artefact, they perform a quick search. **Result:** Agile artefacts (user stories, issues, etc.) related to the code artefact in question are displayed to the user. User can select each and read its description or navigate to its original card on the agile dashboard. User has now better understanding of the original developer's intention of the code artefact and thus is better equipped to perform the refactoring.

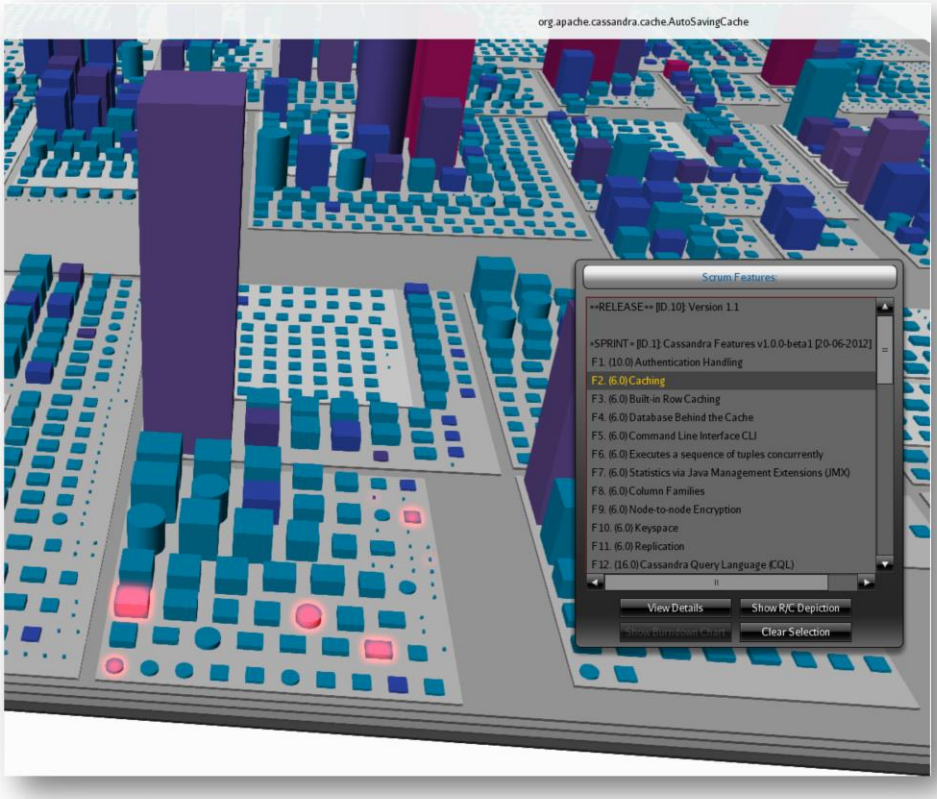
6. Supporting Team Communication

Context. Team are having a technical meeting to discuss quality and code management issues. Their codebase is very large consisting of hundreds of modules. They are interested in possible refinement to the architecture of the modules to decrease coupling and increase cohesion. **Action**: The team decides to make use of the tool to generate an instant 3D visualisation of their entire codebase. The visualisation displays a city landscape depicting the codebase structure, where modules are now visible and recognisable as districts in distinct colours. **Result**: The team are now equipped with an up to date and live visualisation of their codebase structure that conveniently work as a shared groundwork and mental view of their code. The team can physically point to buildings, and regions of the city to better set context to their conversations and ideas. All team members have visual feedback on what is being highlighted or suggested. Buildings colours and sizes give the team more information about the code entity complexity and size. They have instant feedback on what each entity implements. Selecting an entity quickly highlights other entities affected by it. Selecting a system feature quickly reveals its implementation locality. The team concludes their meeting by selecting a few modules of the city landscape that are of special interest to them. They export to 3d printer file and print a physical model to place in their workspace for later reference.

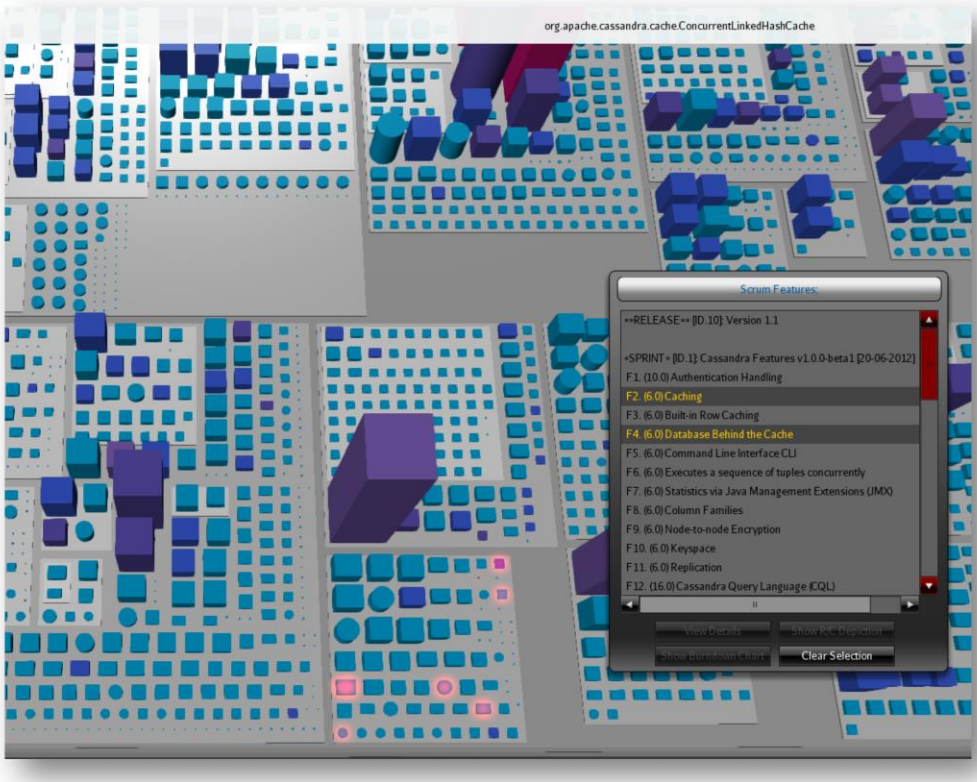
Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

7. Supporting Sprint Reviews?

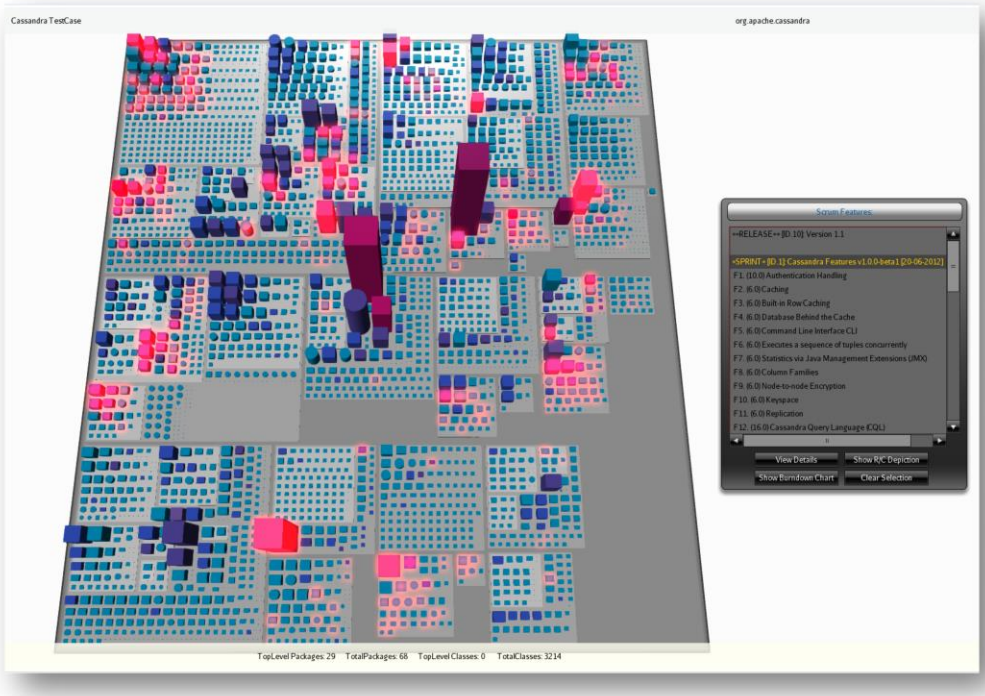
Context. Team is holding a sprint review session where client, product owner, and other stakeholders are expecting to be updated with development progress and a demo of latest increment. Developer wants a convenient way to showcase their latest work and addition. **Action**. The team just recall that the tool has a functionality named "Sprint Review Perspective" so they decide to test it and try it out. They generate a fresh visualisation of their codebase and then invoke the Sprint Review perspective. **Result**: The team is presented with a list of all sprints undertaken so far. They select their latest sprint and are now presented with a large city landscape of building representing their code entities. Scattered across the landscape are distinctly coloured buildings and regions indicating they are recent contributions by this sprint. They can further easily distinguish buildings that are completely new, from those that are partially updated by this sprint. Their quality engineer and product manager are now able to point to exact locations where the recently developed system features are distributed across the system. Their team's contribution of the last sprint is visibly identifiable.



Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

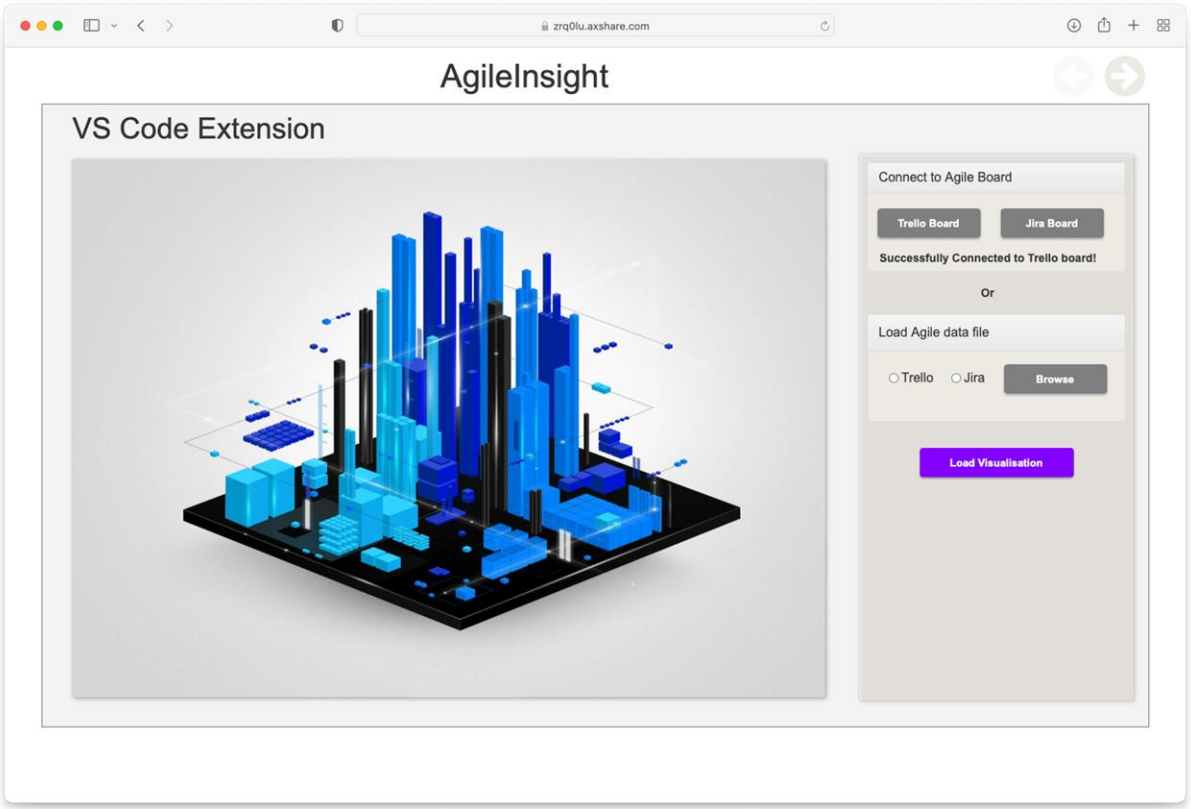


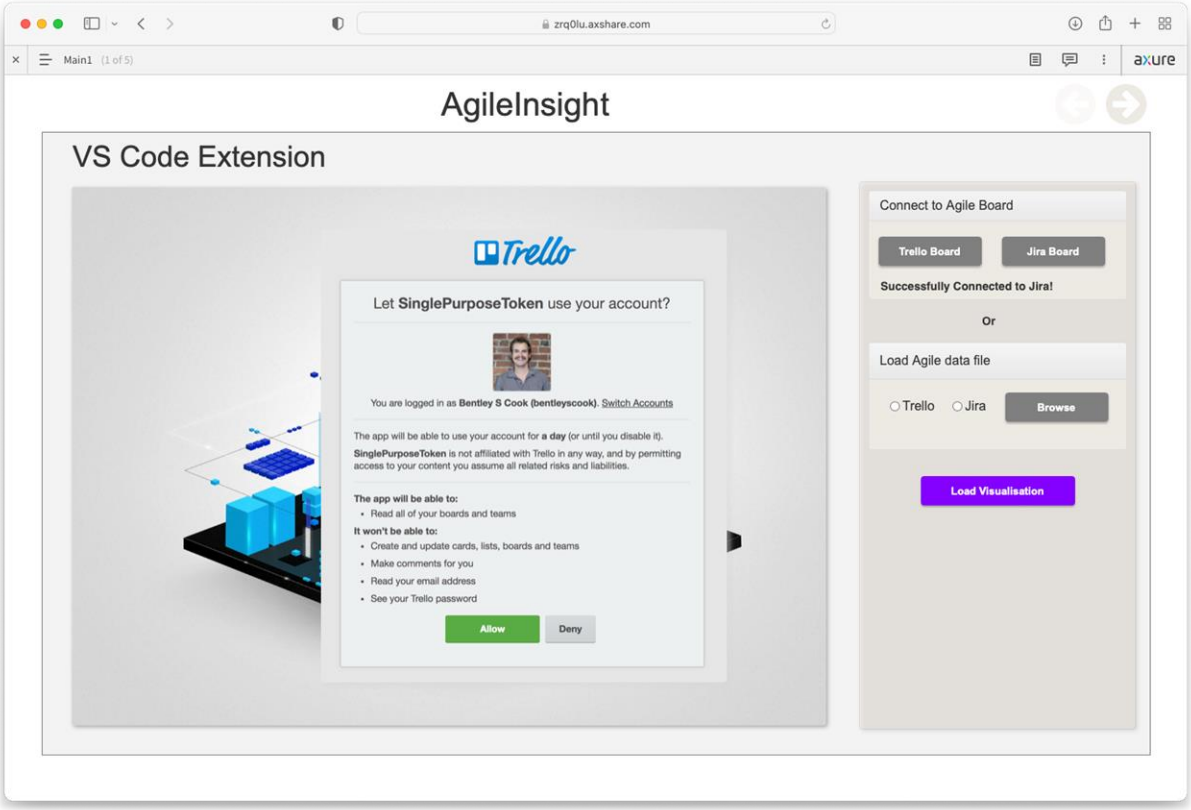
Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

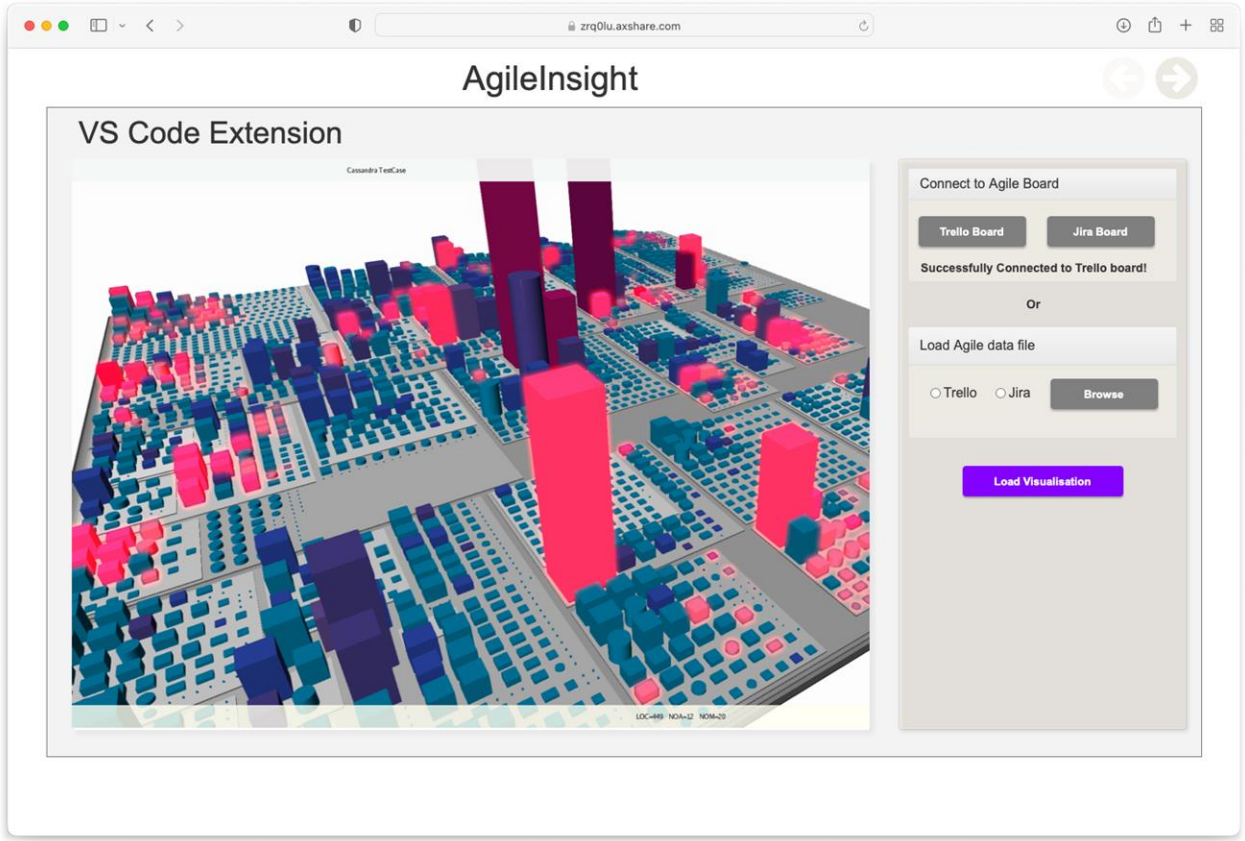


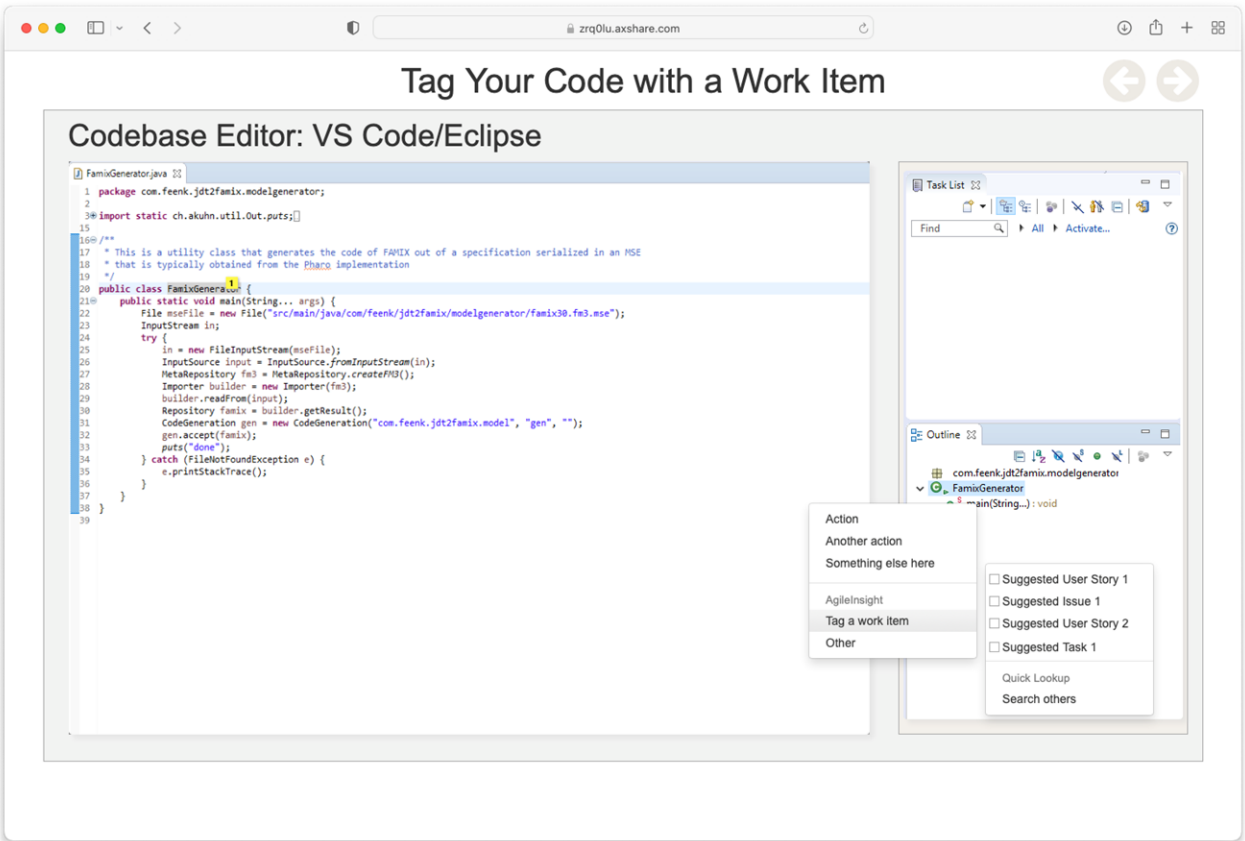
Approved by the Auckland University of Technology Ethics Committee on 21 May 2019 AUTEC Reference number 19/131

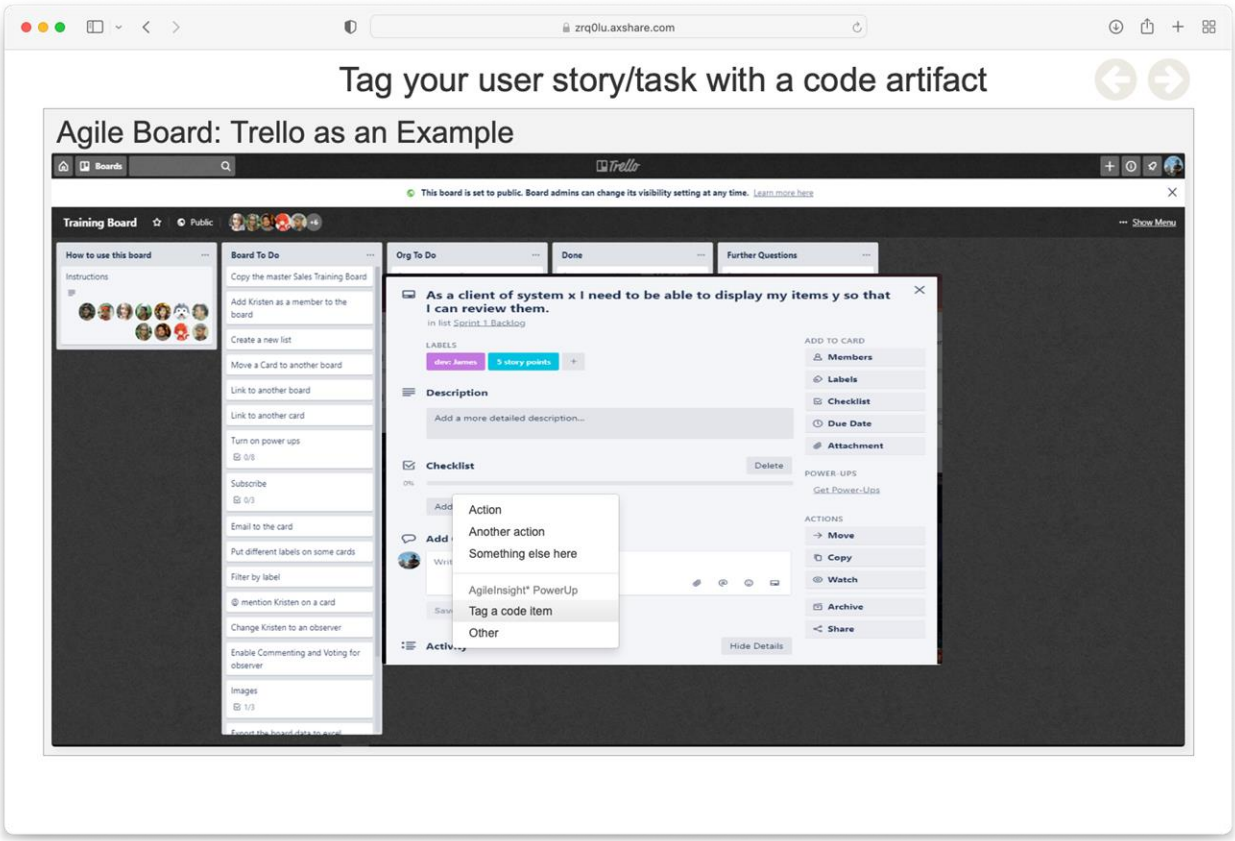
VIII.IV Wireframe Prototype

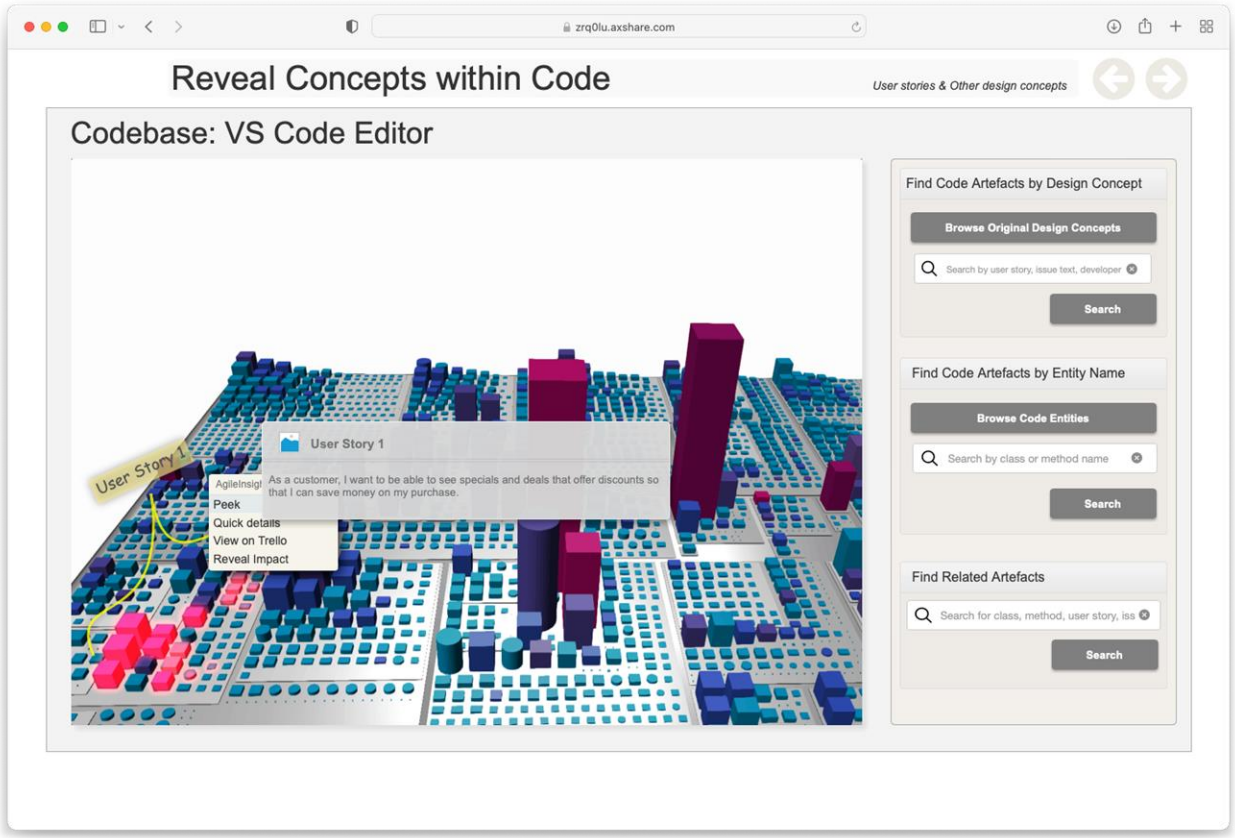


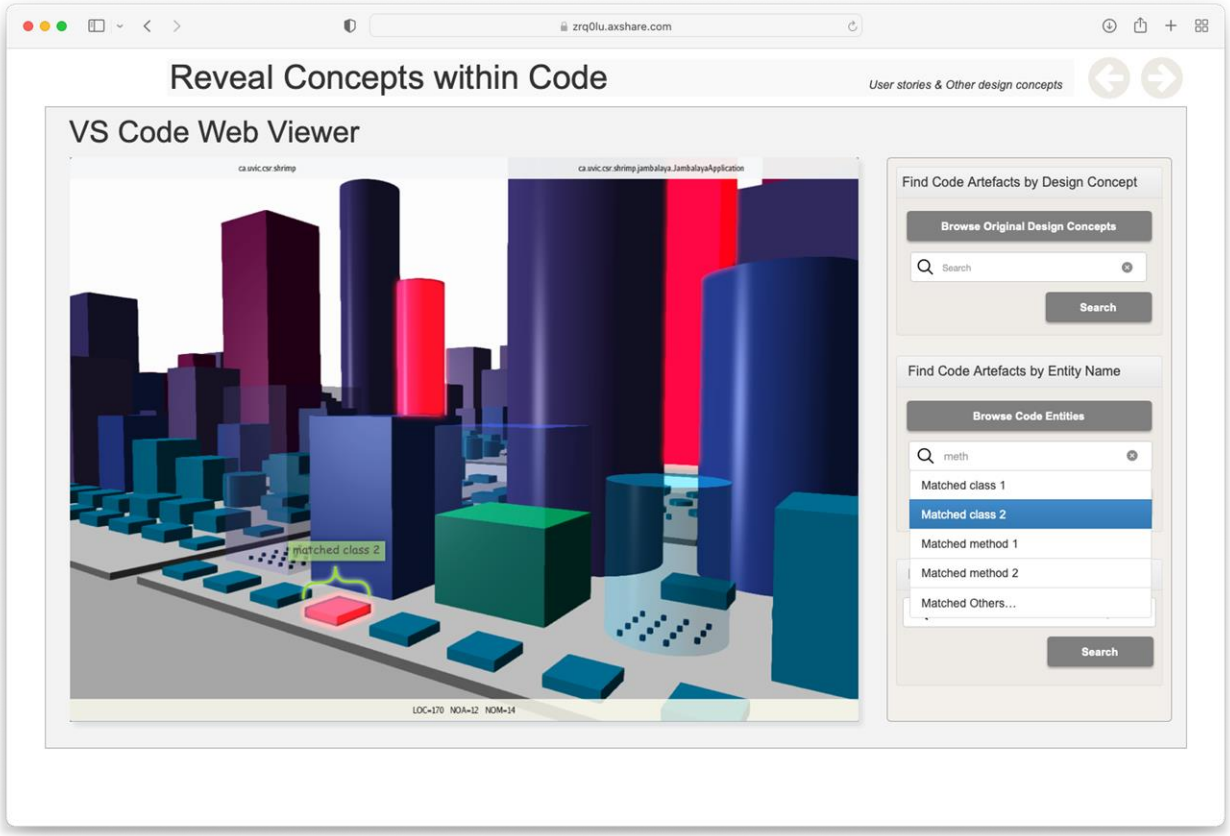


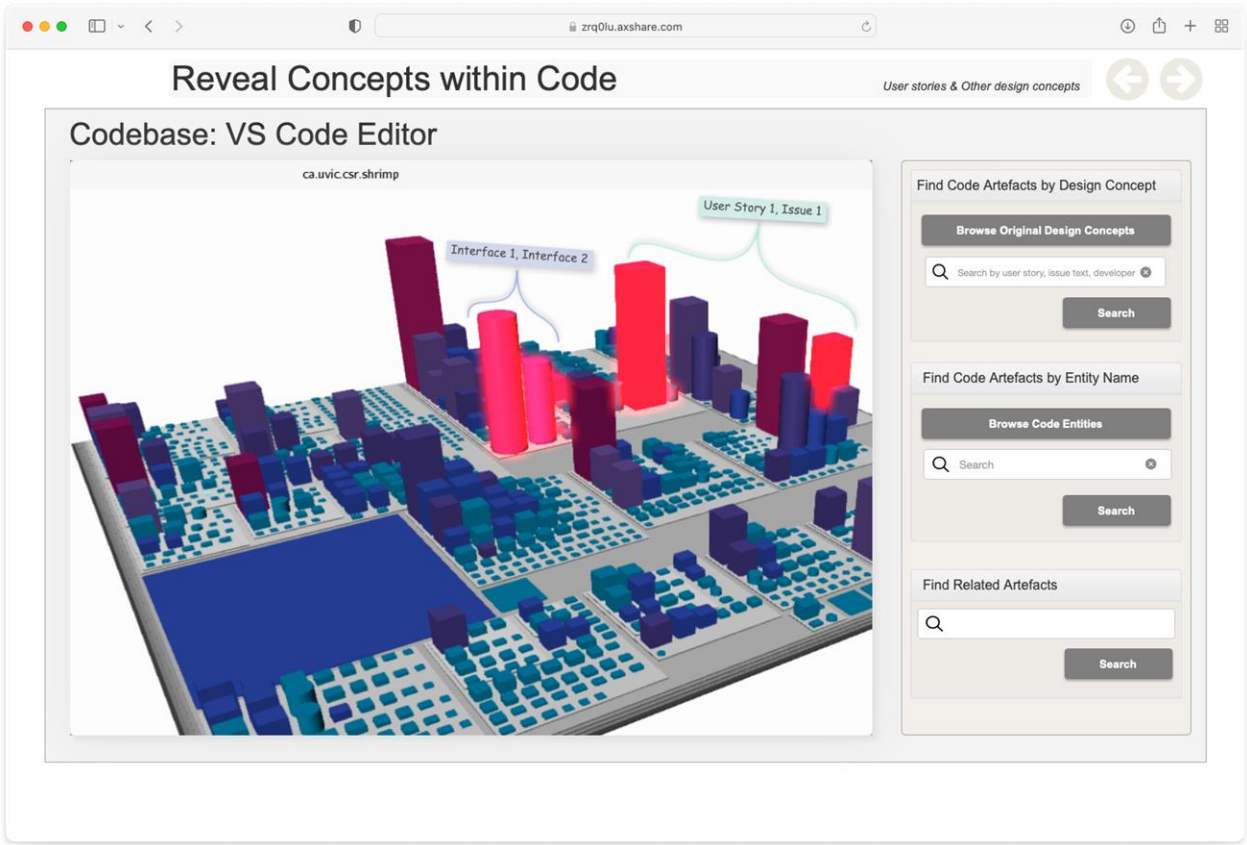

zrq0lu.axshare.com
AgileInsight
VS Code Extension
Cassandra TestCase
Connect to Agile Board
Trello Board
Jira Board
Successfully Connected to Trello board!
Or
Load Agile data file
Trello
Jira
Browse
Load Visualisation

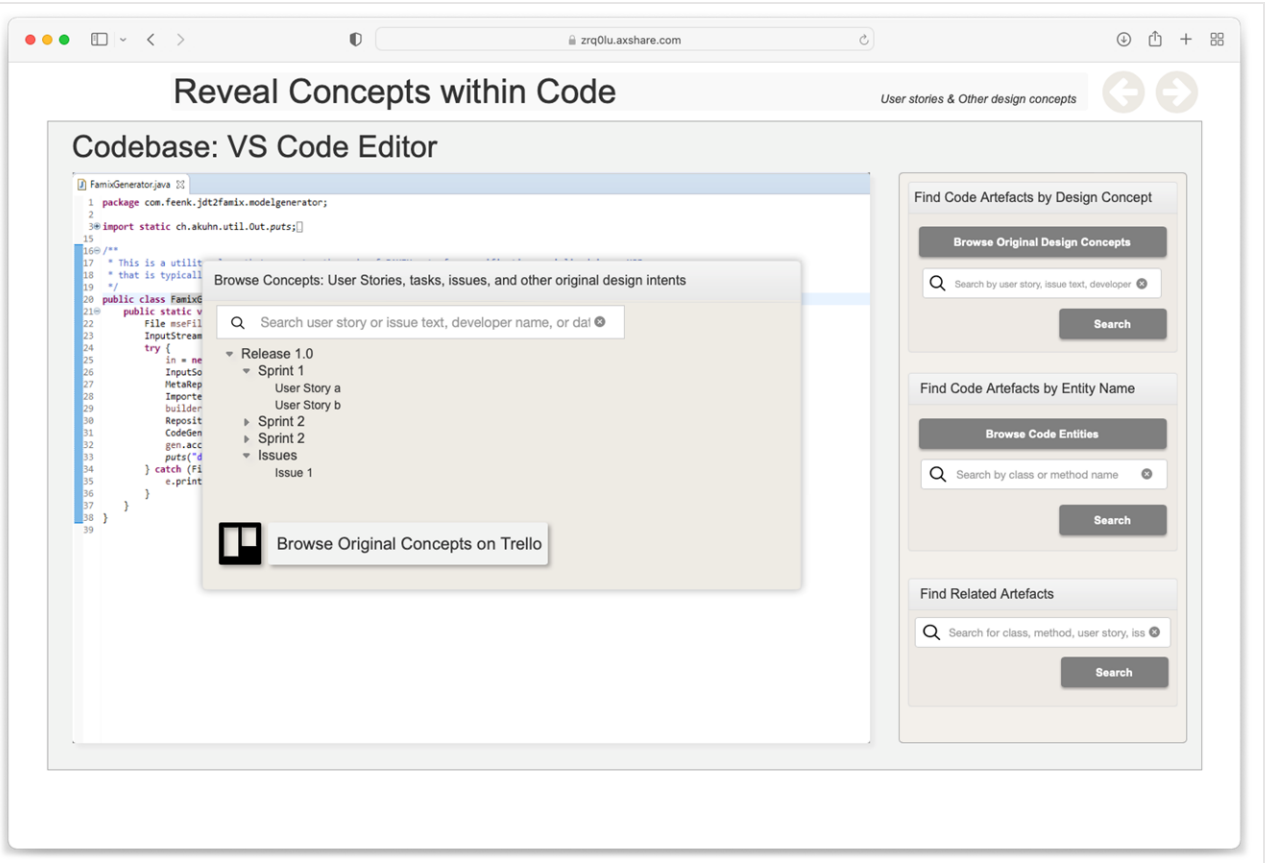

zrq0lu.axshare.com
Tag Your Code with a Work Item
Codebase Editor: VS Code/Eclipse
FamixGenerator.java
Task List
Outline
Action
Another action
Something else here
AgileInsight
Tag a work item
Other
Suggested User Story 1
Suggested Issue 1
Suggested User Story 2
Suggested Task 1
Quick Lookup
Search others

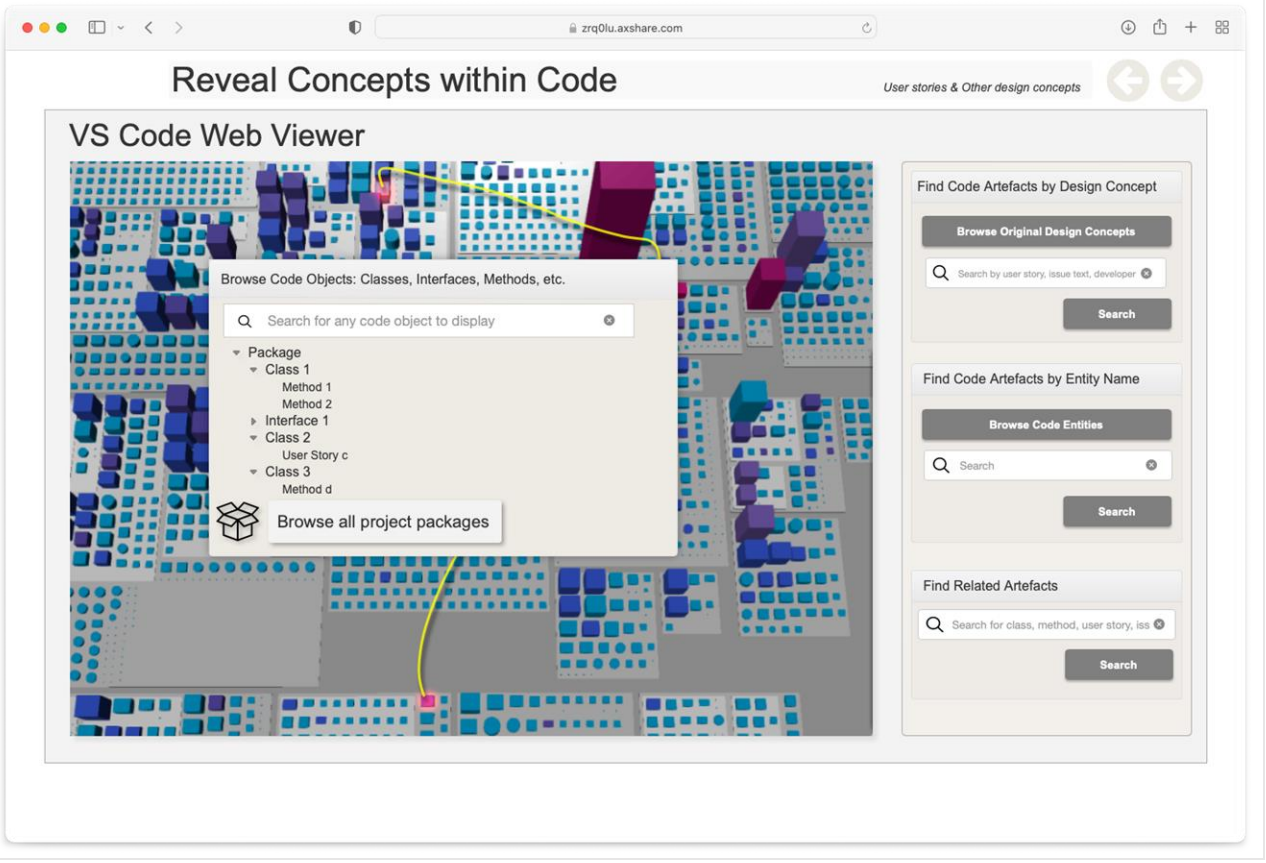

zrq0lu.axshare.com
Tag your user story/task with a code artifact
Agile Board: Trello as an Example
Trello
Training Board
As a client of system x I need to be able to display my items y so that I can review them.
Action
Another action
Something else here
AgileInsight* PowerUp
Tag a code item
Other

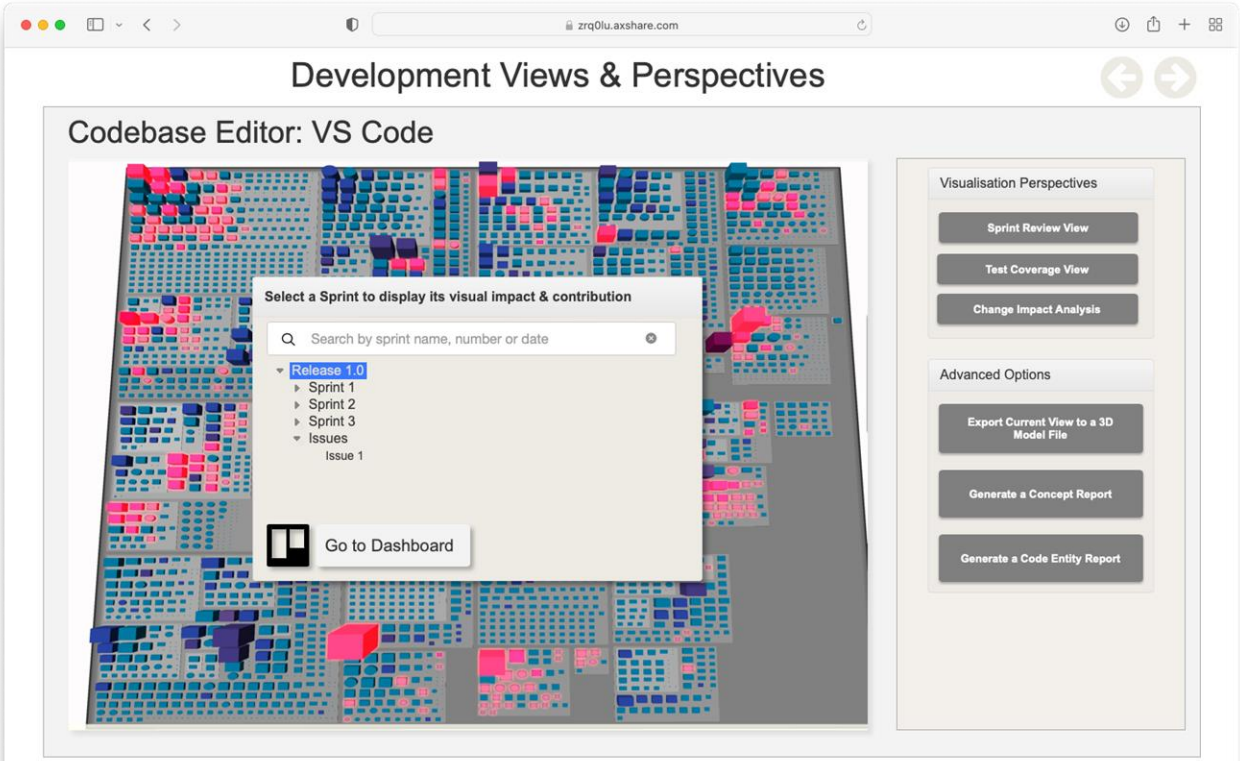

zrq0lu.axshare.com
Reveal Concepts within Code
User stories & Other design concepts
Codebase: VS Code Editor
User Story 1
As a customer, I want to be able to see specials and deals that offer discounts so that I can save money on my purchase.
Peek
Quick details
View on Trello
Reveal Impact
Find Code Artefacts by Design Concept
Browse Original Design Concepts
Search by user story, issue text, developer
Search
Find Code Artefacts by Entity Name
Browse Code Entities
Search by class or method name
Search
Find Related Artefacts
Search for class, method, user story, iss
Search

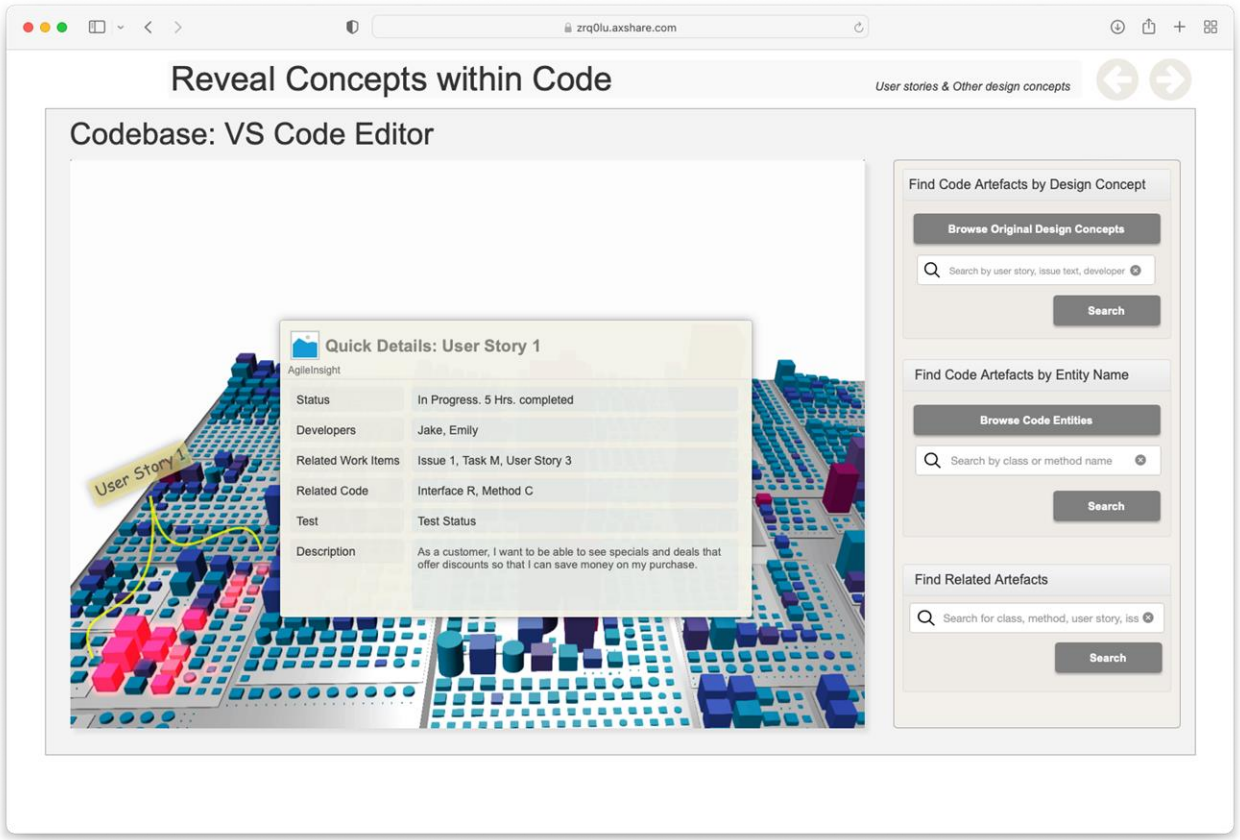

zrq0lu.axshare.com
Reveal Concepts within Code
User stories & Other design concepts
VS Code Web Viewer
ca.uvic.csr.shrimp
ca.uvic.csr.shrimp.jambalaya.JambalayaApplication
matched class 2
LOC=170 NOA=12 NOM=14
Find Code Artefacts by Design Concept
Browse Original Design Concepts
Search
Find Code Artefacts by Entity Name
Browse Code Entities
meth
Matched class 1
Matched class 2
Matched method 1
Matched method 2
Matched Others…
Search

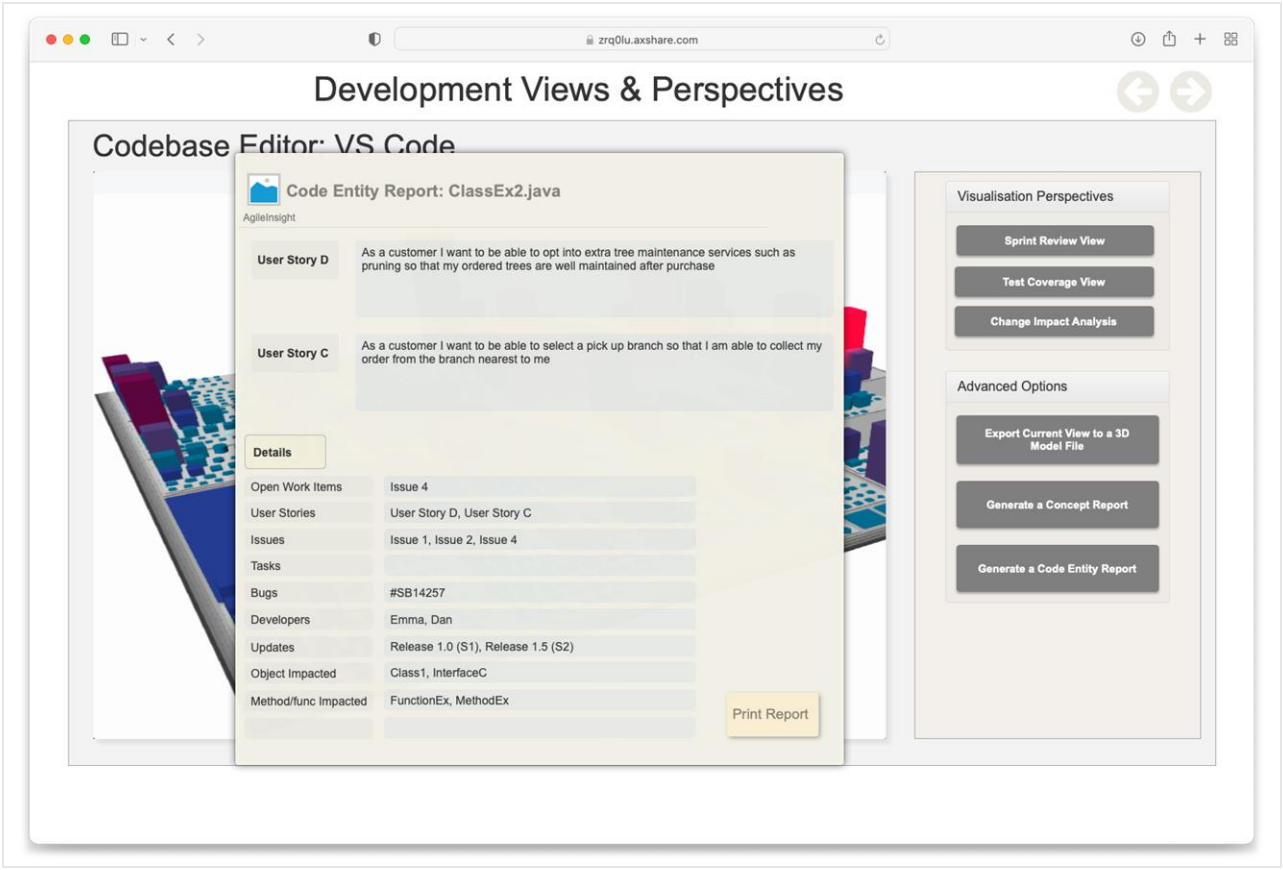

zrq0lu.axshare.com
Reveal Concepts within Code
User stories & Other design concepts
Codebase: VS Code Editor
ca.uvic.csr.shrimp
User Story 1, Issue 1
Interface 1, Interface 2
Find Code Artefacts by Design Concept
Browse Original Design Concepts
Search by user story, issue text, developer
Search
Find Code Artefacts by Entity Name
Browse Code Entities
Search
Search
Find Related Artefacts
Search


zrq0lu.axshare.com
Reveal Concepts within Code
User stories & Other design concepts
Codebase: VS Code Editor
FamixGenerator.java
package com.feenk.jdt2famix.modelgenerator;
import static ch.akuhn.util.Out.puts;
Browse Concepts: User Stories, tasks, issues, and other original design intents
Search user story or issue text, developer name, or dat
Release 1.0
Sprint 1
User Story a
User Story b
Sprint 2
Sprint 2
Issues
Issue 1
Browse Original Concepts on Trello
Find Code Artefacts by Design Concept
Browse Original Design Concepts
Search by user story, issue text, developer
Search
Find Code Artefacts by Entity Name
Browse Code Entities
Search by class or method name
Search
Find Related Artefacts
Search for class, method, user story, iss
Search


zrq0lu.axshare.com
Reveal Concepts within Code
User stories & Other design concepts
VS Code Web Viewer
Browse Code Objects: Classes, Interfaces, Methods, etc.
Search for any code object to display
Package
Class 1
Method 1
Method 2
Interface 1
Class 2
User Story c
Class 3
Method d
Browse all project packages
Find Code Artefacts by Design Concept
Browse Original Design Concepts
Search by user story, issue text, developer
Search
Find Code Artefacts by Entity Name
Browse Code Entities
Search
Search
Find Related Artefacts
Search for class, method, user story, iss
Search


zrq0lu.axshare.com
Development Views & Perspectives
Codebase Editor: VS Code
Select a Sprint to display its visual impact & contribution
Search by sprint name, number or date
Release 1.0
Sprint 1
Sprint 2
Sprint 3
Issues
Issue 1
Go to Dashboard
Visualisation Perspectives
Sprint Review View
Test Coverage View
Change Impact Analysis
Advanced Options
Export Current View to a 3D Model File
Generate a Concept Report
Generate a Code Entity Report


zrq0lu.axshare.com
Reveal Concepts within Code
User stories & Other design concepts
Codebase: VS Code Editor
User Story 1
Quick Details: User Story 1
AgileInsight
Status
In Progress. 5 Hrs. completed
Developers
Jake, Emily
Related Work Items
Issue 1, Task M, User Story 3
Related Code
Interface R, Method C
Test
Test Status
Description
As a customer, I want to be able to see specials and deals that offer discounts so that I can save money on my purchase.
Find Code Artefacts by Design Concept
Browse Original Design Concepts
Search by user story, issue text, developer
Search
Find Code Artefacts by Entity Name
Browse Code Entities
Search by class or method name
Search
Find Related Artefacts
Search for class, method, user story, iss
Search


zrq0lu.axshare.com
Development Views & Perspectives
Codebase Editor: VS Code
Code Entity Report: ClassEx2.java
AgileInsight
User Story D
As a customer I want to be able to opt into extra tree maintenance services such as pruning so that my ordered trees are well maintained after purchase
User Story C
As a customer I want to be able to select a pick up branch so that I am able to collect my order from the branch nearest to me
Details
Open Work Items
Issue 4
User Stories
User Story D, User Story C
Issues
Issue 1, Issue 2, Issue 4
Tasks
Bugs
#SB14257
Developers
Emma, Dan
Updates
Release 1.0 (S1), Release 1.5 (S2)
Object Impacted
Class1, InterfaceC
Method/func Impacted
FunctionEx, MethodEx
Print Report
Visualisation Perspectives
Sprint Review View
Test Coverage View
Change Impact Analysis
Advanced Options
Export Current View to a 3D Model File
Generate a Concept Report
Generate a Code Entity Report

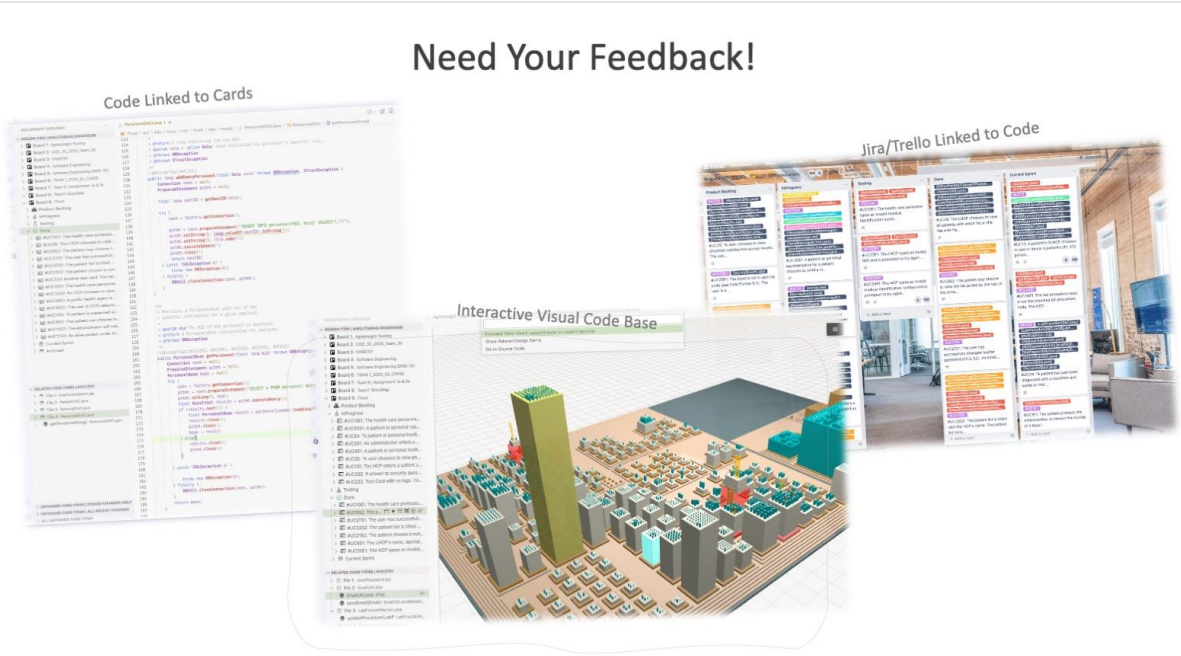
Appendix IX: Phase Three Material

IX.I Evaluation Guideline (Informal)

Greeting ~2 mins
-Thanks for being willing to participate -We have made a tool as part of my doctoral research -Need feedback from experienced developers like you
Overview of this Session ~3 mins
- use overall slide: - Tool + actual product (so we can demonstrate) - plan (3 demos + summary questions) - time
Why AgileInsight ~3 mins
- before I get to show you more, let me explain what the tool is about in the first place - use one slide to explain: problem, opportunity/need, envisioned benefits - a number of mechanisms to present code-to-requirements relationships - answer some common questions: - what a piece of code was originally meant to do - or what parts of source code implement a given feature - uses a visualisation technique to help answer those questions in a natural way that we humans can relate to … that we think results in lower cognitive effort compared to a simple scanning of code base. - demonstrate to you those mechanisms, and the kind of questions we are trying to solve, in order to get your valuable feedback and input
Scenarios Overview + Dataset Overview (don't say dataset) ~2 mins
- go back to overall slide… continue scenario overview (one line each) - use slide for dataset to introduce
Launch Questionnaire ~2 mins
- Remind participants we won't see their answers until all other interviews are done, and answers will never be connected to them individually.
Start Recording
- go back to overall slide… continue scenario overview (one line each)
Scenario 1/2/3 ~30mins [changed to 40]
- brief overview (one statement) - run - summary questions
Verbal discussion + Closing Off ~8 mins
- Thank you

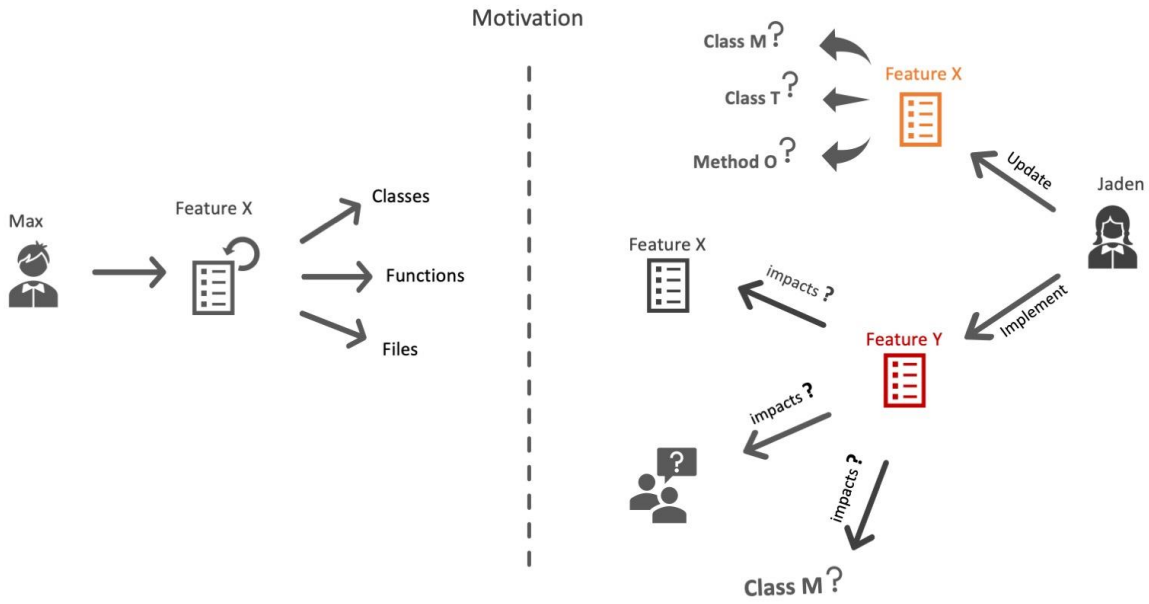
IX.II AgileInsight Evaluation Intro (Slides)

Slide 1

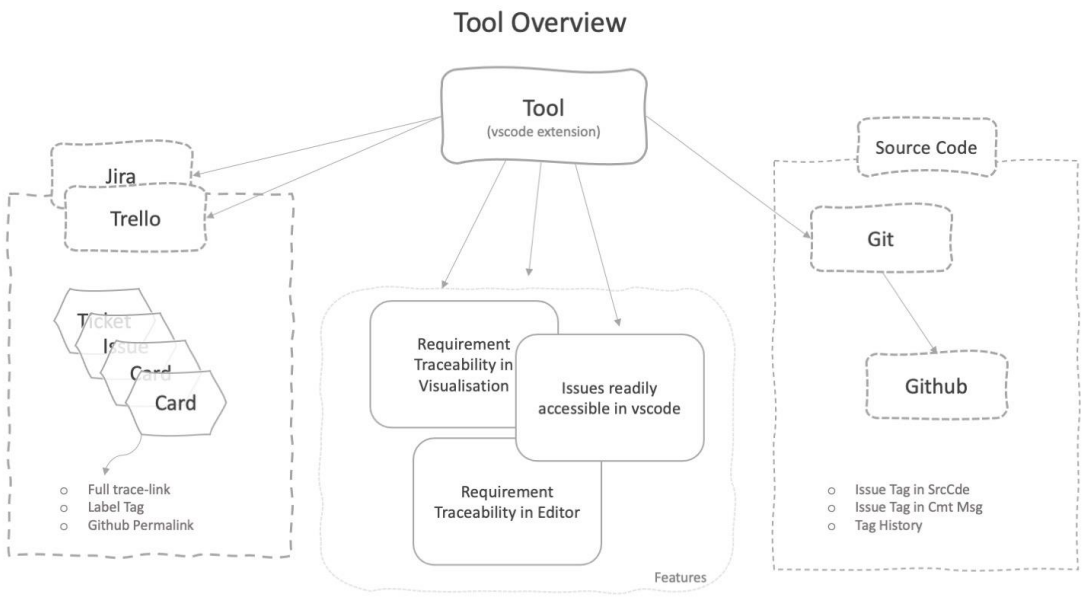


Slide 2

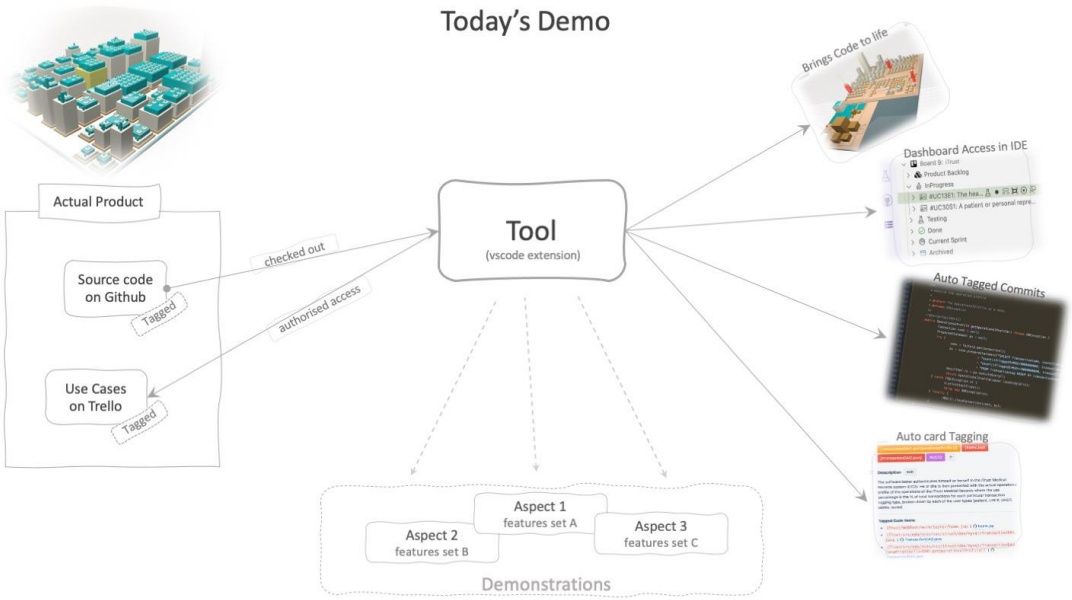
Slide 3



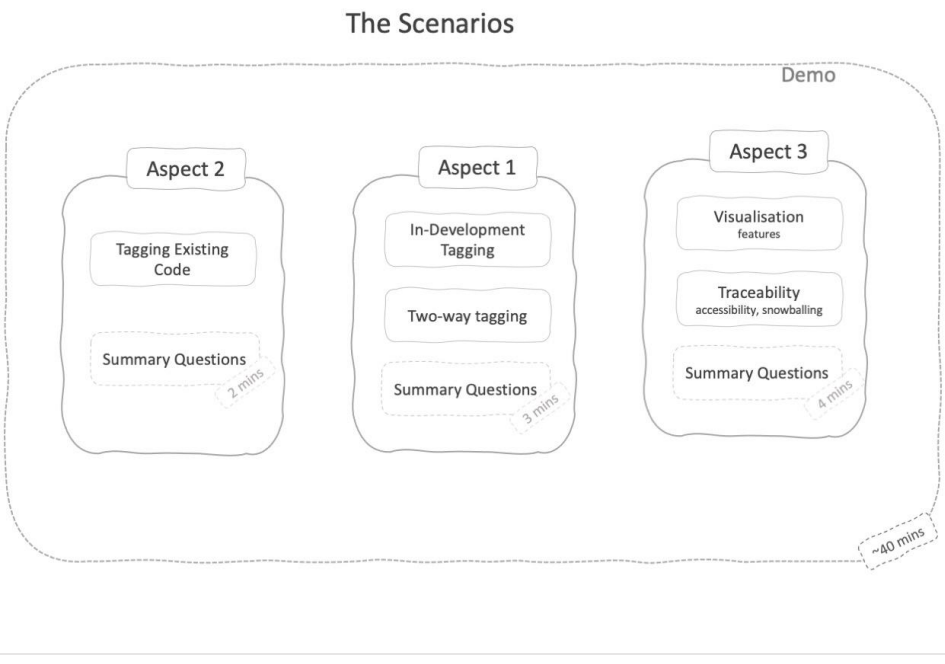
Slide 4



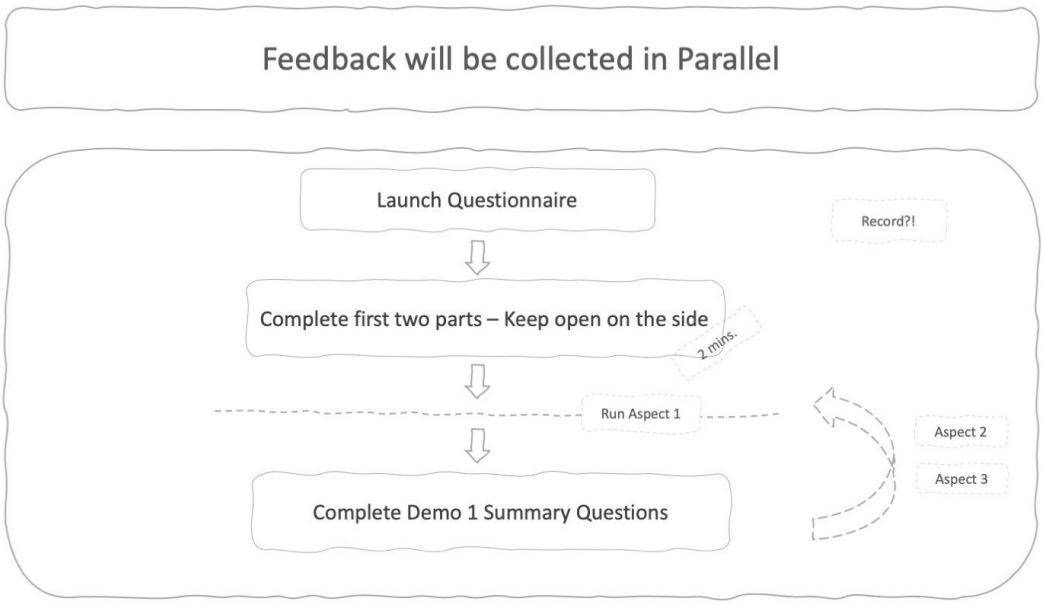
Slide 5



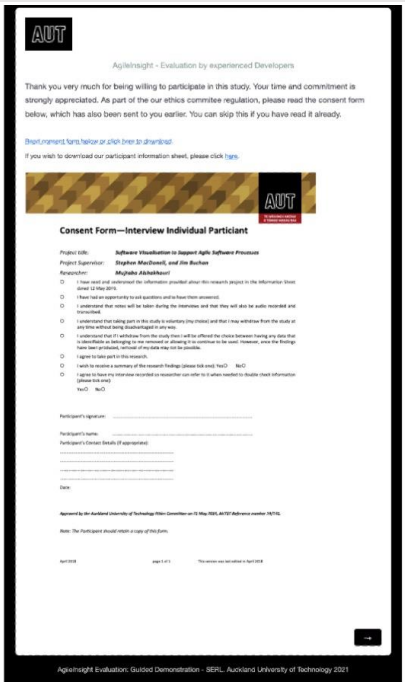
Slide 6

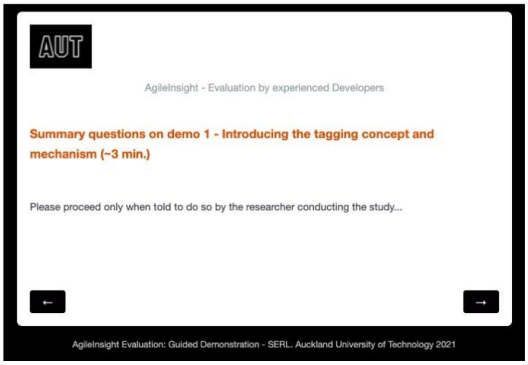


Slide7



Slide 8





Slide 9

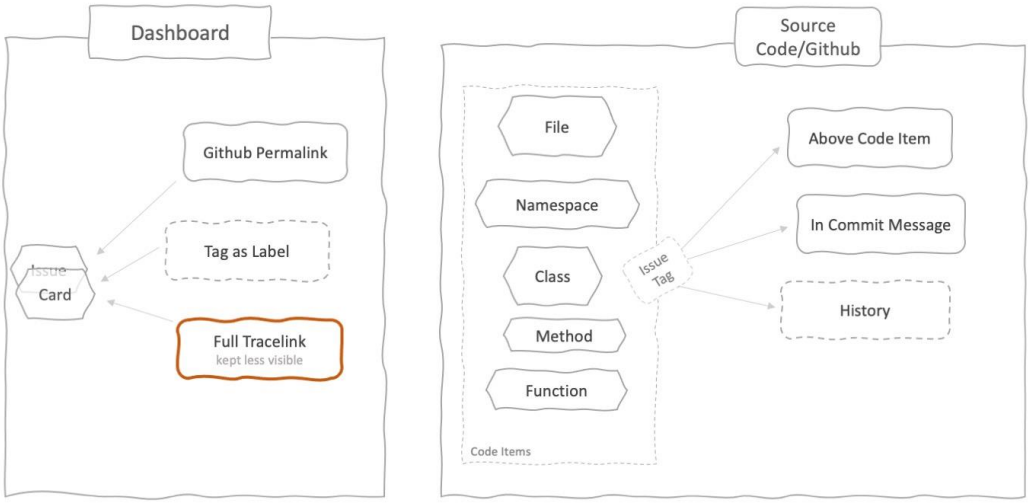
Let’s Begin !

Slide 10

Next two slides will be used as an explanation aid during demonstrations—not part of the intro

Slide 11

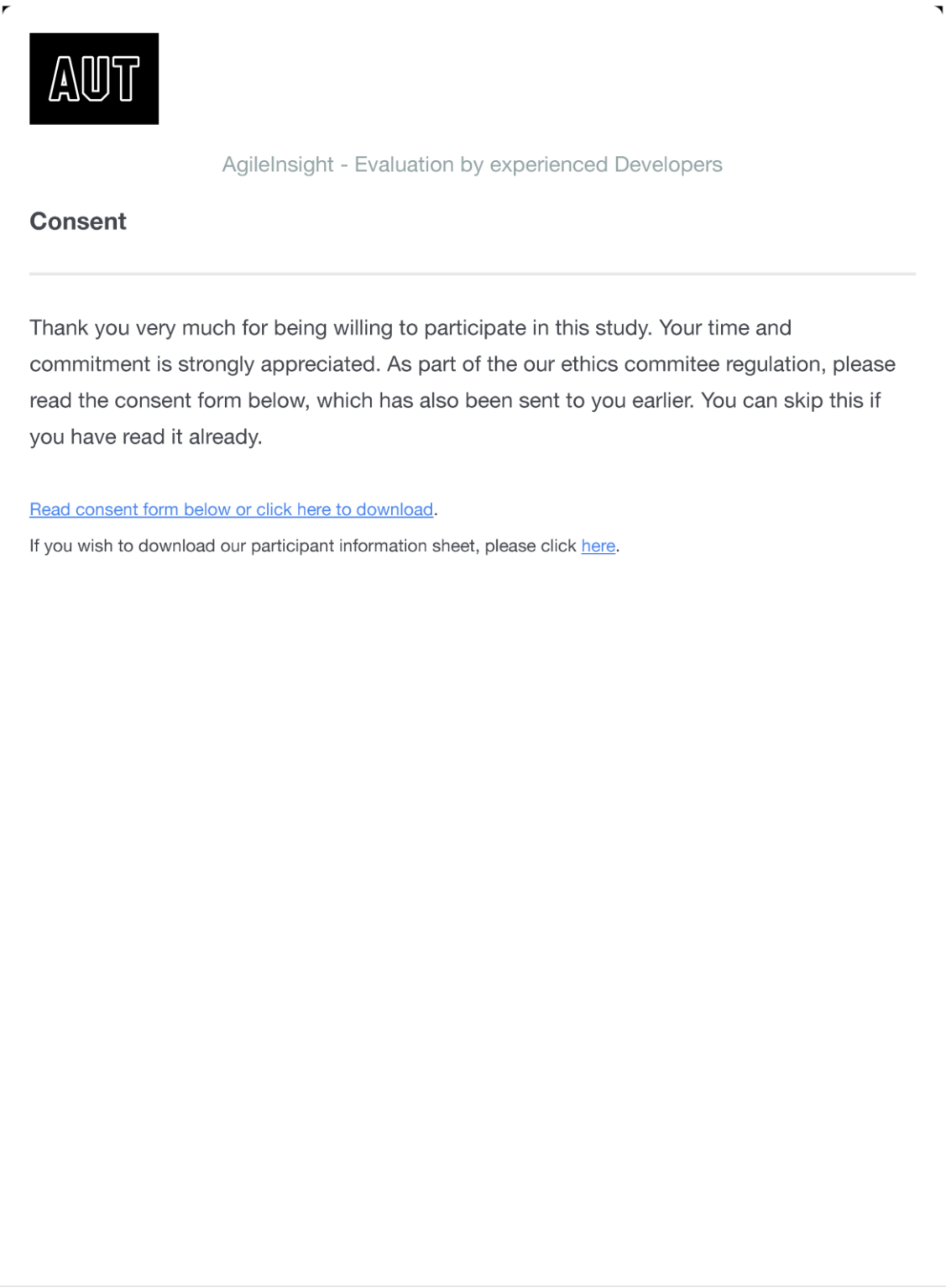
The Tags



Slide 12

Visualisation Keys

IX.III Survey Questions



Consent Form—Interview Individual Particiant

Project title: ***Software Visualisation to Support Agile Software Processes***

Project Supervisor: ***Stephen MacDonell, and Jim Buchan***

Researcher: ***Mujtaba Alshakhouri***

- ○ I have read and understood the information provided about this research project in the Information Sheet dated 12 May 2019.
- ○ I have had an opportunity to ask questions and to have them answered.
- ○ I understand that notes will be taken during the interviews and that they will also be audio recorded and transcribed.
- ○ I understand that taking part in this study is voluntary (my choice) and that I may withdraw from the study at any time without being disadvantaged in any way.
- ○ I understand that if I withdraw from the study then I will be offered the choice between having any data that is identifiable as belonging to me removed or allowing it to continue to be used. However, once the findings have been produced, removal of my data may not be possible.
- ○ I agree to take part in this research.
- ○ I wish to receive a summary of the research findings (please tick one): Yes○ No○
- ○ I agree to have my interview recorded so researcher can refer to it when needed to double check information (please tick one):

  Yes○ No○

Participant's signature: .....

Participant's name: .....

Participant's Contact Details (if appropriate):

.....

.....

.....

.....

Date:

Approved by the Auckland University of Technology Ethics Committee on 21 May 2019, AUTEC Reference number 19/131.

Note: The Participant should retain a copy of this form.

April 2018 page 1 of 1 This version was last edited in April 2018

Do you agree to participate in this study?

Yes

No

This survey consists of four short parts:

Profile Background	approx. 1 min.
Demo 1 Summary Questions	< 3 mins.
Demo 2 Summary Questions	< 2 mins.
Demo 3 (& concluding) Summary Questions	< 4 mins.
Estimated total time	approx. 12 minutes

No writing! -- All questions have been designed with a multi-choice style answers for your convenience.

Ask us anytime! -- If you need any clarification while completing your answers, please feel free to ask the researcher.

Thank you very much for your valuable time!

Participant Profile & Background

Profile & Background (~1 min.)

please proceed whenever ready...

How many years of experience do you have in software industry?

0 to 1 year less than 3 years 3 to 6 years more than 6 years

What is your **current** professional role or title? (chose closest match)

For how many years have you held that role?

0 to 3 years	3 to 6 years	more than 6 years

What other professional roles have you held previously? (please select all that apply)

Software Developer	Product Manager
Software Tester	Scrum Master
Software Engineer	System/Business Analyst
Team Lead/Technical Lead	DevOps Engineer
Junior Developer	UI/UX Designer
Senior Software Engineer/Developer	Project Manager
Software Architect	Other

Which of the following tools do you use at your **current workplace**? (please select all that apply)

Visual studio Code (vscode)	Trello
Microsoft Visual Studio	GitHub
Eclipse	BitBucket
Jira	GitLab

Which of the following tools have you used at **previous jobs**? (please select all that apply)

Visual studio Code (vscode)	Trello
Microsoft Visual Studio	GitHub

Eclipse	BitBucket
Jira	GitLab

Have you previously participated in an earlier phase of this study?

No	Yes

Questions on scenario 1 - Introducing the tagging concept and mechanism

Summary questions on demo 1 - Introducing the tagging concept and mechanism (~3 min.)

Please proceed only when told to do so by the researcher conducting the study...

The previous demo has showcased tagging code items with their original requirements (e.g., issue tickets) in a number of alternative ways during development. How clear do you think the benefits are for this kind of traceability tagging?

Very unclear	Somewhat unclear	Neither clear nor unclear	Somewhat clear	Very clear

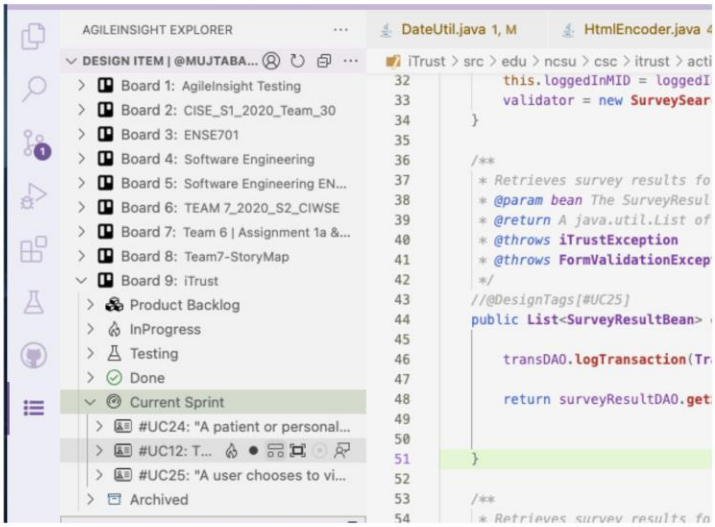
How easy did you find the presented tagging mechanism to be?

Very difficult	Somewhat difficult	Neither easy nor difficult	Somewhat easy	Very easy

How likely is it that you or your team would use a tagging mechanism like this?

	Very unlikely	Somewhat unlikely	Neither likely nor unlikely	Somewhat likely	Very likely
You	○	○	○	○	○
Your team	○	○	○	○	○

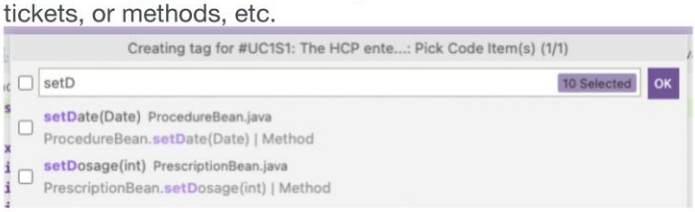
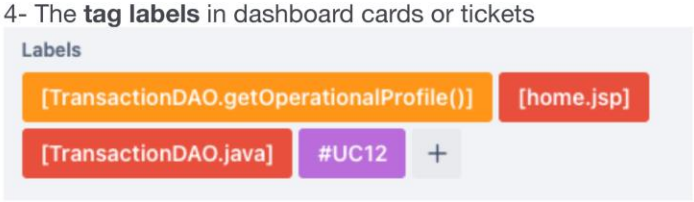
During the demonstrations, you have seen tickets from Trello or Jira board being presented live within vscode and integrated in a number of ways… (e.g.., the Dashboard Tree viewlet).



How useful did you find the idea of having the issue tickets made readily accessible within the development environment (or within the source code)?

Not at all useful	Slightly useful	Moderately useful	Very useful	Extremely useful

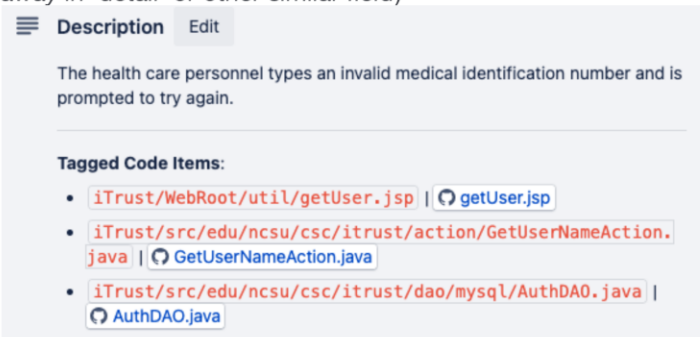
Below are some areas of the demonstrated tagging mechanism. How well do you think each one has worked out? (Part 1)

	Not well at all	Slightly well	Moderately well
1- The **alternative** ways provided to **create a tag** trace-link	○	○	○
2- The Design and Code Items **palettes to select** issue tickets, or methods, etc. 	○	○	○
3- The **tags** placed **in source code** above code blocks 	○	○	○
4- The **tag labels** in dashboard cards or tickets 	○	○	○

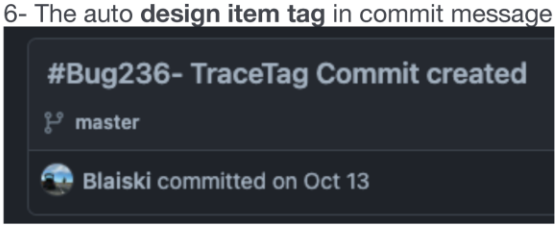
Below are some areas of the demonstrated tagging mechanism. How well do you think has each part worked out? (Part 2)

	Not well at all	Slightly well	Moderately well

5- The **full tracelinks** in dashboard cards or tickets (*tucked away in 'detail' or other similar field*)



6- The auto **design item tag** in commit message



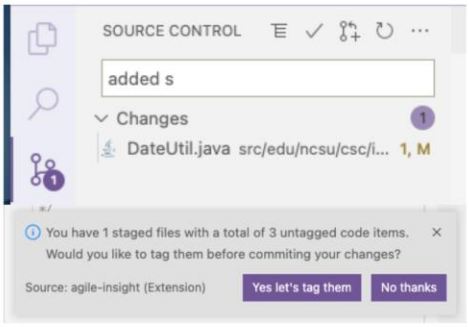
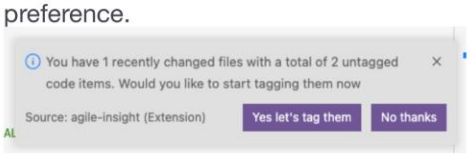
7- The automated **Trace Tag Commit** operation as a whole	○	○		○
8- The **concept itself** of enabling developers to link original requirements to their code implementation	○	○		○

In principle, and aside from our demonstrated tagging method, how much do you support the idea itself of encouraging developers to regularly tag their code with requirements IDs (e.g., issues or ticket numbers)?

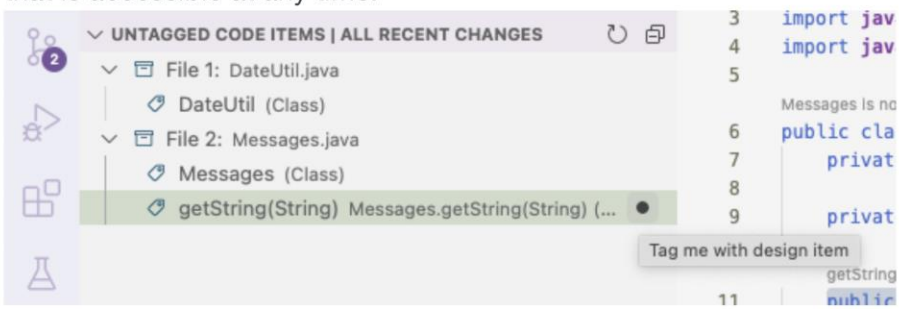
Not at all	A little	Somewhat	Quite a lot	Very much

Below are two tagging reminder approaches that have been presented in the demo. How good do you think each mechanism is in helping users develop the habit of tagging their code?

	Neither

	Very bad	Somewhat bad	good nor bad	Somewhat good	Very good
1- The **staged changes** tagging reminder that is triggered when user is **about to commit** their changes (starts typing in box). 	○	○	○	○	○
2- The **all recent changes** tagging reminder that is triggered every once in a while **as per configured timing** preference. 	○	○	○	○	○

The demo has showcased a mechanism by which recommended tagging for **recently changed** code items was presented to the user in a tree list viewlet that is accessible at any time.



How useful did you find this feature to be in facilitating (or encouraging) a tagging

practice?

Not at all useful	Slightly useful	Moderately useful	Very useful	Extremely useful

Thank you for your responses! :)

Part 1 is now complete, please return back to the researcher to continue with the second guided demonstration.

Questions on scenario 2 - An outlook on Post-Development Tagging

Summary Questions on demo 2 - An outlook on Tagging Existing Code Bases (~2 min.)

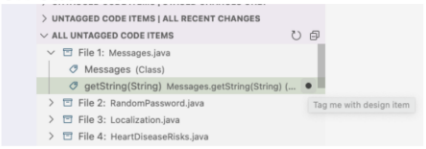
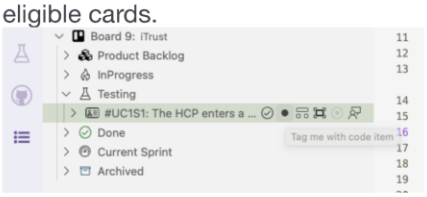
Please proceed only when told to do so by the researcher conducting the study...

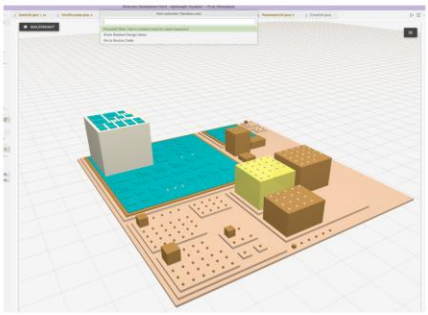
The previous demo has showcased and claimed some potential benefits of having a code base tagged with original requirements…

How likely is it that your team or organisation could come to be interested in tagging an existing code base?

Very unlikely	Somewhat unlikely	Neither likely nor unlikely	Somewhat likely	Very likely

Below are some of the ways presented in the demo to support tagging an existing code base. How well do you think each has worked out?

	Not well at all	Slightly well	Moderately well	Very well	Extremely well
Code Lens hover action shown above eligible code items in an open source code file 	○	○	○	○	○
The **Tree list viewlet** of **all eligible untagged code items** arranged by file, with 'Tag me' action presented on each item 	○	○	○	○	○
The Dashboard **tree list viewlet of all tickets** arranged by category, with 'Tag me' action provided on eligible cards. 	○	○	○	○	○
Using **visualisation view** to locate code items of specific interest or priority to tag					



Overall, how pleased are you with the tool's level of support for tagging an existing code base?

Very displeased	Somewhat displeased	Neither pleased nor displeased	Somewhat pleased	Very pleased

Thank you for your responses! :)

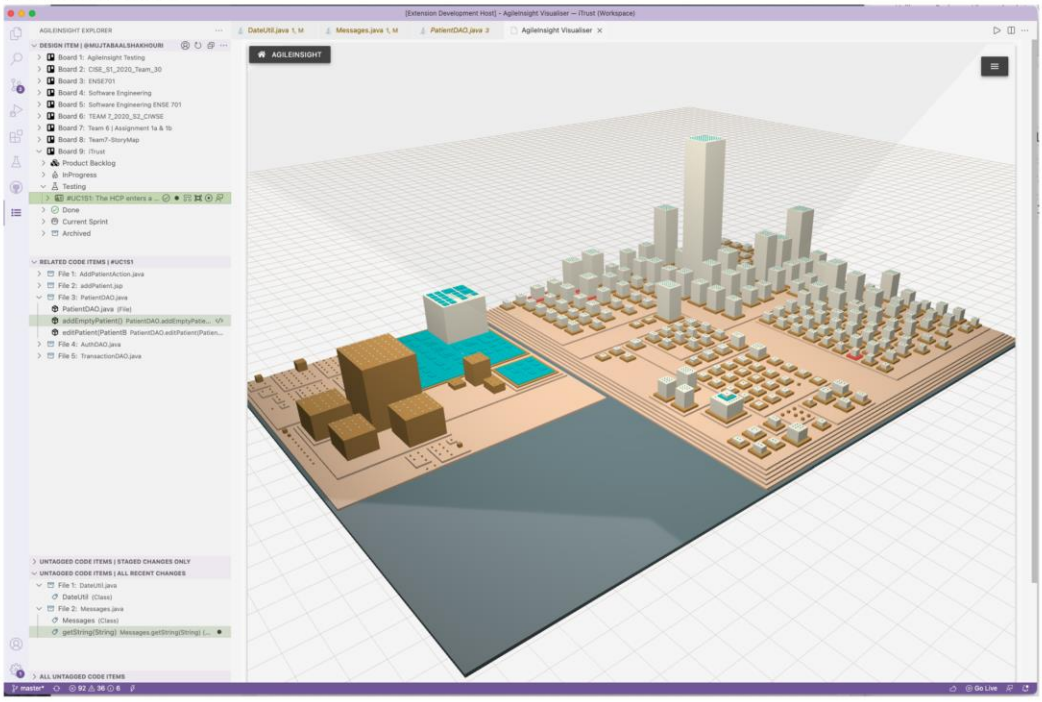
Part 2 is now complete, please return back to the researcher to continue with the last guided demonstration.

Questions on scenario 3 - Code base learning & exploration aided by synchronised visualisation

Summary Questions on demo 3 & Survey Concluding (~4 min.)

Please proceed only when told to do so by the researcher conducting the study...

The first two demos have primarily showcased how the tag tracelinks mechanism works and some potential uses it could bring. In the preceding demo, you have seen a visualisation technique taking advantage of those tag tracelinks by synching them into a visual representation of the code base to enable yet other potential uses.



Overall, how clear to you was the benefit of the visualisation in supporting the different tasks that were demonstrated?

Extremely unclear	Somewhat unclear	Neither clear nor unclear	Somewhat clear	Extremely clear

If you had such a visualisation tool at your workplace (that is already synched with a fully tagged code base), how likely do you think that you would refer to it to help with some development task?

Extremely unlikely	Somewhat unlikely	Neither likely nor unlikely	Somewhat likely	Extremely likely

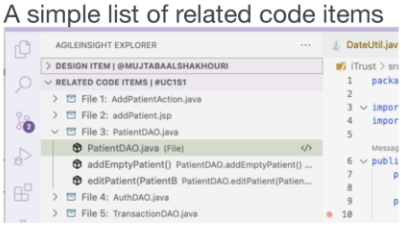
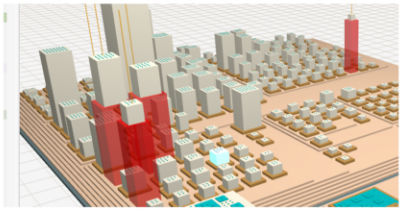
How likely do you think that your TEAM could come to be interested in using such visualisation?

Extremely unlikely	Somewhat unlikely	Neither likely nor unlikely	Somewhat likely	Extremely likely

Below are some tasks that were illustrated or carried out in the demo by utilising features of the synchronised visualisation. For each task, how useful did you find the visualisation in helping to accomplish it? :

	Not at all useful	Slightly useful	Moderately useful	Very useful	Extremely useful
Aiding newcomer in learning about and getting up to speed with a code base	○	○	○	○	○
Group communication (e.g., shared and interactive visual mental map displayed on large screen)	○	○	○	○	○
Revealing insight about the evolution (or erosion) of code base architecture over time	○	○	○	○	○

Feature location (the ability to tell what part of source code implements a given feature or resolve a bug fix) was demonstrated in two ways. How good do you think was each in accomplishing the task?

	Extremely bad	Somewhat bad	Neither good nor bad	Somewhat good	Extre gc
A simple list of related code items 	○	○	○	○	(
A visual impact revealed distinctly in the visualisation scene 	○	○	○	○	(

The potential benefits showcased in the tool were primarily made possible by synchronising source code with requirements, which ultimately enabled the illustrated visualisation and traceability features. We are wondering if it is a **good idea to extend the technique** by incorporating **code tests** and test related data.

How useful do you think such extension would be to developers in answering important questions around code testing (e.g., test coverage)?

Not at all useful	Slightly useful	Moderately useful	Very useful	Extremely useful

In the demos, you have seen **bi-directional traceability** between source code and original requirements demonstrated in a number of ways (including

traceability **snowballing**), which were made readily accessible to users within vscode environment and within the source code itself.

How much help do you think this offers developers of having the information within their immediate reach?

Not at all	A little	Somewhat	Quite a lot	Very much

In summary, the tool tried to combine visualisation with requirement tracelinks to enable a number of functionalities to support the developer.

Overall, how well do you think the tool did in supporting the following tasks?

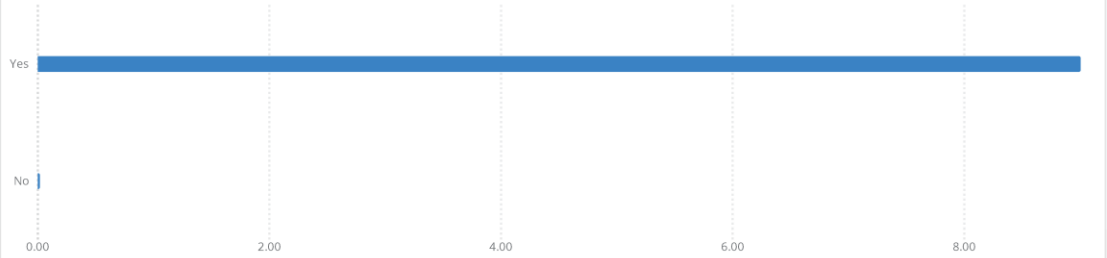
	Not well at all	Slightly well	Moderately well	Very well	Extremely well
Finding out dependencies before commencing a change or a bug fix	○	○	○	○	○
Learning about and understanding a code base	○	○	○	○	○
Long term maintainability	○	○	○	○	○
Code base documentation	○	○	○	○	○
Enforcing traceability compliance (e.g., keeping track of what each piece of code was exactly written for, in case of safety critical systems)	○	○	○	○	○

IX.IV Full Survey Results

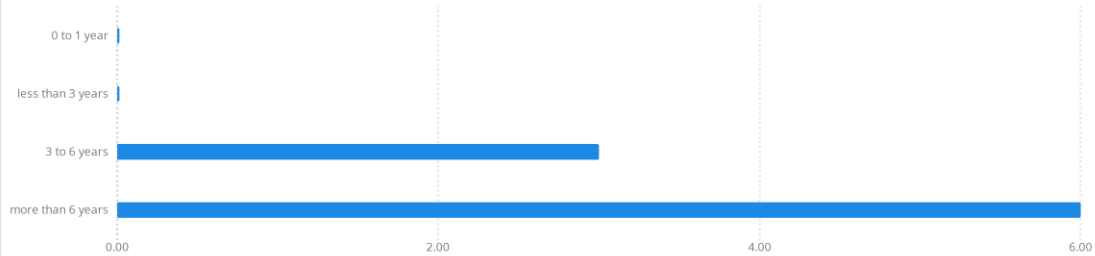
AgileInsight Evaluation - Stage 3 / **Survey Results**

Responses: 9

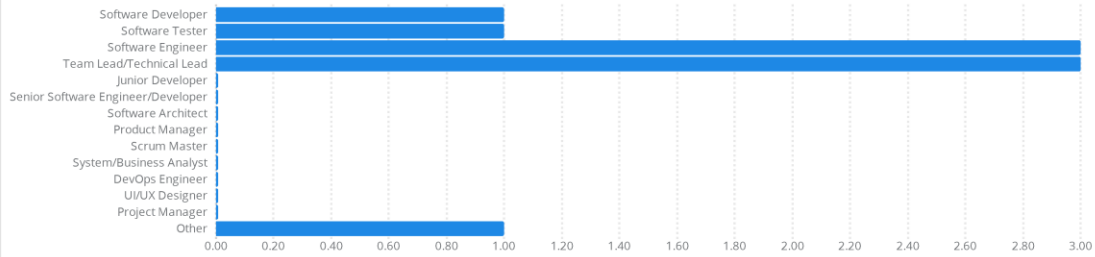
Do you agree to participate in this study?



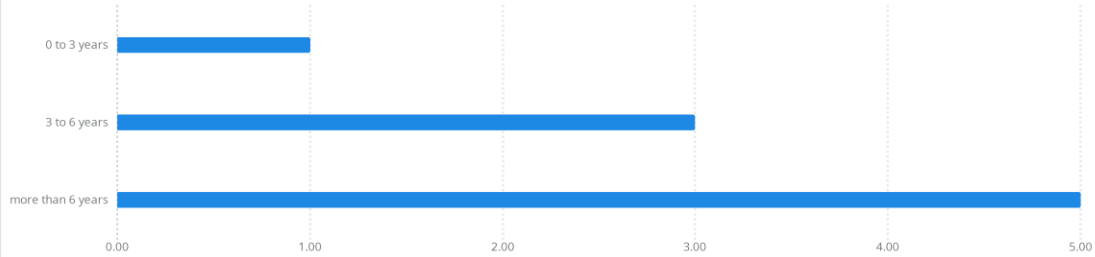
How many years of experience do you have in software industry?



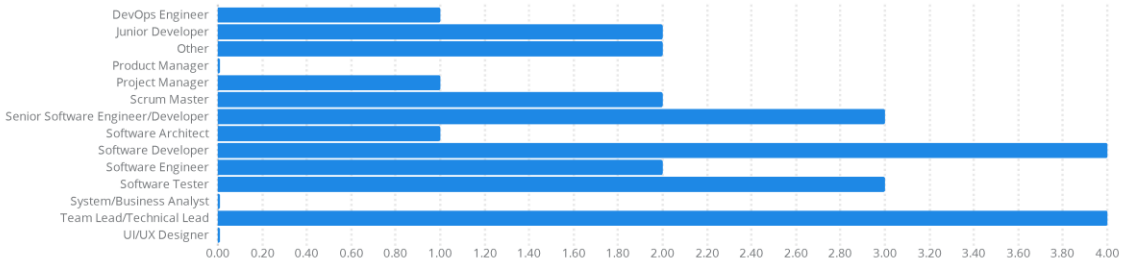
What is your current professional role or title? (chose closest match)



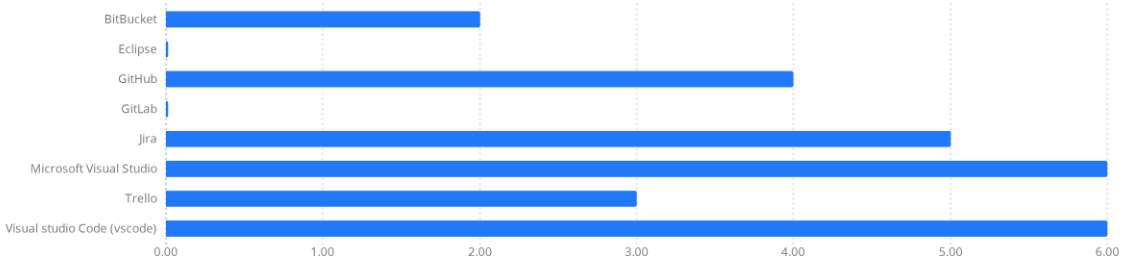
For how many years have you held that role?



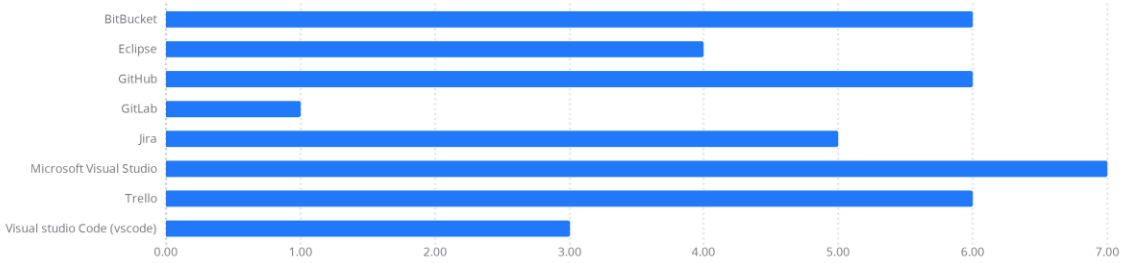
What other professional roles have you held previously? (please select all that apply)



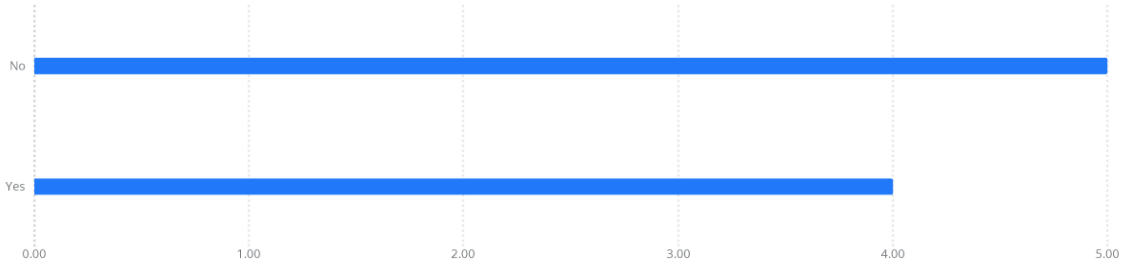
Which of the following tools do you use at your current workplace? (please select all that apply)



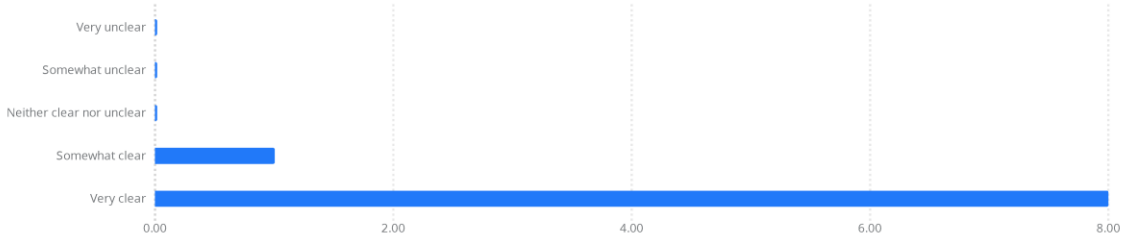
Which of the following tools have you used at previous jobs? (please select all that apply)



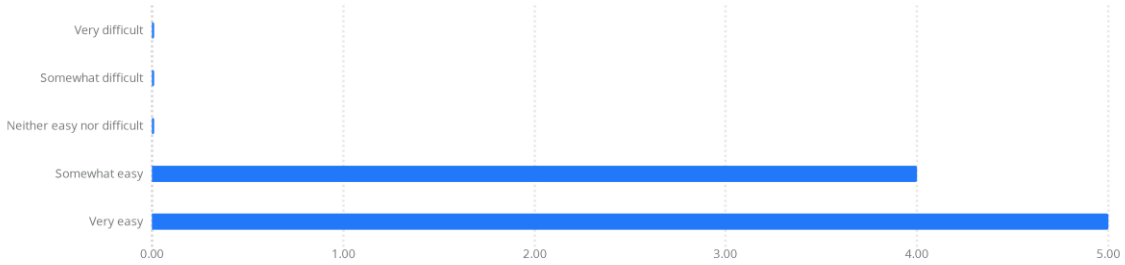
Have you previously participated in an earlier phase of this study?



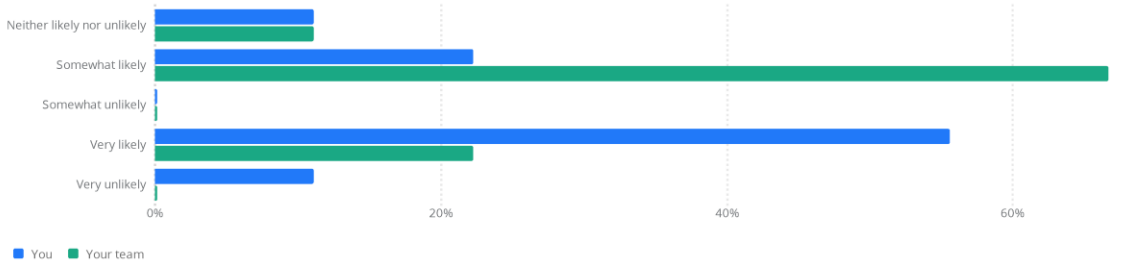
The previous demo has showcased tagging code items with their original requirements (e.g., issue tickets) in a number of alternative ways during development. How clear do you think the benefits are for this kind of traceability tagging?



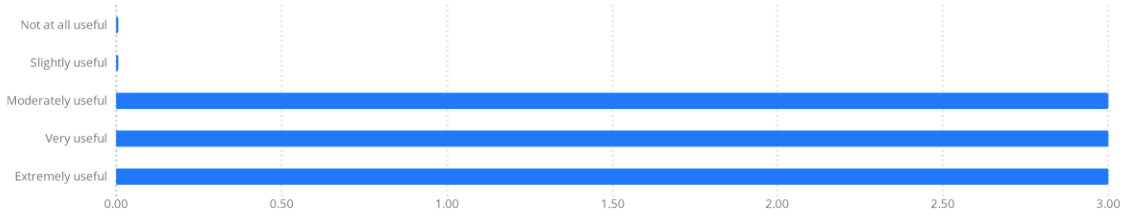
How easy did you find the presented tagging mechanism to be?



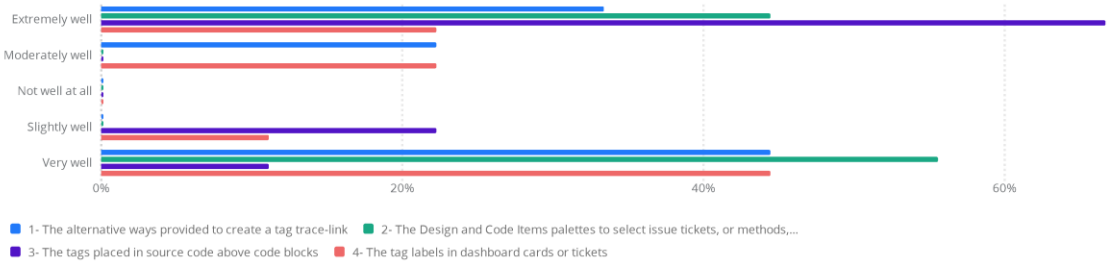
How likely is it that you or your team would use a tagging mechanism like this?



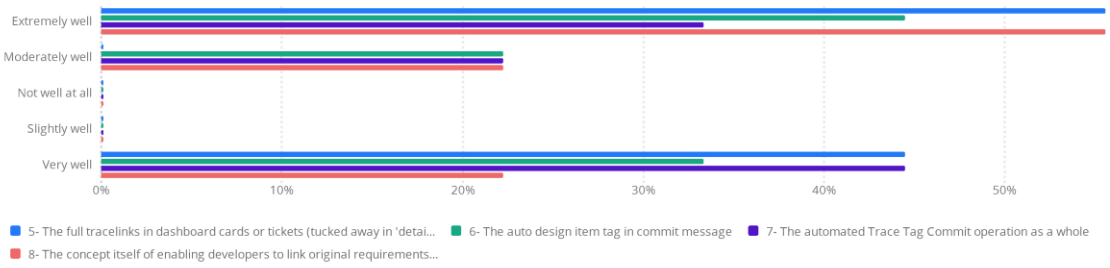
During the demonstrations, you have seen tickets from Trello or Jira board being presented live within vscode and integrated in a number of ways... (e.g.., the Dashboard Tree viewlet). How useful did you find the idea of having the issue tickets made readily accessible within the development environment (or within the source code)?



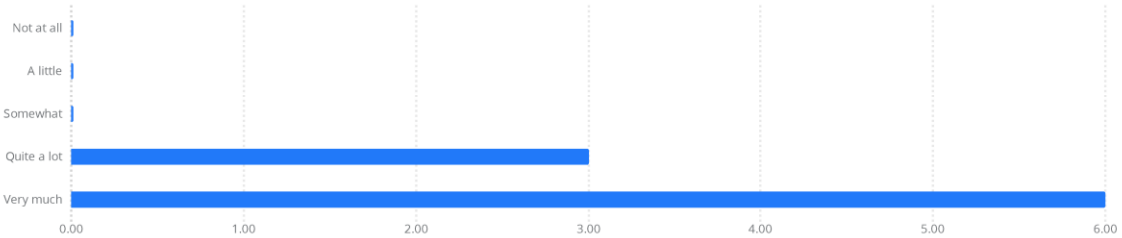
Below are some areas of the demonstrated tagging mechanism. How well do you think each one has worked out? (Part 1)



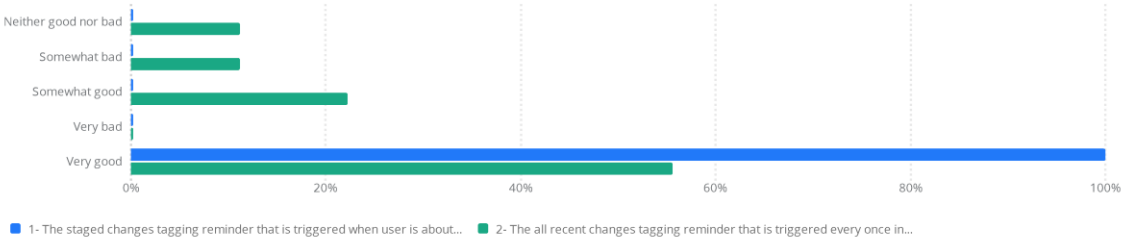
Below are some areas of the demonstrated tagging mechanism. How well do you think has each part worked out? (Part 2)



In principle, and aside from our demonstrated tagging method, how much do you support the idea itself of encouraging developers to regularly tag their code with requirements IDs (e.g., issues or ticket numbers)?

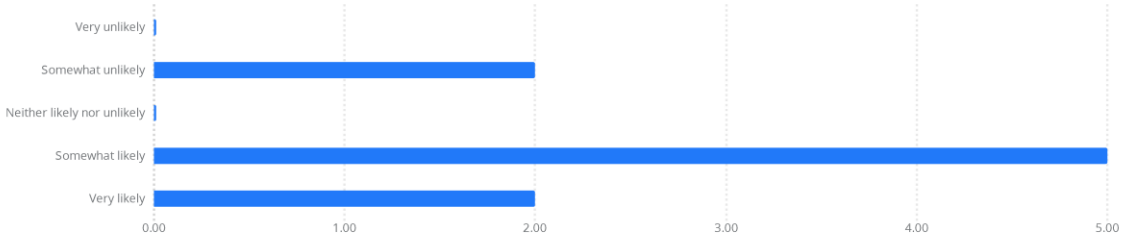


Below are two tagging reminder approaches that have been presented in the demo. How good do you think each mechanism is in helping users develop the habit of tagging their code?

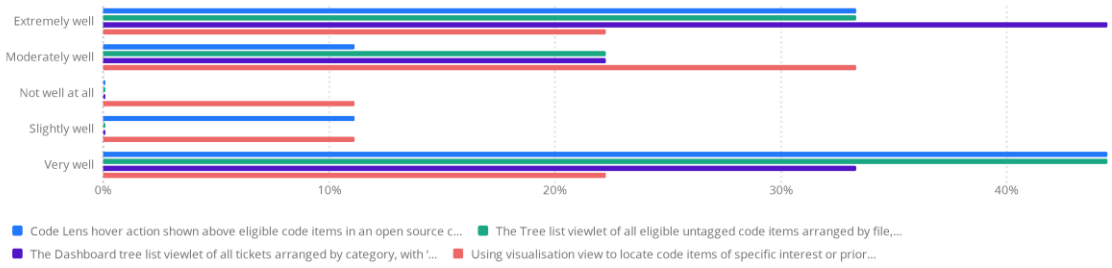


The demo has showcased a mechanism by which recommended tagging for recently changed code items was presented to the user in a tree list viewlet that is accessible at any time. How useful did you find this feature to be in facilitating (or encouraging) a tagging practice?

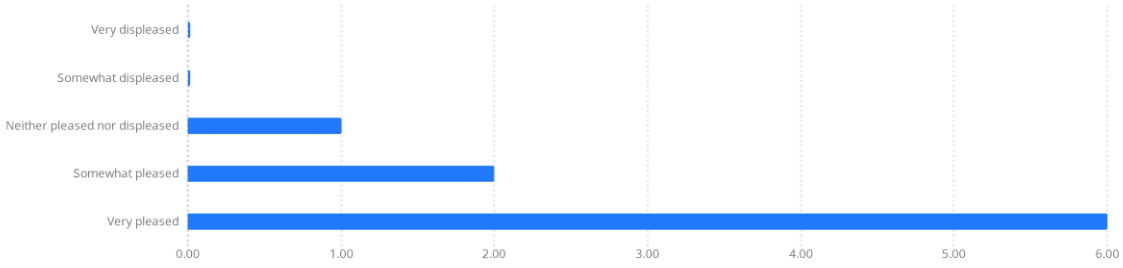
The previous demo has showcased and claimed some potential benefits of having a code base tagged with original requirements... How likely is it that your team or organisation could come to be interested in tagging an existing code base?



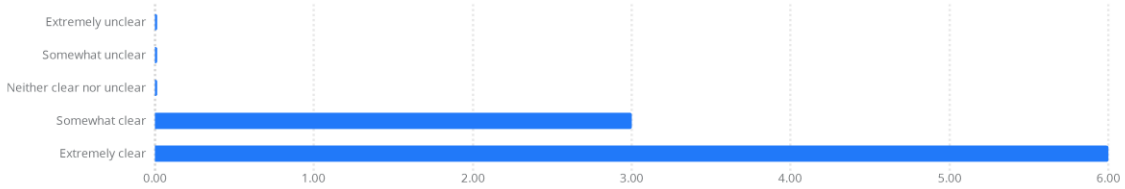
Below are some of the ways presented in the demo to support tagging an existing code base. How well do you think each has worked out?



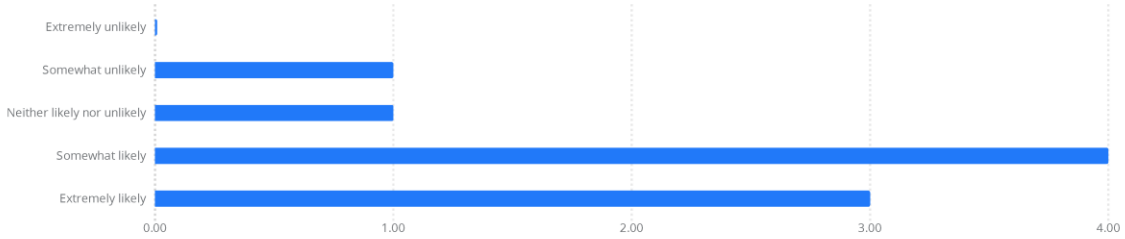
Overall, how pleased are you with the tool's level of support for tagging an existing code base?



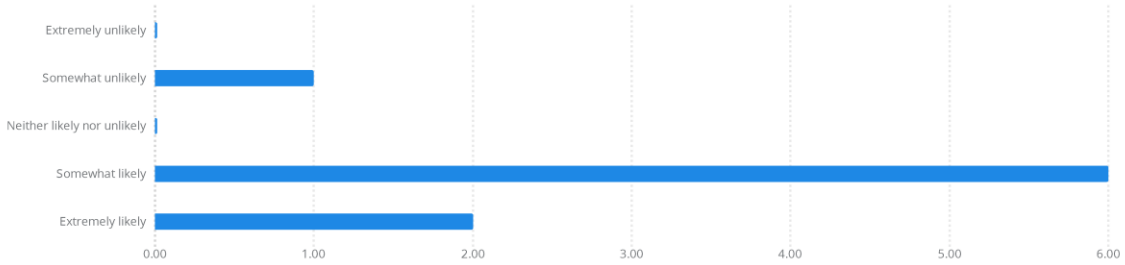
The first two demos have primarily showcased how the tag tracelinks mechanism works and some potential uses it could bring. In the preceding demo, you have seen a visualisation technique taking advantage of those tag tracelinks by synching them into a visual representation of the code base to enable yet other potential uses. Overall, how clear to you was the benefit of the visualisation in supporting the different tasks that were demonstrated?



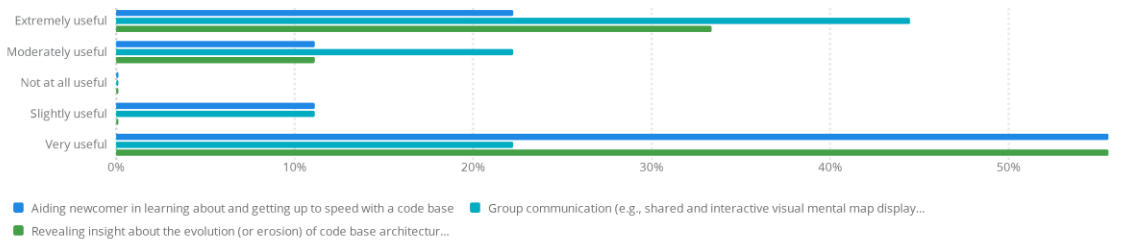
If you had such a visualisation tool at your workplace (that is already synched with a fully tagged code base), how likely do you think that you would refer to it to help with some development task?



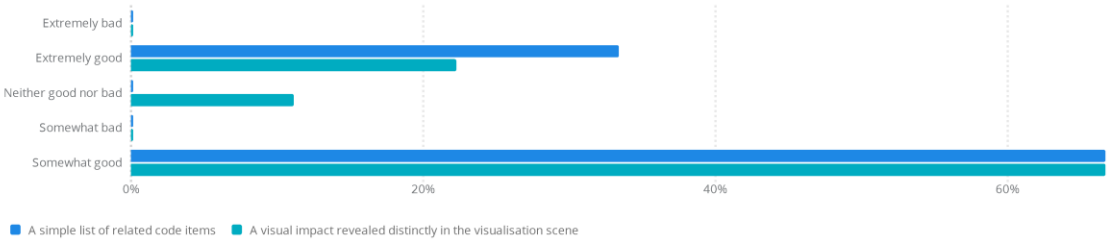
How likely do you think that your TEAM could come to be interested in using such visualisation?



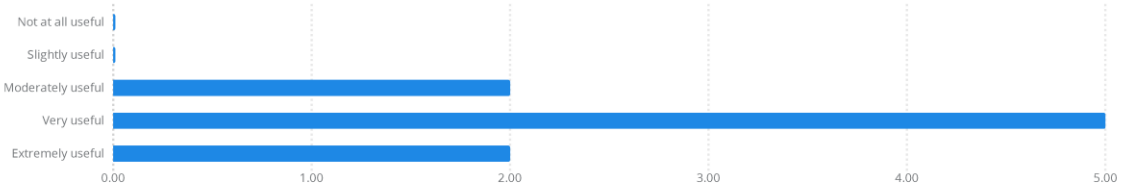
Below are some tasks that were illustrated or carried out in the demo by utilising features of the synchronised visualisation. For each task, how useful did you find the visualisation in helping to accomplish it? :



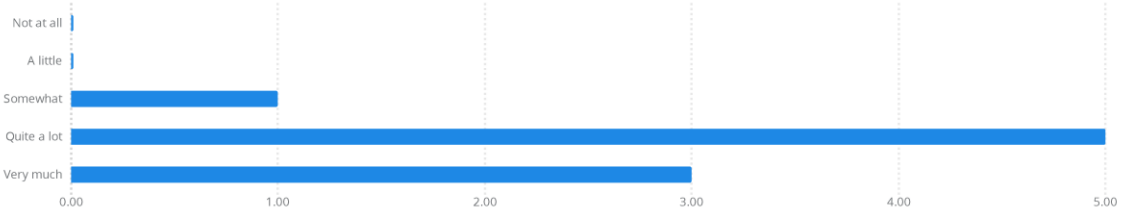
Feature location (the ability to tell what part of source code implements a given feature or resolve a bug fix) was demonstrated in two ways. How good do you think was each in accomplishing the task?



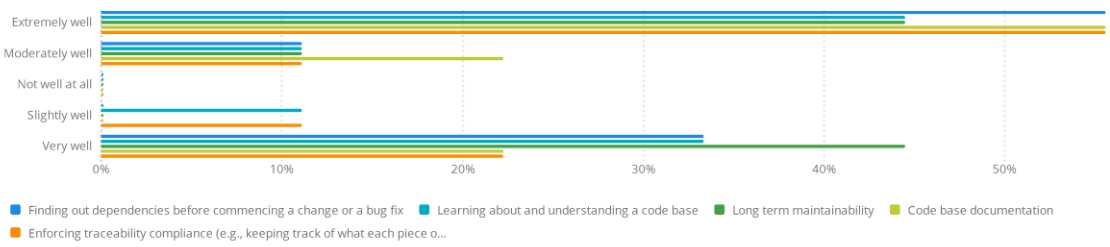
The potential benefits showcased in the tool were primarily made possible by synchronising source code with requirements, which ultimately enabled the illustrated visualisation and traceability features. We are wondering if it is a good idea to extend the technique by incorporating code tests and test related data. How useful do you think such extension would be to developers in answering important questions around code testing (e.g., test coverage)?



In the demos, you have seen bi-directional traceability between source code and original requirements demonstrated in a number of ways (including traceability snowballing), which were made readily accessible to users within vscode environment and within the source code itself. How much help do you think this offers developers of having the information within their immediate reach?



In summary, the tool tried to combine visualisation with requirement tracelinks to enable a number of functionalities to support the developer. Overall, how well do you think the tool did in supporting the following tasks?



IX.V List of Selected Participants' Quotations

The table below shows the full list of selected participants' quotations that were extracted from the demonstration interviews of phase three evaluation activity. Part of these have appeared in Table 8.7.

Table 10.2: Selected quotations capturing participants impressions and feedback at phase three of the evaluation study. These are provided to give a feeling of experts original thought lines.

Feedback/Reaction on Tagging Mechanism	• Participant raised concerns if tags would reflect on most recent version of a file: ", if that file has changed multiple times… are you looking at the final version of the file or you show all the history of the file?... So maybe someone tag the final?" It was then explained that since the tagging involves a commit, tags will always reflect on latest version on a branch. Similar concern was then raised if tag are reflected on feature branch or a master branch, to which clarification was given that tags are always reflected on the branch currently checked out by user, and then reflected on master when merged. • A participant prefers the DI-Initiated method, but still see need for further automation: "I say to do it manually for every method that would probably be unusable to do it. The second way is better, but I wonder if there's a way to actually automate what you've done." • "…so what if ticket has hanged like 80 files 20th that's 3550 then how many labels going to be there?" • "Yeah, it's pretty cool. I don't think it's adding too much of a burden" • "Yeah, I, I think that it can fit into a process… uhm, without disrupting…" • "what if I made the mistake of tagging a file? And can I change my mind in the end or something?" • "Yeah, so far looks quite unintrusive, so it's more about publishing the practice I suppose… it's good, that is straightforward" • A participant was concerned that offering file- and class-level tagging might encourage inaccuracy among developers, choosing instead to only tag at these higher levels: "when it comes to the tagging itself, you know for developers… want to get something out of the way quickly as possible. He may just sort of ripped through and just say, Oh yeah, they're all files. Or they're all classes. …they just require a bit more discipline." • "this makes complete sense"
Feedback on Post-Development (retroactive) tagging features	• "Uh... Yeah I can see where that come in handy… I think maybe the untagged section could be a bit scary to try and look at if you introduce this to a big a big codebase…" • "…but I think you've really just added a lot of useful stuff and given a lot of power to the individual… Uh, as to how they can utilize this, which I think is a positive thing" • Participant comments on untagged list: "Yeah, yeah, that's a good idea. Well, but again it is not so good idea if it's a huge project that has been going on for years because no one wants to spend time tagging... But if your example, if it's a new each project, then yeah, that's a good idea." • Participant comments on untagged DIs: "Yeah, the more the more it's flexible in the more option is the you know for me, the better."

	• "Sounds good, uh... I guess that if you hit big codebase you will have. Will be quite daunting activity to try to tag everything you know… Normally you would try to do things iterative. I guess I start talking as you go along and building it up." • "I can see this useful if you have a relatively new project, something that is, you know within reasonable amount of work to be done. Yeah, it's definitely helpful." • "Yeah, so much trouble with this… legacy. There will be heaps of file.." • When showed the ability to identify which cards/design items are untagged, a participant reacted: "I wonder if that's not a job for the product owner… or party manager. …as a developer, I don't think I would be too worried about seeing which cards *untagged." • After prompted if different ways could suit different users, participant commented: "Yeah, it's good. I think the fact that you've got the flexibility in there too. …teams work differently... you know, so having that option is certainly beneficial" • "I think it might be a little overwhelming to come in and start tagging an existing codebase if you got a huge project already… one of those lists is going to be… an overwhelming number of things" • On the flexibility of different methods implemented to identify untagged code: "if you have a project where you've been pretty consistent about tagging things already, then I think either of those two second solutions will be good to identify things that you missed… what are the areas where we missed adding tags. If they're shorter list, short enough that you can actually browse them, and I think that would be pretty easy." • "Yeah, I think code lens is probably going to take a while. Whereas the other one is better (referring to Untagged Lists)… I think what would make it really good is that you would have like a checkbox, maybe at the top if you were wanting to select all of them, which would then allow you to just skip having to go down and individually do…" • Participant reacted to the Untagged Design Items List: "Yeah, it might be something that um, maybe the development manager would do… he may well do like an audit and go through, but again, on a large project whether he would even have time to do that, I don't know. So yeah, I think it comes down to changing daily work practices and just getting into the habit of doing it. Because you know… importance of fitting in work routine/way of working so it can be done routinely"
Perceived value of AgileInsight's Traceability Features	• Participant commenting on ability to pick a design item (e.g., a bug) and instantly reveal all its related code items, or vice versa, to pick a file and instantly reveal all related design items: "Yep good, yeah, I think that will be powerful. I guess I would say this is one of the most powerful. It's fairly well I will use it, if I can… so give me the file that has most of the bug related to it and then, if then in the visualisation I can find OK this this file got like 20 bugs related to it and then OK this is a problem file, how we can fix that file? Yeah I think yeah cool good." • "I do see the these being the useful like for example something that we do now is that if you have an incident or something breaking, it's good to trace back. Looking back, doing troubleshooting, this was changed through this piece of work and what else was changed about as part of that piece of work and whatnot that now we normally do it with. You know, inspecting pull requests

	and commit logs and whatnot, so that could be a good use for ways to do that too" • Participant describes how DI-to-code traceability could support his developers who work on small increments as part of their continuous delivery scheme to be able to pick up where they were: "…you can kind of go home for the day and then start up again in the morning on the same feature. You can then just go and click on that ticket and find out okay where all the code changes made with the files that I was working in yesterday, right." • "it gives you that very like immediate like context of where you are" • Participant comments on AgileInsight's bi-directional traceability functionalities: "could be useful if you have a few engineers working together on a feature just live right, as people make commits you know other engineers working on the same problem can immediately see what changes you've made and where, right. They can access that directly, so that might be another good benefit as well." • Participant comments on bi-directional traceability functionality of AgileInsight: "… about if you're coming to fix a bug, that's actually really handy what you've got there, because generally when you get a bug, there's no direct correlation between the bug and the code. There's no definite… it's this method or it's in this file that you have to go searching through your code… can be for days. I mean you know on a big project it's just ridiculous." • Participant describes how AgileInsight traceability features could help in debugging: "… by having a list like this of the user stories but also the files that are relevant to that particular uhm method, say if you've put a break point and then you see that's where there's a problem, you can immediately then look to see which file files you could be looking at. So that's actually a really handy part… all that one I like that." • Participant comments on bi-directional traceability functionality of AgileInsight: "That's a really good that you would be able to immediately see, OK, where am I actually going to be looking… I don't have to sift through all my code again to try and work out what I'm actually trying to look for initially, so yeah."
In-editor DI Peek (Quick Details)	• "Make sense cool? Yeah for the usability…, so far so good for me. Yeah, that one is testing in progress… and the board and the person who's assigned. Yeah for me that one is like, uh, more than enough kind of information" • "Yeah, I, I think it's useful. but it has a caveat.. for example, if that piece of code may serve multiple features and each feature might have multiple cards that you have worked on over years… so then that list could become quite large… you know if you have a long list of issue names in there… So yeah, so it's kind of like how, how, how you look into how these will work at scale" • "So the only thing I would say everything looks really good I. I mean I love the way that it all links straight away and you get the tooltip when you hover over that so that you can see exactly what's going on. The only thing I would say is if you have to manually do that for every method that you have Just worked on that could end up being very time consuming."
Feedback/Reaction on Visualisation & mapping technique	• "Awesome. But for me personally I might not... I think it's good to use at first, but for me.. maybe no, I'm only gonna use it like to play around. Then afterwards I'll stick to the tree hierarchy. So that's just me. Personally, I think some people are more visual people, I guess."

	• Participant states that visualisation is useful for architects and system designers, but not for developers who are focused on getting their tickets implemented: "Yeah,… but I just feel this part, maybe yeah, extremely valuable for maybe architect or someone designing the system like a software architect. I don't think it's very valuable for the default developer. I'm against it as they want to get their ticket down." "…but for the system architect who do the integrations with different systems or designing… it will be very good for them.... I think rather than printing something and then try to draw all the diagrams... this will help them a lot and I think yeah, it's great feature." • Early in session, participant found the visualisation valuable and was wondering if it works with non object-oriented code: "..so I'm just wondering so there are programming languages there. They don't have class and then basically like Go language from Google… They just basically get rid of the class, they just don't have any class, but have functions... There's another functional language… JavaScript now have classes, but that lots of people write JavaScript... They don't use class at all, it just file and *bunch of functions. So then how can they show in this *visualisation?" After explaining and demonstrating how it works with any code structure, participant reacted: "OK, that's actually quite cool" • When asked if visualisation could be of use to learn a codebase, participant answered: "… maybe the lead or the senior engineer can explain to the newcomers on this visualisation rather than go through the file. Like look at this file and this. Yeah, I see that yeah nice." • Asked if it would be useful for team meetings or sprint review/retro: "Uh, I agree could be quite useful… and I could definitely imagine having this up next to a slide at a Sprint… it would end up being quite impressive and a useful little tool to explain what you've done, but also engage people more into some of those. Uh, more boring Code changes. • When asked if visualisation could be of use to learn a codebase, participant answered: "Uh, yeah, I can also say the use case of like you know a brand new person coming in...uhm.. 'cause it's you know it's not as daunting as going into the code base and then flicking through each module." • "I guess especially for you know, test engineers and testers who are expected to learn the codebase… this could maybe be a nicer way for them to get started perhaps. Because for some of those people…they're not always too comfortable with code" • "I think this is more of a, let's say a scrum master thing." • "Now that the best thing for this visualization is that when the team is presenting… to the higher management, the management will not understand they are seeing the code… they will be more fascinated when they will see that OK, this is how it is being represented" • "To talk or discuss with the team's so kind of some... Yeah, so with this kind of information makes you more things to talk about. You know to discuss." • "…once you get the labels… working, being able to move around and get that sort of instant *view of these individual affected components was really good too" • "…in a form *give a wider perspective in terms of looking at the health of the codebase and kind of dependencies and things like that.."

	• "… they can generate like pre and post commit *views… go back and double check make sure that they haven't changed anything that they haven't needed to… so yeah, I think it's really helpful" • Participant gave disparately different reaction compared to others: "This is really directed more at the developer than anybody else, isn't it?... It would appeal more to a developer than to any of the other members of a team" • They continue: "… if you were looking to try and visualize something for a scrum master or product project manager, you might want to look at a different style of visualization because I think this they wouldn't quite understand what's going on. They have a much higher level view of the whole development process, so I think a different visualization would suit those" • "I think this is great in terms of demonstrating what you've done. Maybe if you had, uh, a developers meeting and the manager of the development team wanted to see what everybody had been doing, then that's fine. That would work." • "…But I think in a Sprint review or a Sprint retrospective where you've got those more the business owners, and that they really just don't understand this kind of thing. So yeah, there may be a different style of visualization to show..." • Asked if the visualisation help in exposing code structure, participant answered: "I think from a developer's point of view, this would actually be very useful… to understand how the how those parts are and what each one means" • Participants describes visualisation to potential help code review: "that would actually be really helpful for a code review or with developers… if you had a huge block and you can see that, then you've got a lot of methods and a lot of code in that one file or class that you probably would want to break that down into smaller..." • "… wondering if this visualization can reveal aspect of like a raw sense of the architecture, or like maybe something that doesn't look right"
Visual Code Impact Feature	• "That's good, yeah, it's like you know the information you can get out of that... This representation here is kind of good for developers and also for kind of technical leads… they will learn through the software architecture. Another can sort of visualize and see the changes that are happening in the code base, and I'm not aware of any other tools that you can… generate this. So succinctly you know… out of the box. I think it's as a lot of benefit using something like this." • "I think the product owner is happy to see the visual impact because they will not be able to understand… this code and all these stuff, which is kind of maybe boring for them… so this is a good representation for the work which is completed, yeah" • "This is really helpful and it saves the kind of mental effort of actually… scanning through the files in a particular change that you've got to commit to see what it is that you've changed" • "I can see this being potentially useful. Maybe not during day to day coding when you're working on a feature but when you're trying to understand." • "…The sense of like what are the big areas in the application, where's the most of the business logic layer, and all of this right, you might be able to get that from this sort of map"

	• "…You'll be able to see like how widely… the number of things that have been affected by your particular *implementation… sort of alert you to the fact that I shouldn't really have been touching there… I need to refactor or kind of put some sort of an adapter. I think yeah, just in shortly and visually, that's a huge help…. as humans we sort of work better with pictures, so that's definitely be helpful" • Participant describes potential benefit to identify feature distribution across a codebase: "… also maybe identifying problem areas in an application right, what are the features that light up this whole map, what are the things that are spread throughout the application and not very well encapsulated right that sort of stuff you can probably identify with this map room." • Participant related to three use cases then concluded: "…So that is more helpful for like three people, the developers, the reviewers, and the testers… so it will be a good addition as well" • Participants comments on being able to reveal traceability relations in Visualisation: "Cool yeah, I think that was clear good stuff to see the things linked in there"
Dashboard View	• On having design items (cards) presented within the IDE: "I think just in general, I do like the idea of minimizing context switching… gives you a little bit more flexibility. So nice about it" • "…if you got most of the information from the ticket into the IDE then developers might not need to open their ticket tracking system at all right… because you have your backlog right here, you can pick your task off the list and start working on it right in the IDE without switching to a different tool, so that might be beneficial" • "…I like that a lot… I mean that just makes life a lot simpler. You've only got a very small part of the title or the user story… and if you've got a keyword like it's to do with this and you put that in *(to lookup), that's great" • "If you were able to group your cards in sort of like an epic card for a large a feature, then you don't have as many cards down in that list and you have just another layer of a hierarchy… 'cause certainly the number of cards… would be into well the hundreds and trying to scroll down to see which one you want to tag to certain things… would just maybe good to have that extra layer."
Suggestions to further Improve Tagging Mechanism	• "…functions within this file, for example… do you have to go in and tag those independently or can you just get them all kind of automatically tagged when you commit the source?" • Participants recommends further automation if we are to get user adoption: "I think… engineers are fundamentally… lazy about their day to day work, they want to automate things as much as possible and minimize the amount of admin work that you do for each task. So if you help engineers you know automate this process as far as possible. Then that that's when you're going to see adoption." • Participant react to the suggested idea of enabling users to check out a feature at beginning of session to help further automate the tagging: "Yeah I mean, I think that makes sense. You might be able to simplify even further by just saying that when the developer makes it commits at that point link everything that they did to the ticket right."

	• "I think, if you depend on an engineer remembering to do something… just you know, out of their free will, then you probably not going to get a consistent result." • "…we probably need to come up with a way where engineers are either prompted to do it, or it just happens automatically as part of their workflow, so it doesn't depend on them always remembering to do this thing right" • A participant suggests selecting a card at beginning of session: "…if you were to select a card that you're working on initially that then it can actually automate and see where you've made changes and automatically add those up…" • They further add: "when we use DevOps for our Visual Studio Code, when I go to push changes it will have a list of all of the files that I've made changes to. If you were able to take that list and automatically add those to the card, from what I was about to push or do something similar to that that would then…" • Participant backs up the importance of supporting different ways of working: "Yeah, that's good though… if you can set it up so that you can tag your code specifically to the way you work, that would definitely make it very usable. Because like I say, everybody works differently anyway. I think just giving people options… like to see is that it's not been developed just for one specific style of working. It's giving them an option to come to fit it into this work style."
Suggestions to Improve Visualisation	• "I think if like every unit test and UI tests were tagged and they could be included that that would be awesome." • Participant comments on the need to properly scale the buildings ensuring they don't grow exceedingly large—supporting our argument and approach to adopt dynamic mapping functions: "in terms of the class size… You may want to come to the surface in how you might sort of scale that… you know some of those bigger or legacy codebase out there, we've gotta some really big classes… god classes. But then you'll have some very very small ones as well. So maybe scaling could be something you might wanna make sure works" • Participant comments: "might be good to have the name of the folder there…", in reference to the lack of the labels in the visualisation
Tagging Reminder	• "I did kind of wonder, you know what happens when some people do this and some people don't. Uh, so seeing that there's a reminder there too. Uh, it's quite cool" • "Yeah… that for me that's great." • "Yeah, that's good. You could even… enforce that a little bit further, like you could actually fail the build like if you detect some changes in a commit that hasn't been linked… sort of forced everyone the same way… have sort of quality checks.." • After questioning if teams would want enforcement of tagging, participant suggests to have it configurable (similar to linting/static analysis tools): "if you were to build something like that… I would kind of make it a configurable… option, down to the quality processes of individual teams"
General Impressions & Feedback	• Participant asked early in demonstration if extension is already available for download, then commented: "…OK great, we will use definitely when available" • Privacy. Participant was wondering if data was stored in backend database and cited concerns on privacy of collected data: "…do you have a database

	somewhere to track keep track of what happened." After explaining that all data is fetched live from users dashboard and repositories, and that no caching or local storing of data, participant commented: "Oh yeah, so that's great… I was thinking if you got the database then maybe there will be lots of security related problems but you got rid of that problem… you don't save anything anywhere, so I think that's good." "…developers want to know… or the company choosing a tool to use… they maybe want to know… like where do you take my data to somewhere, store it somewhere and analyse it.. so people want to see is there a clear kind of message.. that there's no external storage anywhere… that probably give them more confidence to want to explore more" • Performance. Participant raised concern on tool performance for large projects: "…if you've got a big project like… millions lines of code, how fast this thing can run, like is that *would require very expensive computer to be able to use it without delay or loss of time?" • "…I think it's a really great tool here. But then you know there will be in terms of adoption…" • "I think particularly around extending the tagging and how you might enforce that. But I think from first look that covers visualization and the ability to relate chunks and classes of code to tickets and back again. There's is really, really helpful and having that altogether succinctly… in one tool like within the IDE. It's really great. I can see a lot of time being saved there with them" • Participant comparing their existing practice versus what the tool could offer them: "We otherwise have to sort of chop, chop back and forward to update tickets and things with descriptions and things… like you know the work item tickets and also within source control too. So having this thing to be able to tie them all together is really good. I think it's going to increase productivity... so it's good here" • Participant elaborated on importance of fitting into way of working, while reiterating the potential they see in the tool: "When you're working every day and you just get used to doing the same routine as it were … tend to sort of do that blindly without thinking after awhile, so adding a new step would be just a little bit of a learning you know, just remembering to do that. But… we change the way we do things when you're using different software or the time as things get updated, so I don't see it being a big problem as long as it's streamlined enough that you're not having to take yourself out of the flow of your normal daily routine. • Participant continues: "But yeah, I mean I really like the idea and certainly in terms of bug fixing I think it would just be make things a lot easier to try and trace… 'cause like I say in the past I've had to sift through lots of extraneous files and you can't always debug… so yeah it would make things a lot easier if you knew which files were linked to what you're trying to fix that. That would be great. • "My takeaway would be definitely the bug fixing because it doesn't matter how carefully you code, doesn't matter how well you think you've done your unit testing, bugs just come out of nowhere from sometimes things you didn't even realize you know… so that would be really helpful, really good. ", participant added in concluding the session, referring to the bi-directional traceability features offered by AgileInsight as mostly useful to them.
General Suggestions	• Participant expressed the tool is of more value to large projects: "I think maybe not for every project… for the very small project like.. they may not feel this is a very good thing, but if you know, mission critical, you know SpaceX and

	all those regularity of banking financial, they definitely would look at something like this." • "… I think it's a cool tool it's something that you definitely have to kind of play with it a little bit to figure out how to best use it, but I can definitely see applications." • Participant suggested to have labels showing in cards to be configurable so can hide if label numbers go high. • Participant concerned if user accidentally deletes identifiers from card: "if the user don't know what they're doing, they go to the Trello board and then they just somehow deleted the label". It was then explained to them that ideally identifiers would be stored on a hidden or less visible field, but current Dashboard API does not offer such properties. • "in general, it's hard to visualize source code, I think this is a definitely interesting visualization to… especially to highlight problem area, so think I'd love to run this on our applications and find out what it is like." • "… but no really good presentation. I love the idea. I think you've got a winner there. If you were able to get that commercialized and as an extension to… dev programs that would be awesome"