ScrumCity: Synchronised Visualisation of Software Process and Product Artefacts

Mujtaba Alshakhouri

A thesis submitted to Auckland University of Technology
in partial fulfilment of the requirements for the degree
of
Master of Computer and Information Sciences (MCIS)

School of Computing and Mathematical Sciences
March, 2013

Supervised by
Professor Stephen MacDonell

Contents

List of Figures

List of Tables

List of Abbreviations

SV –Software Visualisation

SE –Software Engineering

LOC –Lines of Code

NOM –Number of Methods

NOA –Number of Attributes

WBS –Work Breakdown Structure

jME3 –jMonkeyEngine3 (a 3D graphics library)

Attestation of Authorship

"I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgments), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university of other institution of higher learning"

Mujtaba Alshakhouri

.....

Acknowledgments

This research endeavour would not have been realised without the academic and personal support of several people. I hereby would like to express my sincere thanks to everyone who has supported me either academically or personally during this important period of my academic career. I especially would like to extend my utmost gratitude and thanks to my supervisor, Professor Stephen MacDonell, whose continued guidance, support, trust, and encouragement were truly invaluable to my completion of this research. Thank you, Steve!

My sincere thanks and appreciation go also to all of the staff and colleagues at the SERL laboratory for their personal support and encouragement and for making this research experience both pleasant and inspirational. A special thank you goes to Professor Neville Churcher from the University of Canterbury who was specifically behind my first introduction to the exciting research field of Software Visualisation during my earlier studies back there. A genuine thank you goes out to Jacopo Malnati, whom I have never personally met but who has heartedly provided advice and support over a distance.

Lastly, I would like to wholeheartedly express my warm recognition and appreciation for my wife Zainab Alqallaf, and two daughters Mariam and Reima, for their endless patience, for keeping a high spirit, and for standing beside me during those difficult and challenging times.

Abstract

Software comprehension is a complex and ongoing challenge facing the software development industry. The often immense number of interrelated components in contemporary software systems places a high cognitive load on software stakeholders, whose job requires deep understanding and awareness of those constituting components. Among many approaches, 3D visualisation of the software static structure has recently emerged as a promising approach that is increasingly being demonstrated to significantly help in alleviating that cognitive burden by exploiting and leveraging humans' natural perceptual abilities.

Furthermore, in addition to easing comprehension and increasing awareness of constituting software artefacts, this technology has the potential to bring *visible* various important aspects of the software *process* that could potentially make this technology a valuable tool for a wider spectrum of software practitioners. Recent literature, however, shows that the majority of prior research has limited itself to visualising the software product and in the best cases, only highlighted *some effects* of the software process.

This thesis identifies and attends to this gap in software visualisation research by introducing a novel visualisation approach named Conceptual Visualisation. It asserts that visualising the software process not only has several potentially beneficial implications for the software industry, but that from a cognitive perspective, visualising that process *in the context* of the software structure is particularly suitable and significant to increase human awareness and understanding of both the processes and their implemented product artefacts. The proposed approach is designed and constructed following a systems development research methodology and adhering to the principles of sound design science research. It is then assessed via functional demonstration, being applied to six open source systems of varying size and complexity. Conceptual Visualisation is shown to make a novel contribution to the software visualisation research literature, addressing many prior stated requirements in doing so. Once developed beyond a proof of concept, its use in practice should bring multiple benefits to a range of software stakeholders.

1

Introduction

This chapter provides a brief background and introduction to the software visualisation field of research and highlights the contexts and motivations from which this research arises. This leads to a statement of the problem that is addressed through the research reported here. Research objectives and anticipated contributions are also introduced and discussed.

1.1 Background and Problem Statement

Software intangibility is a well-known problem that is frequently attributed in the Software Engineering (SE) literature as being the major cause behind the inherently high complexity of software systems (Caserta & Zendra, 2010; Gračanin, Matković, & Eltoweissy, 2005; Claire Knight & Munro, 2000a). The often large number of constituting components is one factor that contributes significantly to the complexity of a system, but the virtual and non-physical nature of those components further exacerbates the difficulty faced and adds further cognitive strain on stakeholders who are trying to understand a software system at hand.

Unsurprisingly, the SE discipline is hence abundant with a wide range of research that is specifically directed to tackling this intangibility and the comprehension challenge associated with contemporary software systems. Several approaches and techniques have been studied and proposed to mitigate this issue, and software visualisation (SV) has over the past two decades gained prominent popularity and attention as one of the most promising solutions. Indeed, software visualisation is now a well-recognised field of research and practice in the SE community (Wettel & Lanza, 2011; Wettel, 2010) and has been empirically demonstrated to significantly support comprehension and reduce the cognitive load faced by software stakeholders while undertaking various categories of comprehension tasks (Carneiro, Orrico, & Mendonça, 2007; Cornelissen, Zaidman, & van Deursen, 2011; Sensalire, Ogao, & Telea, 2009; Wettel, Lanza, & Robbes, 2010; Wettel & Lanza, 2011). More specifically, and more relevant to the research path adopted here, recent 3D visualisation techniques and prototype tools are now at the forefront and state-of-the-art in SV research that aims to aid human comprehension (Caserta & Zendra, 2010; Teyseyre & Campo, 2009). The fundamental concept behind most software visualisation work revolves around the application of visual metaphorical imagery to bring alive the intangible software artefacts (products) to the human perceptual skills, hence scaffolding humans' cognitive abilities to comprehend software systems. The drive behind this is to harness and leverage the natural visual – and typically subconscious – capabilities of the human brain for obtaining knowledge from the outside world. More detailed discussion on this topic is provided in Chapter 2 as an exploration of the relevant literature.

Software comprehension, nonetheless, is a very broad topic that encompasses a wide range and categories of software *tasks and processes*. Yet, the literature shows that the vast majority of both established and recent research on SV is mainly concerned with one of (or a combination of) three classic contexts of use; comprehension of the static structure of software (e.g., packages, classes, methods), comprehension of software evolution (evolution of those static structures and their attributes), and comprehension of program behaviour (during runtime) (Diehl, 2007a). Most recently, a few studies and approaches have explored software visualisation from the perspective of *human activities* rather than the software structure (Ogawa & Ma, 2008, 2009), while even fewer have attempted to combine both perspectives in the same visual scene (such as the work of Rigotti (2011)), by providing metaphorical representations of the software structure and then highlighting some aspects of (typically) team activities on top of those visual metaphors.

Software visualisation, as a field of research in the SE discipline, is however "*still in its infancy*", as remarked by Michele Lanza (an authority[1] on contemporary SV research) in the 2009 Java and Object Orientation (JAOO) conference (now known as *GOTO*). Indeed, there are numerous potential contexts of applications that are still waiting to be explored and investigated in the area of software visualisation. Software visualisation has successfully materialised what has long existed only as textual and intangible product artefacts and this opens the door for various other software processes to benefit and take advantage of such technologies. One pertinent issue to highlight here is that most current research dealing with visualising the *static structure* of software only *re-presents* data (albeit, visually) that is already found there in the source code (Petre & de Quincey, 2006). They do not provide users with new knowledge or information that could potentially help them in their various tasks and activities. While some approaches exist that extract potentially beneficial data from version control systems or configuration management systems, the visualisation techniques employed in these approaches are based on perspectives other than the software structure (most are in fact *information* visualisation more so than they are *software* visualisation), and are found for that reason to have not utilised the fundamental capabilities of software visualisation for promoting comprehension of

[1] http://jaoo.dk/aarhus-2009/speaker/Michele+Lanza

software (Storey, Čubranić, & German, 2005). Furthermore, of those that visualise the software structure, very few were found to have augmented their visualisations with new knowledge, examples being Wettel's (2010) approach that highlights design problems and the work of Steinbruckner et al. (Steinbrückner & Lewerentz, 2010; Steinbrückner, 2010) that contextually highlights some aspects of software evolution and modification history.

1.1.1 Conceptual Visualisation

Petre and de Quincey (2006) have emphasized the need for software visualisations to present information that is typically not found in the code, stressing specifically on capturing the originators' real intentions and their rationale behind the implemented code components. In an earlier paper (2002), Petre introduced the term "*Conceptual Visualization"* suggesting that it is more significant to visually present the *conceptual design* in the visualisation rather than merely re-presenting the *implementation* by itself. She further added (2006, p.6) "… *because the information most crucial to the programmer – what the program represents, rather than the computer representation of it – is not in the code. At best, the programmer's intentions might be captured in the comment".*

The expectation underlying Petre's viewpoint is that software will quickly evolve, developers and team members will change, and it is very common for documentation to quickly fall behind. Maintenance can then become overwhelming as developers will have only the source code from which they have to deduce the intentions of the original developers and the rationale behind the implemented code components before they can contribute further components to the system or maintain its existing code base. While presenting the code components — that is, the implementation — *visually* is significantly valuable and can ease comprehension (according to the literature), still yet, augmenting that visualisation with some form of original conceptual model and information concerning the processes that created each of those components (which are normally not available in the code) is considered to be particularly of even more significance and value. This typically unavailable knowledge has the potential to inform other software tasks and activities for various stakeholders.

Further details and discussions on this prospect are presented in the next chapter as well as in Chapter 5.

According to the literature survey conducted for this thesis, and reported in the next chapter, presenting the *conceptual model* as well as the *software development processes* that resulted in the different software product artefacts, right within the context of the software structure, has not been undertaken to date in SV research.

1.1.2 Problem Statement

> *Current software visualisation approaches do not make available the conceptual design — being a representation of developers' original intentions, their rationale, and development activities (also referred to in this thesis as 'software development processes') — behind software systems. Most approaches focus on visually re-presenting the already available implementation (which exists as textual data) using visual imagery to support comprehension.*

In this research, a need has been identified to augment software visualisation techniques of the static structure with information about the software development processes — *which capture developers' original intentions, rationale, design concepts, and activities* — and present this collectively in a synchronised mechanism alongside related software product artefacts directly into the visualisation scene. Those software processes possess information that is important and valuable for various tasks but are found to be commonly not documented. Moreover, developers inspecting the source code typically have no immediate access to this information even if it were actually available somewhere in the organization. Hence, linking and synchronizing these valuable pieces of information right alongside their relevant code artefacts, that is, the artefacts created as a result of those processes, and then presenting that visually in an interactive environment is conceived to potentially introduce many advantages and implications for the software development industry (Petre & de Quincey, 2006). Further justification for the significance of presenting this information to stakeholders is presented in the following sections.

1.2 Motivation and Rationale

The motivation and rationale behind this research can be realised by considering three points: the original fundamental drive behind software visualisation, the state of the art in software visualisation taking into account specific areas lacking attention, and the availability of an opportunity where addressing such a lack has the potential to bring real benefits and applications to the current software industry. The discussion that follows presents and discusses these three essential points.

Departing the Textual(-Only) Dimension

The role of visual models and diagrams in helping to convey and share knowledge is well-known across almost all scientific disciplines. Communicating information through imagery has been universally utilized by humans since the beginning of time. It is also a well-known fact that the human brain perceives knowledge most primarily via the visual system (Teyseyre & Campo, 2009). Moreover, all modern day engineering disciplines depend heavily on visual diagrams to convey knowledge. Despite all that, and despite the fact that software systems are considered amongst the most complex artefacts that humans has ever created (Panas, Epperly, Quinlan, Saebjornsen, & Vuduc, 2007), prominent software architect Grady Booch remarked in a 2010 keynote presentation (at SOFTVIS'10) that the vast majority of developers still live entirely in the textual dimension, and "*like Flatlanders, have no understanding of or desire for the visual dimension save for a few diagrams with dubious semantics that they may hastily and ethereally sketch on a whiteboard*".

Rather than the developers being at fault, it is contended that the above situation is largely the case due to the fact that software is intangible – something that sits it apart from almost all other engineering disciplines. Software development shares with other disciplines the fact that each has their own special processes that are systematically followed to produce a desired product. However, in the case of software development, the product is made unique by its non-physical and intangible nature. Hence, visual models and diagrams cannot be put into immediate application. The products in this case have no direct physical manifestation against which they can be compared. Instead, metaphorical representations have to be devised first and then mapped to the

different aspects of the intangible software artefacts. It is presumed here that this intermediary step is the obstacle that has truly hindered – and hinders still – the software engineering discipline from taking full advantage of the visual dimension, i.e., from adopting visualisation technologies early on like other natural science and engineering disciplines. In fact, almost all software visualisation researchers agree that the main challenge for SV is in finding the effective mappings from the different software aspects of interest to suitable graphical representations (metaphors), that can support the maximum exploitation of humans' visual perceptual (and interpretational) skills (Caserta & Zendra, 2010; Gračanin et al., 2005; Teyseyre & Campo, 2009). Taking the above into context, this research is motivated by an assumption that software visualisation has numerous implications and potential applications to offer to the software industry.

Contemporary Software Visualisation

The section above sheds some light on the original motives behind the emergence of software visualisation as a field of research, and has concluded that it is fundamentally concerned with supporting human comprehension by using metaphorical imagery to minimise cognitive load. As addressed in depth in the literature review (Chapter 2), SV research has evolved rapidly over the past two decades and researchers have introduced a multitude of techniques and prototype tools, some of which have proved to be highly effective and promising. It has been observed, however, that the majority of current works in SV are primarily oriented toward addressing the *product* – the software artefacts – whereas the *process* has been rather left behind. Petre and de Quincey (2006) noted that visualisation in other domains is primarily oriented toward the *development process*, not the artefact.

It is not difficult to deduce why this is the case. In most other disciplines, the eventual product is physical and can be readily experienced and examined, whereas the *process*, which can be seen as being more important to the producing organization, is ethereal. It is therefore unsurprising to see visualisation being utilised in those fields to capture and communicate a common picture of the development process amongst interested stakeholders. On the other hand, in the software industry, both the product and the development process are intangible. Software visualisation has largely succeeded in

materialising the product artefacts; it now needs to move on to the next step, which is visualising the development process. Some researchers have even expressed that there is currently no lack of *techniques* in software visualisation, but rather, there is a true lack in addressing other important aspects of software and developing proper mappings to present those aspects in context alongside the visualised artefacts (Storey et al., 2005). In fact, no single tool or approach seems to exist that considers the presentation of software development processes in the context of the software structure; that is, presenting artefacts alongside their original development processes.

Software Development Processes in Agile Methods

Since their early advent, agile development paradigms have been much celebrated as being amongst the most successful software development processes. They have been instrumental in saving the software industry from a long notorious period of high project failure rates. Thus there is significant motivation to consider agile processes in particular in terms of supporting contemporary software development. Nonetheless, most agile methods have been criticised for advocating a minimalistic approach when it comes to documentation. For example, in the Scrum methodology, system requirements typically exist only as user stories that are recorded, temporarily, on sticky paper notes (or sometimes their digital equivalent). Those user stories are eventually transformed or manifest into real software artefacts – the *product*. Over time, as the development of the system advances, the system requirements, as originally represented on those paper fragments, tend to get neglected and forgotten. Soon, it becomes almost impossible to track individual software artefacts back to their original processes – the user stories and the ***Scrum Artefacts*** that have produced them.

This is a well-known issue in software engineering that is often referred to as requirements traceability (and is also described more generally as artefact traceability), referring to the ability to trace the implemented code components back to the original functional requirements (as further discussed in Chapter 2). The entire software system becomes eventually detached from its original development processes. Those development processes, however, carry important information that is considered valuable for various stakeholders. Managers (or developers), for

example, cannot then trace a certain feature in the system back to its original specification. Such tracing-ability is important if there is a need to verify whether a certain feature was implemented according to its original specification. Another example is a developer who is trying to maintain someone else's work but who may not be able to easily understand the original intent or concept of the component that needs to be maintained. In fact, researchers claim that one of the most practised software tasks amongst developers is Concept Location (also called Feature Location), which means finding the part of the source code that implements a specific domain concept (Kuhn, Erni, & Nierstrasz, 2010; Xie, Poshyvanyk, & Marcus, 2006). Chapter 5 presents more details on this topic when considering the potential applications of SV.

This '*detachment*' state between *product* and *process* is in fact a common issue not just evident in Scrum practice, but even in many traditional development approaches.

Summary

Reflecting on the prior discussion of the fact that current software visualisation approaches have failed to address the conceptual design model in their visualisations, the complete picture of the primary motivation behind this research should emerge. Software development processes in the case of Scrum practices are being captured by user stories and those in turn collectively capture the conceptual design model –the originators' intents, concepts, and activities. In other words, the user stories and details of their enactment account for the system's conceptual design. Currently, Scrum processes and their resulting software product artefacts typically exist separately. This research envisions that a promising potential exists in exploiting software visualisation to bring together and synchronise these two currently detached artefacts – the Scrum process artefacts and the software product artefacts. This research work emanates and derives from this particular vision.

1.3 Research Objectives and Contributions

Taking the above discussion into account, and addressing the legitimate need to bring together software artefacts with their original software processes, this work develops a novel visualisation technique that captures and visually presents software processes — design concepts, intents, and development activities — linked and synchronized to the software artefacts produced by those processes, directly into the visualisation scene. A prototype tool is further developed as part of this research to implement this new visualisation approach and demonstrate its potential capabilities. The three subsections that follow describe the three core objectives of the research and the contributions that derive from their achievement.

1.3.1 Capturing and Presenting the Conceptual Design

This work shares many connections and commonality with the Petre and de Quincey (2006) earlier call to integrate a system's conceptual design within the visualisation scene, a call that forms the first research objective. The research reported here is seen as an extension of Petre's original idea that takes it to its natural next level. The popular *Scrum* practice of the agile software development paradigm has been identified as presenting a promising opportunity to demonstrate how a conceptual design can be integrated into and presented using present-day software visualisation techniques. Furthermore, this work demonstrates several potential application contexts resulting from the proposed visualisation approach that are foreseen to support different stakeholder groups in undertaking various software tasks and activities. This research builds on and makes use of two pivotal concepts, the Scrum practice and the City Metaphor.

Assuming a Scrum development environment, the *Scrum artefacts* – primarily user stories in this case – and their enactment activities capture to a great extent the original developers' intentions, concepts, and rationale behind each *software artefact* created. Hence, *Scrum artefacts and activities* in this situation represent the conceptual design that Petre emphasised as a critical aspect missing from current software visualisation techniques. Furthermore, the Scrum practice is highly systematic and modular, and so its data model has a consistent and reasonably standardised

format rendering it fairly easy for collection and then correlation with software artefacts. Hence, by merging and synchronising these two normally detached artefacts, the visualisation approach introduced here enables stakeholders to visually examine and reason about individual system artefacts contextually and alongside their original concepts and processes.

The popular city metaphor of 3D software visualisation is considered to be appropriate in terms of supporting the synchronised representation of product and process. Its metaphorical representations capture and convey the true structure of software artefacts and it has been shown to be highly flexible in accommodating various facets of software information. Moreover, the city metaphor has been empirically demonstrated to significantly reduce cognitive load and aid human comprehension of large-scale software systems (Wettel et al., 2010; Wettel & Lanza, 2011). Taking this into consideration, this research adopts the Wettel and Lanza version of the city metaphor introduced first in their 2007 paper (with some slight modifications) (Wettel & Lanza, 2007a). In fact, by integrating the Scrum artefacts within the city metaphor, this work lends further support to Wettel's primary claim in his PhD thesis that the city metaphor is highly versatile (Wettel, 2010).

1.3.2 IDE Integration

One criticism directed towards many current software visualisation tools is that they are implemented as standalone applications, hence significantly impeding their adoption and practical use in the software industry and the wider SE research and practice communities (Kienle & Muller, 2007; Sensalire, Ogao, & Telea, 2008). Even though the recent few years have witnessed the appearance of a few IDE-integrated tools, their number is still very limited (and is more so for 3D-based tools, in particular) and most are short of functionality that would render them practical or useful to stakeholders. In fact, based on the literature survey of this research, only *four* such 3D-based tools that are IDE-integrated exist, namely Citylyzer, Manhattan, EvoSpaces (see Chapter 2, section 2.2) and VisMOOS (Fronk, Bruckhoff, & Kern, 2006). Hence, a second objective of this research is to address the lack of integration of visualisation tools with IDEs by developing a proof-of-concept prototype tool as an Eclipse plug-in, to make it potentially more accessible and hence useful to developers and

practitioners in the SE community. Eclipse was chosen due to its large user base and strong support for plug-in development.

1.3.3 Feature Richness in Visualisation

Besides the shortcoming of being standalone, another highlighted drawback of existing software visualisation tools is their limited support and the modest set of features made available to the user. While the visual presentation of otherwise intangible system artefacts is undoubtedly an advantage, to render a tool practically usable and realise the true power and capabilities of software visualisation many researchers (see Chapter 5) have stressed the need for feature-rich tools that can solidly demonstrate the practical benefits of SV to the SE community and industry. Even though the tool presented here is a prototype and a proof-of-concept, one important objective of this research is to take advantage of new technological advances in 3D graphics. For this reason, the tool is built as a fully automated and immersive 3D environment directly into the Eclipse IDE. Many features that specifically aid different user tasks are implemented such as artefact search, animated transition to targets and in-situ documentation. User disorientation is a major issue in 3D environments so special attention is also paid to user interactivity and navigation enhancement. The tool supports a rich user experience through the use of multiple native GUIs right within the virtual 3D environment, significantly enhancing the immersive nature. It is anticipated that with the various task-oriented features made available, the tool can further contribute in facilitating the adoption of 3D software visualisation in the industry as a respected and practically useful technology.

To summarise the research objectives, this work introduces a new contextual visualisation approach for synchronising the **product** (the software artefacts) with its original development **processes** (represented by Scrum artefacts and activities) and then presents this approach in an immersive and interactive visual representation that has the potential to inform several software tasks. The approach contextually highlights human activities and reveals their impact on individual software artefacts.

1.4 Scope of Research

It has been stated above that research in software visualisation is primarily undertaken to visualise three different aspects of software; software static structure, software evolution, and runtime behaviour. The work reported in this thesis is specifically focused on visualising the *static structure of software systems* while also exploring other techniques to extend this form of visualisation to accommodate other aspects of software development. In other words, this work is primarily concerned with investigating other possible application contexts of software structure visualisation. To achieve this, a new technique has been devised to augment the city metaphor approach with information regarding software development processes (Scrum artefacts and activities) and then a tool developed that implements the new visualisation technique. While the Scrum process presents a promising opportunity for synchronising the product with its original process (and hence realising the Conceptual Visualisation notion), it is acknowledged nonetheless that the choice of Scrum as a specific process represents an inherent limitation on the scope of this research.

Lastly, as with all academic research endeavours, such an undertaking is constrained by various restricting and limiting factors, the most prominent of which is the course of time. This work has involved the exploration and evaluation of existing 3D software visualisation metaphors and the 3D graphics libraries available in order to choose those that best match the needs of this work. This has been essential but costly in terms of research time. For that reason, undertaking a form of empirical evaluation to soundly support and validate this research has not been feasible. Nonetheless, to demonstrate the effectiveness of the devised technique and its success in addressing the research objectives, several case studies have been carried out where the tool is used to visualise several open source software systems of different sizes. The different task-oriented features of the tool and its potential capabilities for supporting various software tasks are also demonstrated. This material is presented in Chapter 5.

1.5 Research Methodology

This research is primarily concerned with the *development* of a new technique to address the advancement, growth, and improvement of a relatively novice discipline in software engineering. The technique is practically demonstrated by implementing it in a newly developed tool. This form of research is commonly known as Design Science, which can be simply defined as the practice of undertaking research with the aim of developing a solution that addresses a particular need or problem (Hevner, March, Park, & Ram, 2004). The design science research methodology aligns very well with the direction and intent of this research and hence it is adopted here. This research approach is also referred to as employing a *Constructivist Methodology* in other philosophical schools of thoughts. Chapter 3 of this thesis elaborates further on this topic.

1.6 Structure of the Thesis

This chapter has presented software visualisation as a discipline, pinpointing particularly important but less explored and investigated areas, and has described how this research is intended to contribute to the discipline by addressing those specific areas. Chapter 2 of this thesis introduces a more focused and elaborated literature review of past SV research, paying particular attention to previous related works. Chapter 3 describes in detail the approach followed in this research for achieving its objectives in light of the methodology adopted. The selected tools and technologies are also introduced in this chapter. Details of the design and architecture of the developed technique and system, along with discussion of the implementation process, are presented in Chapter 4. In Chapter 5 some evaluation criteria drawn from the literature are introduced and then a discussion of the laboratory evaluations based on those criteria is presented. Chapter 6 concludes this research presenting a summary and highlighting its major contributions, as well as its limitations, and then ends with recommendations for desired future improvements and research paths.

2

Literature Review

This chapter presents a focused and concise summary of software visualisation research. Prior research that is particularly relevant to this work is discussed focusing primarily on three areas of interest: software structure visualisation, 3D software visualisation techniques, and recent emerging issues in software visualisation research. Prominent and influential tools/techniques in each respect are highlighted in the process. Most importantly, the inadequacy of current visualisation techniques in presenting the user with several important aspects of the software engineering discipline is discussed, revealing thereupon the point and context from which this work specifically emanates.

2.1 Software Visualisation

The introductory chapter has provided a brief background on Software Visualisation (SV) as a research discipline explaining its core concepts and assumptions, its fundamental motives, and why it is deemed valuable for various software engineering (SE) tasks. This section presents a more formal and detailed introduction to the domain of Software Visualisation, shedding light on its early beginnings and how it came to be a recognised and respected field of research in the SE discipline.

Definition. Software Visualisation has been defined in the literature as:

> "*… the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software*".

This is the definition provided by Price et al. in 1994. While aspects of this definition remain accurate, the field has since grown significantly, making this definition somewhat obsolete according to some researchers. Knight and Munro in 1999 put forward a similar definition but added to it the emphasis on '*reduction of complexity*' as a fundamental characteristic of the field. Michele Lanza in a 2009 conference on software visualisation reduced the Price et al. definition to simply "*The use of computer graphics to facilitate the understanding of software*", which more succinctly conveys the essence of the field as it is currently practised.

Classification. Although a range of definitions such as those just described exist, current software visualisation research is also commonly defined by how it is used. Use is often classified under three primary categories (with respect to the *aspect of software* they address) which are: visualisation of software *static structure (including lower source code and higher architectural levels)*, visualisation of *runtime behaviour (also called Trace Visualisation)*, and visualisation of *software evolution* (Diehl, 2007b). This widely used categorisation shapes almost all research in the field and might have even unintentionally played a role in limiting its growth and its slow-paced embrace of other aspects of software. As was introduced in Chapter 1, software visualisation is essentially concerned with supporting user comprehension, and thus the above categorisation implies that research in each category should address the comprehension of those three aspects of software. However, the software industry and software engineering disciplines constitute several other aspects that are likely to

benefit from software visualisation, such as software management, software development processes, and software maintenance. As is revealed in the following sections, it is only recently that researchers have recognised the need, and the potential, to utilise contemporary visualisation technologies to aid comprehension and awareness of these and other aspects of software engineering practice.

Comprehension. The case for software visualisation and its role in supporting cognition and comprehension, amplifying knowledge and intelligence, and increasing awareness in the domain of software engineering, has already been introduced in Chapter 1. The potential and value of software visualisation as a research discipline is in fact very much established in literature and has its roots in older disciplines, particularly in the information visualisation domain. Some prominent research works that deal with software visualisation's role in supporting comprehension include (Bassil & Keller, 2001; Boccuzzo & Gall, 2009a; Brooks, 1983; Carneiro, Magnavita, & Mendonça, 2008; Eichberg, Haupt, Mezini, & Schäfer, 2005; Knight & Munro, 1999, 2002; Knight, 1999; Kot, Grundy, & Hosking, 2005; Lemieux & Salois, 2006; Pacione, 2004; Panas, Epperly, Quinlan, Sæbjørnsen, & Vuduc, 2007; Von Mayrhauser & Vans, 1995; Wettel & Lanza, 2007a; Young & Munro, 1998). Moreover, some recent empirical studies that further attest for SV in aiding software comprehension have already been mentioned (in Chapter 1). The essential concepts underpinning those studies revolve around the fact that software systems often comprise a large number of complex, interrelated artefacts, that those artefacts are intangible, and that they generally involve relatively large numbers of people working on them. With these concepts in mind, building a visualisation of a system and augmenting it with the information of interest is then conjectured to make it significantly easier and more effective for stakeholders to construct a mental picture of the system and to gain awareness of different aspects of it, such as the scale and state of its components, their inter-relationships, and who is/are involved in their development.

Software visualisation also draws significant support from existing cognitive theories and from cognitive models that were introduced by researchers studying comprehension in software engineering (Bacim et al., 2010; Cockburn, 2004; Petre, Blackwell, & Green, 1998; Sulaiman, Idris, & Sahibuddin, 2005; Tudoreanu, 2003; Xu,

Chen, & Liu, 2009). One of the most important (and highly cited) works investigating the cognitive aspects involved in the software comprehension process is the 1999 paper by Storey, Fracchia, & Müller titled "*Cognitive design elements to support the construction of a mental model during software exploration*" where they studied different strategies and cognitive models involved in program understanding as undertaken by different stakeholders performing various tasks. Their paper is noted for promoting the top-down comprehension model as being particularly suitable for software visualisation techniques. Tudoreanu (2003) has also studied how visualisation tools can reach the goal of reducing cognitive effort by focusing on maintaining '*cognitive economy*'. He suggested that to reduce cognitive effort, a tool needs to reduce the overall amount of information handled by users while at the same time maximising those specific elements of information that are directly related to the problem at hand, implying that to be truly effective, a visualisation tool needs to be specifically customised for each category of task it intends to support.

SV Literature. The software visualisation body of literature is abundant with a wide spectrum of research works that have noticeably flourished after the field gained momentum with the inauguration of its first international Dagstuhl (one day symposium) seminar in 2001. The very first work on software visualisation can, however, be traced back to the mid-1980s with the appearance of probably the earliest visualisation tool called Rigi in 1986 (Eichberg et al., 2005) that provided visualisation of high level software structure (subsystems and modules). Since then, a plethora of visualisation techniques and tools have appeared, with several other review papers that have sought to provide classifications, taxonomies, and surveys of those studies. During the early stages of the field's establishment the majority of studies were focused on '*algorithms' runtime visualisation and animation'* as can be seen from one of the earliest survey studies conducted by Ellershaw and Oudshoorn in 1994. That said, some researchers tend to classify algorithm visualisation as a separate field, referring to it instead as algorithm simulation.

More directly relevant to the research reported here are the surveys and literature studies focused on the three categories of software visualisation mentioned above. Some of those studies were comprehensive general surveys such as (Gračanin et al.,

2005; Lemieux & Salois, 2006; Sensalire et al., 2009; Sensalire & Ogao, 2007a), some focused specifically on techniques and tools that visualise software *static structure* (Caserta & Zendra, 2010; Ghanam & Carpendale, 2008; Sharafi, 2011), others focused exclusively on 3D techniques (Rilling & Mudur, 2005; Teyseyre & Campo, 2009), while others have conducted survey studies to primarily investigate issues in existing software visualisations and to help guide future researchers. Among the most prominent works falling into that last category are the series of studies compiled by Sensalire et al. (Sensalire et al., 2008, 2009; Sensalire & Ogao, 2007a, 2007b) and the Petre & de Quincey (2006) paper titled "*A gentle overview of software visualisation*". In that same theme Kienle and Muller, in their 2007 paper titled "*Requirements of Software Visualization Tools: A Literature Survey*", reported several quality attributes and functional requirements that are presumed to make a visualisation tool more effective, based on a comprehensive literature survey. Gallagher, Hatch, & Munro (2008) have also proposed an evaluation framework to help researchers assess the effectiveness of software *architecture* visualisation tools in particular, and they have highlighted some of the most desired requirements in that regard. Given the existence of this number of diverse literature studies, it is largely redundant to present here a broad overview of past work in this field. Moreover, taking into consideration the fact that this work is focused on ***3D*** visualisations of software ***structure***, the following section is hence purposefully limited to presenting and exploring a selected number of past related studies that either involve *3D* visualisation techniques, or that specifically visualise the software *structure*, or that address a combination of both.

It is relevant also to highlight here that the last two survey studies just mentioned, namely the Sensalire et al. series and the Petre and de Quincey (2006) paper, were particularly valuable and indispensable for this work as they inspired different aspects of it. The relevance of the Petre and de Quincey work has already been discussed in the introductory chapter, while that of Sensalire et al. features prominently in Chapter 5 when discussing the '*desired features*' for software visualisation tools.

2.2 Related Work in Software Visualisation

This section provides an overview and brief description of some past research works that are deemed most relevant to this research. Where appropriate, newly introduced visualisation metaphors or techniques that are of particular interest are highlighted.

SeeSoft and SHriMP are widely held to be the earliest tools to have introduced visualisation of the static structure of software systems, and so are among the most frequently discussed works in the SV literature. SeeSoft was introduced in 1992 by Eick et al. and its visualisation technique was based on mapping a system's source code lines to 2D coloured pixels. It gained a high profile in the 1990's after the authors reported its successful application at Bell Laboratories "*on a software [sic] containing millions of lines of code and developed by thousands of software developers*" (Caserta & Zendra, 2010). SHriMP was introduced in 1995 by Storey et al. and it primarily provided a hierarchical Treemap-based 2D visualisation of software systems at high levels (packages, classes, and so on), but also incorporated several other different views including one at source code level. SHriMP (which is still being maintained and supported today) is best known for being a well-supported visualisation tool that incorporated many core functionalities including versatile zooming approaches and animated transition, both of which were later attributed by other researchers to be highly valuable to users from a cognitive perspective.

In 1997, Andrews et al. introduced a 3D visualisation technique called Information Pyramids (Figure 2.1) that used nested 3D cuboids to visualise hierarchical structures. The authors applied the technique to visualise file systems (directories and documents). While this approach belongs to the *Information* Visualisation domain rather than *Software* visualisation, it has however inspired some later metaphors of software visualisation work.

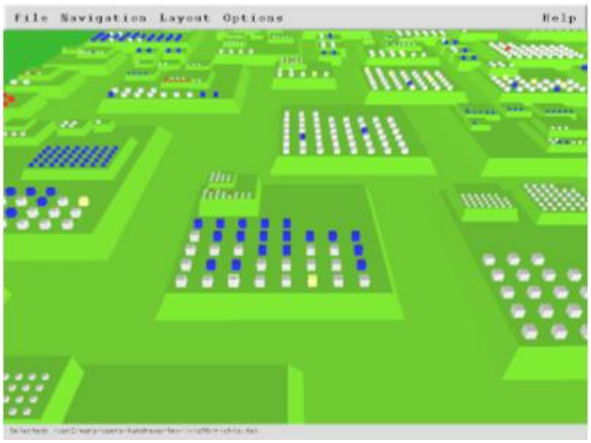


Figure 2.1: A view from Information Pyramids, from (Andrews et al., 1997)

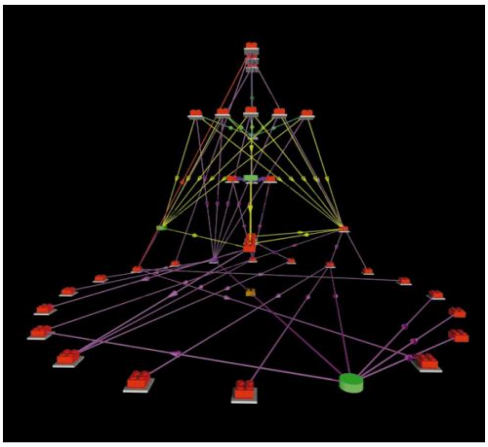


Figure 2.2: Lego Bricks and Geons-based early software structure visualisation, from (Feijs & De Jong, 1998)

A year later, Feijs & De Jong used the VRML language (now replaced by its successor X3D) to introduce a simplistic form of 3D visualisation of software architecture based on what was to be later known as Geons (3D primitive shapes) and Lego Bricks. Being based on VRML, the created 3D worlds were viewable in web browsers and allowed for basic navigation and user interactions (see Figure 2.2). In 1999, Churcher et al. also used the 3D VRML language in experimental studies to implement different 3D visualisation techniques such as cone trees, treemaps, and forests (see Figure 2.3) that primarily visualised the inheritance structures in systems.

Figure 2.3: 3D treemaps (left) and 3D cone trees (right), from (Churcher et al., 1999)

Knight & Munro (2000) in a paper titled "*Virtual but Visible Software*" introduced the first full-scale virtual reality environment for visualising software. Inspired by earlier researchers that highlighted the benefits of virtual reality worlds in exploiting the natural perceptual skills of humans (such as navigation, orientation, and sub-conscious filtering) to comprehend vast amounts of information, they developed a *Software World* visualisation technique to visualise the structure of Java source code using a *real world* metaphor (see Figure 2.4). In their visual metaphor, they mapped source files to cities, classes to districts within cities, and methods as individual buildings. The

buildings' height, colour, and the number of doors they had, were further mapped to different metrics (lines of code (LOC), access modifiers, and number of parameters, respectively). An immediately notable issue associated with this approach is that by depicting each source file as a separate city and individual methods as buildings, the visualisation will hardly be capable of accommodating large-scale systems and will have scalability issues in both graphic cost and view complexity. Knight et al. in following studies explored several aspects of software visualisation, focusing particularly on virtual reality worlds and their advantages over simple 3D shapes, and have investigated issues such as the aspects of software that can take advantage of this technology (or as they put it, '*are acquiescent to visualisation*'), the nature of the data that can be visualised, and the scope and nature of tasks that can be supported (Knight & Munro, 2000a, 2000c, 2001). In 2002, joined by Charters and Thomas, they extended their Software World to visualise software structure at the component (module) level, which they named *Component City* (Charters, Knight, Thomas, & Munro, 2002).

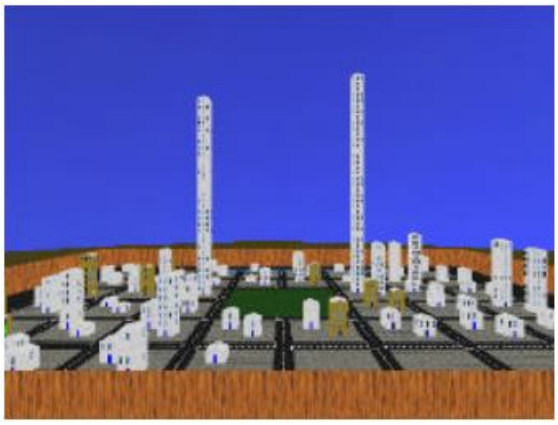


Figure 2.4: Software World, from (Claire Knight & Munro, 2000a)

Figure 2.5: Top (a): Original proposed approach of Panas et al. (2003), from (T. Panas et al., 2003). Bottom (b): A view of the 2007 implementation, from (Panas, Epperly, Quinlan, Saebjornsen, et al., 2007)

In 2003, Panas et al., motivated by the concept that realism better exploits humans' natural and intuitive interpretation skills, took real world metaphors to the extreme

and proposed a highly realistic visualisation approach based on a 3D *Cities Metaphor* that incorporated trees, street lamps, moving cars, and a variety of buildings, all realistically textured (see Figure 2.5a). The approach was intended primarily to inform project decision makers by visualising software systems at the production stage in order to visually highlight cost-related information and issues. Even though the authors originally reported their metaphor as '*3D City Metaphor*', it is described here as '*Cities Metaphor*' to highlight the fact that their metaphor mapped each package to a separate city (hence creating *multiple cities*) and classes were then mapped to individual buildings on top of those cities – this classification strategy was adopted originally by Caserta & Zendra in their 2010 survey study. Two years later, the same authors developed a versatile visualisation framework called *Vizz3D* that allowed users to create different visualisation views by configuring the model-to-view and view-to-scene mappings instead of hardcoding them (Panas, Lincke, & Löwe, 2005). In 2007, the authors used Vizz3D to build a single-view visualisation technique (Figure 2.5b) that incorporated many ideas from their original 2003 proposition (Panas, Epperly, Quinlan, Saebjornsen, & Vuduc, 2007).

In a further effort to utilise real world metaphors, Graham et al. (2003) proposed a 3D visualisation approach named *Solar System* (see Figure 2.6) to visualise software structure (at package and class levels) augmented with software metrics, and implemented a prototype tool to demonstrate the concept (Graham, Yang, & Berrigan, 2004; Yang & Graham, 2003).

Bonyuet et al. (2004) have also experimented with new 3D technologies to help push the interactivity level in software visualisations to even greater realism (see Figure 2.7). They utilised a 4-wall digital CAVE© display, a virtual reality magic wand, and a pair of LCD glasses to allow users to literally 'inhabit' their 3D software world, named *Code Mapping*. However, the visualisation technique itself was very simplistic and was based on connected geons. This type of visualisation is referred to as graph-based, in relation to graph theory, and is known in SV literature to be impractical at scale, leading to extremely cluttered views (Ghanam & Carpendale, 2008; Marcus, Feng, & Maletic, 2003).

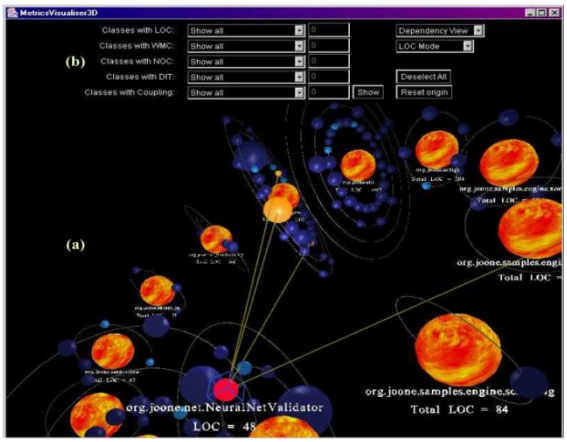


Figure 2.6: Solar System metaphor, from (Yang & Graham, 2003)

Figure 2.7: A view from Code Mapping, from (Bonyuet et al., 2004)

In 2004, Balzer et al. presented yet another 3D visualisation technique called Software Landscapes aimed at visualising the static structure of large-scale object-oriented software systems. In their metaphor, they used 3D spheres to represent packages with the spheres being nested inside each other to depict the hierarchical structure of packages. Each sphere can have a platform with 2D circles on its surface representing classes, and in each circle, 3D coloured cuboids were used to represent both methods and attributes (see Figure 2.8). One key contribution of their work, however, is the introduction of the *Hierarchical Net* approach to route the connections between different artefacts through their parents, eliminating the elusive problem of cluttered-views associated with representing interrelations in many other software visualisation techniques. Their approach also involved an automatically-adjusted transparency approach to reduce visual complexity such that as users approached a sphere, it became more and more transparent until the sphere's surface was completely invisible. On the other hand, the approach makes it very difficult, if not impossible, for the user to have a global overview of the whole system structure. Users cannot see the contents of a package, including the classes, until they approach it, hence losing the ability to see the entire structure at once.

In 2007, Wettel and Lanza introduced a new City Metaphor called CodeCity that has attracted extensive attention from the SV community and according to (Lanza, Gall, & Dugerdil, 2009) had '*a remarkable impact in terms of scientific publications*'. Wettel (as part of his PhD thesis) implemented the visualisation technique in a tool with the same name and that was reported to have been downloaded more than 1400 times in less than a year.

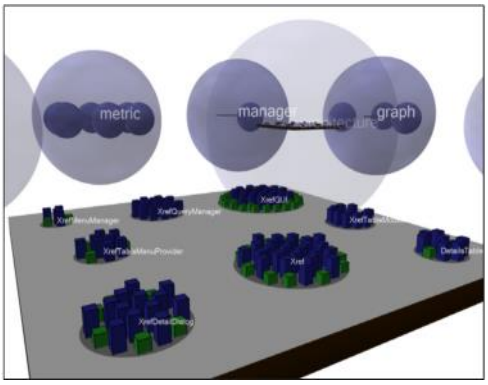


Figure 2.8: Software landscape, from (Balzer et al., 2004)

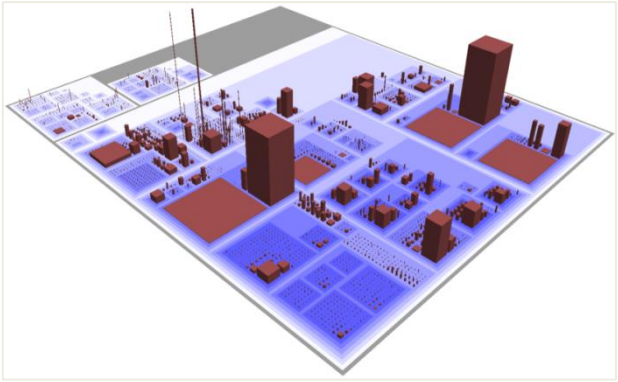


Figure 2.9: A view from CodeCity, from (Wettel & Lanza, 2007a)

CodeCity was promoted as supporting software *habitability* through a well-designed version of the city metaphor that corresponded better to reality, allowing viewers to feel '*at home*' when navigating and exploring the virtual city, hence maximising the exploitation of humans' orientation and perceptual skills to support comprehension of software (see Figure 2.9). Wettel further developed the tool to support identification of specific design problems, and to support different software evolution views. Amongst all the 3D metaphors studied in this research, CodeCity was found to possess simplicity and good navigability experience, and displays the most natural-looking structure when it comes to its layout technique. It was also empirically evaluated and reported as significantly supporting user comprehension of software structure, as was noted above (see Chapter 1, section 1.3). The research work reported here is based on a slightly modified version of CodeCity, as was disclosed in Chapter 1, and hence more details and discussion of this visualisation approach are provided later in this work. A simplified version of this metaphor was later integrated into the Eclipse platform as a plug-in called Citylyzer[1] (Biaggi 2008).

CodeCity was in fact part of a Swiss-wide research project aimed at researching and exploring new 3D software visualisation techniques that further included two other separate works. One of those is called CocoViz (Boccuzzo & Gall, 2007a, 2007b, 2009b) and it explored the novel idea of introducing audio into software visualisation (see Figure 2.10). The third tool is named EvoSpaces and was introduced in 2007 by Alam and Dugerdil. EvoSpaces has a dedicated developer and is intended to be a distillate of the successful concepts introduced in both CodeCity and CocoViz. It has extended CodeCity's original city metaphor and so allows users to navigate inside individual

[1] http://www.pedevilla.net/down.php

buildings where they can find *people* working on different levels, in a representation of methods (Figure 2.11). EvoSpaces also supports trace visualisation at runtime.

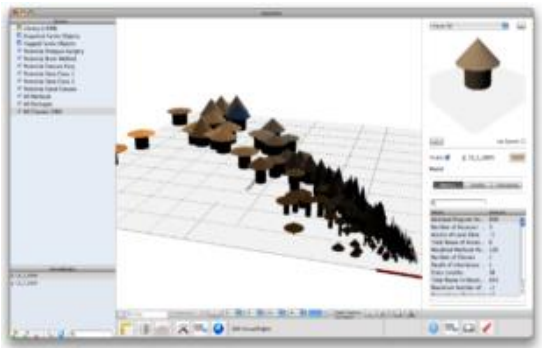


Figure 2.10: CocoViz, from (Lanza et al., 2009)

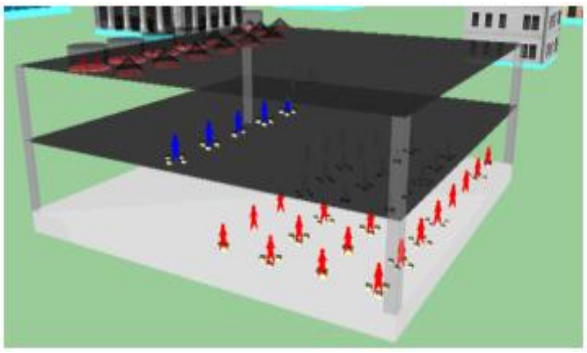


Figure 2.11: A view from EvoSpaces, from (Alam & Dugerdil, 2007b)

Another 3D visualisation approach based on a city metaphor was presented in 2010 by Steinbrückner and Lewerentz who introduced a novel layout technique that not only accounted for the software structure, but also made software development history explicitly visible in the layout. Their layout mechanism was called Evo-Streets and is based on a hierarchically-structured street layout where a certain subsystem (or package) is represented by a street and then contained subsystems or sub-packages subsequently form branching streets. Classes are then represented as buildings attached to their immediate parent package. To represent development history for each class (or module), a terrain is then introduced to elevate each class according to its development version (higher is older) where contour lines are also added to indicate the number of versions for each class (see Figure 2.12). The approach was implemented and demonstrated in a prototype tool that provided different views for system evolution.

In 2011, Francesco Rigotti developed an Eclipse plug-in tool called *Manhattan* that was based on the City Metaphor of Wettel and Lanza and that introduced a new aspect of software to the 3D city metaphor. Utilising another Eclipse plug-in called Syde (Hattori & Lanza, 2010) that extracted and made available software data from a versioning repository, Rigotti added the capability to monitor team activities (mainly commits) projected on top of the visualised city. The approach highlighted modifications made to the system in real time and accounted for deletion, additions, and updates (see Figure 2.13a).

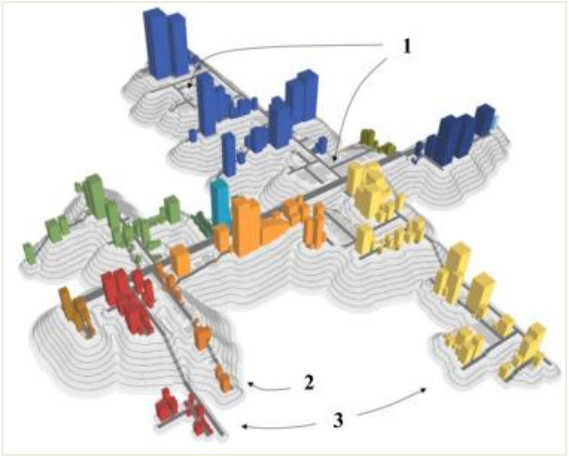
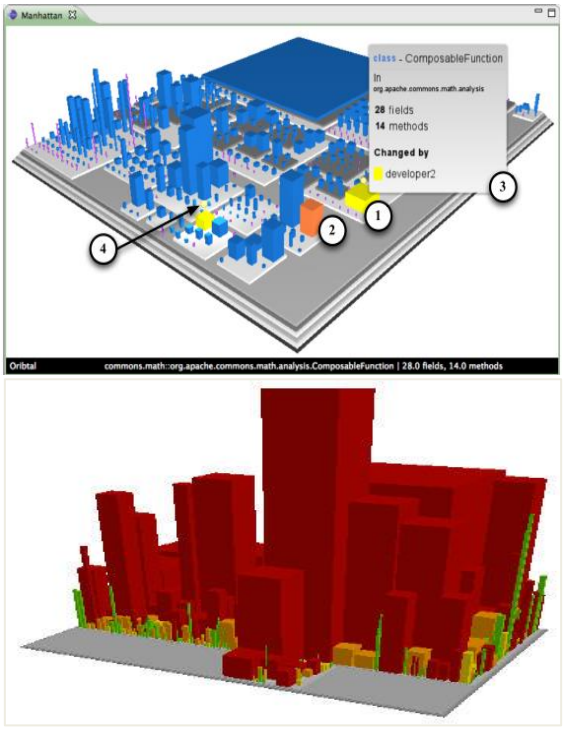


Figure 2.12: Evo-Streets layout with contour-lined elevated terrain, from (Steinbrückner & Lewerentz, 2010)



Figures 2.13: Top (a) a view from Manhattan, adapted from (Bacchelli et al., 2011). Bottom (b) a view from City Model, from the tool's webpage.

To conclude this section, during the course of this research (in what might be the first industrial adoption of 3D software visualisation techniques) a tool named City Model[1] was released in early 2012 as a plug-in to the widely used SONAR code quality management and analysis platform (see figure 2.13b). The tool is based on a city layout that is remarkably similar to Wettel's CodeCity and was developed by a company named *eXcentia*. However, it uses different metaphor mappings than Wettel's approach such that the LOC metric is depicted as the height of a building (in Wettel's CodeCity, buildings are colour-coded to depict LOC range) and the number of methods is represented by the width of a building. While it is particularly significant from a scientific point of view to see this technology being adopted beyond the academic arena and being applied in the real world, the specific metric mappings used in City Model lead to undesirable results. Extremely large buildings can be observed to quickly dominate the city landscape while the rest of buildings become, on the other hand, comparatively very small and jammed between the gigantic buildings, making them hard to distinguishing and interact with. More importantly, mapping the LOC metric to buildings' height was specifically reported by (Kuhn et al., 2010) as

[1] http://qualilogy.com/en/city-model/

significantly confusing and misleading in an experimental study involving experienced developers. The confusion was attributed to the fact that the LOC metric was not always an indicator of centrality or importance.

The above focused review summarises the current scope and breadth of research that has been undertaken in the space of 3D software visualisation and/or visualisation of software static structure. As explained above, the fundamental concept behind all these explored approaches revolves around the notion of using 'spatial' representations to help materialise the virtual components (and aspects) constituting software systems, hence rendering them more comprehensible for humans. The distinguishing characteristic of each approach that could essentially deem it effective or not in achieving that goal relies heavily on the nature of the visualisation metaphor being employed. The SV literature is abundant with studies and guidelines highlighting the characteristics and aspects of metaphors that are presumed more effective than others but experimental validation is nonetheless an essential factor to truly justify the effectiveness of any such metaphor. A notable trend, however, is that both *real-world* as well as *3D* metaphors are found to dominate most recent SV works (Caserta & Zendra, 2010). The research conducted and reported in this thesis is well-informed by many of these prior works as the following chapters reveals (and as reported in Chapter 1) and it directly addresses some of their shortcomings.

2.3 Software Processes and Software Artefacts

Software project management is a highly complex task often accompanied by a high risk of failure. It involves a multitude of inter-related tasks and decision making activities that are critically dependent on obtaining accurate information from various sources, on top of which is the development process. This data is critical for informing the various activities and tasks of project management such as cost management, scheduling and human resource allocation, quality and risk management, performance monitoring, and even for stakeholder communication.

2.3.1 Software Process and Artefact management

To support software managers in their day-to-day work many project management and planning tools have been developed and are considered currently indispensable assets to any mature and well-managed software development organisation. However, most such tools have a common missing link. They provide process information about the product being developed but that information is completely and physically disconnected from that product. Information therefore can get quickly out of sync and outdated as the product evolves or is updated. Moreover, as time passes, and particularly after deployment, tracing the product back to the original process that created it becomes very difficult. In software management this problem is known as '*artefact traceability*' and it has led to the emergence of a new area of research called artefact management. In a recent study on this subject, Fasano and Oliveto (2009) highlighted this issue stating that both project planning tools and Process Support Systems (PSSs) are often missing adequate support for enabling artefact traceability, making management of changes difficult. In their paper, they introduced a novel software management tool named *ADAMS* to enable fine-grained traceability between software artefacts and the software processes that produced them. In their own words (p.146), ADAMS "*enables the definition of a process in terms of the artefacts to be produced and the relations among them, supporting a more agile software process management than activity-based PSSs*". To explain further, their tool allows a manager to link a file (or even entities inside that file) to its/their related processes and team members. The concept behind their tool is based on allowing project managers to use a *product-oriented* work breakdown structure (WBS) augmented with extra process information to define a hierarchy of the software artefacts that would be produced by each team member, and to define any relations and dependencies between them. The emergent model then forms the bases for their tool where each defined WBS entity can be linked to the actual code implementation.

The overall idea behind this work is that, by linking and unifying the software artefacts with their related processes, various kinds of management tasks and activities can be significantly better-informed.

The efforts of Fasano and Oliveto work are well-aligned to the work presented here in terms of objective – which is to synchronise the software product with its processes (down to an individual and fine-grained level) in order to better inform various aspects of software management and software development processes. A key difference between the two is that the work presented here makes this synchronisation process explicitly visible within the context of the software structure by taking advantage of software visualisation technology. During the course of this work different frameworks of software decomposition were in fact considered (as discussed in Chapter 3) including the WBS framework. However, it became evident that use of this framework is declining in present day software development organisations as agile approaches become predominant. Furthermore, the product-oriented WBS is typically defined as a high-level architecture decomposition of a project, that is, in terms of modules and components, never reaching the fine-grained granularity anticipated by the Fasano and Oliveto tool. In this regard, it can be argued that their approach is disconnected from current practice in the software community. For the aforementioned reasons and as is revealed later, a different approach that is more connected to current practice has been adopted in this research for defining and consolidating the product artefacts (code implementations) with their processes.

2.3.2 The Software Development Process in Present SV Research

In addition to the various anticipated benefits of representing the software development process in software visualisation, this research is further motivated by the fact that this aspect of software is not accounted for in existing software visualisation techniques. In fact, many aspects of software are absent or only weakly supported in current software visualisation techniques, and many recent researchers have highlighted the need to address this inadequacy. Storey et al. in their (2005) survey dealing with the support of human awareness in SV tools specifically stated (p.200): "*there does not appear to be a lack of visualization techniques that can be applied to providing activity awareness in software development. What is lacking is how to integrate the various techniques so that they can be effectively used in combination to answer the questions the users will have.*" Petre and de Quincey (2006)

also emphasised this issue, stating that to adequately support a particular application of use in software visualisation one needs to carefully determine and account for the information that will specifically support those tasks – thus highlighting the fact that different tasks require different information. They further discussed four application scenarios and illustrated the possible information that would need to be accounted for by the visualisation technique in order to adequately support each scenario.

Petre and de Quincey (2006) focused particularly on the lack of representation of the "*concepts"* and "*developers' original intentions and rationale*" that together underpin the delivered software products, and highlighted their importance for supporting design reasoning. They specifically stated that, in contrast to other fields of information visualisation, software visualisation has limited itself to the ***artefact*** and has left behind the development ***process*** (where the concepts and rationale are captured). This was discussed in detail in the previous chapter when introducing the notion and importance of *Conceptual Visualisation*.

Storey et al., on the other hand, focused in their study on representing ***human activities*** in order to support *awareness* in software visualisation techniques. They defined awareness as "*an understanding of the activities of others, which provide a context for [one's] own activity*". They explored several existing SV tools in this respect and concluded that only few have offered reasonable support for human activity awareness. To adequately support activity awareness they cited and listed specific questions that a visualisation tool should be able to answer and those questions have been recognised earlier by some researchers as '*important elements*' for supporting awareness. These questions are:

- *Who is or has been working on the artefacts?*
- *Who is the person responsible for or expert in a particular part of the system?*
- *What happened since a developer last worked on the project? (modifications, additions details)*
- *Where did this take place?*
- *When did this happen?*
- *Why were these changes made?*
- *How has a file changed and is there a relationship with other files?*

The authors have finally summarised these questions into four categories; **authorship**, **rationale**, **time**, and **artefacts**, from which users should be able to gain insight when using the tool.

Notably, Petre and de Quincey considered the missing *development process* as the focal point behind promoting awareness and referred to the concerns of Storey et al. as '*subtle'* aspects of awareness that are a by-product of attention to software change. Indeed, as is revealed in the following sections, by attending to the development process and explicitly representing it in the visualisation, almost all of the questions that Storey et al. considered as '*important elements*' of activity awareness become readily available in addition to a more important element which is the 'rationale' or 'original concept' behind each artefact.

There are two important aspects of the Storey et al. study that also need to be highlighted. The first is the ***source*** and ***nature*** of the **data** that the tools they surveyed depended on to present 'awareness' in the visualisation. Their survey included 12 tools, 9 of which extracted their data mainly from version control systems, two by analysing differences between different versions of the system (parsing source code files), and one by capturing information directly from the development environment. Some of the tools used a multiple of those sources. It is evident that the amount and value of information/knowledge that can be extracted from these sources and then visually represented to the user is constrained, especially when compared to what Petre and de Quincey propose in terms of capturing and representing aspects of the entire development processes (conceptual design data) and making the associated data available in the visualisation.

The second aspect that needs to be highlighted is the ***nature*** of the **visualisations** that the tools produced. All the tools surveyed had not considered providing the extracted data within the context of the software structure (two exceptions were a tool called *Creole* and another proposition called *VRCS* (Figure 2.14) but both used a graph-based approach which does not expose well the actual structure of the artefacts). Most of the tools used different graphs, bar charts, pie charts, and text displays to present the extracted data.

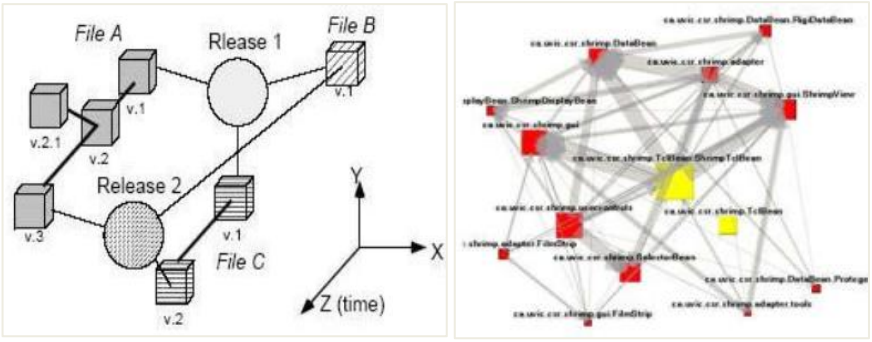


Figure 2.14: A conceptual view from VRCS (left) and a visualisation view from Creole (right), from: (Storey et al., 2005)

The above two highlighted aspects are predominant and characterise almost all existing software visualisation tools that are classified as 'supportive of activity awareness'. Hence, for all these tools, the knowledge and information that they represent is inherently limited to what the version control systems make available, which is generally commit-related information. Those two characterising aspects were found to apply even in recent tools and studies published after 2005 (i.e., after the Storey et al. survey). Examples of those later studies include *StarGate* (Ogawa & Ma, 2008), a chart-based tool (Theron, Gonzalez, & Garcia, 2008), and *Code_Swarm* (Ogawa & Ma, 2009). Figure 2.15 shows views from all three tools. The Manhattan tool, introduced in the previous section, represents a slight exception in that it presents the data within the context of the system structure, but it still relies on the same type of data.

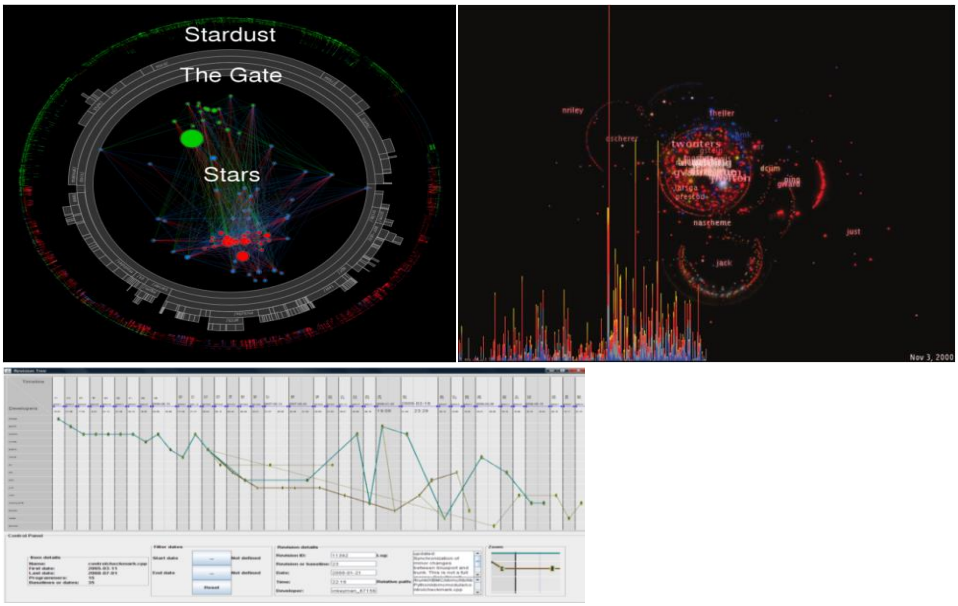


Figure 2.15: Top Left: A view from *StarGate,* from (Ogawa & Ma, 2008). Top Right: a view from Code_Swarm, adapted from (Ogawa & Ma, 2009). Bottom: a view from the chart-based visualisation tool, from (Theron et al., 2008).

2.3.3 The Role of Software Structure Decomposition in the Comprehension Process

The previously stated emphasis on representing relevant and important data within the context of the software structure is purposeful and well-justified. Evidence exists in the literature that strongly indicates that the software structural decomposition is strongly connected to how developers construct understanding and build knowledge about the system they are developing (or maintaining). For instance, in an experimental study that involved a software visualisation approach called *Codemap* (Kuhn et al., 2010), the authors were surprised to find that even though their visualisation technique used a topic-based layout, participants used the tool *"as if its layout were based on package structure—even though they were aware of the underlying topic-based layout",* when solving the various comprehension tasks of the experiment; suggesting that this decomposition structure plays a significant role in the software comprehension process. Developers appear to intuitively (and sub-consciously) build an internal map of the software structure decomposition that is then utilised by the brain to recall and locate an artefact of interest.

Evidence supporting this inference can also be found in earlier studies of software cognition theories and comprehension models. Most of the established cognition models are found to incorporate some form of a mental model that developers construct internally based on the inherent *'decomposition'* of software (examples include the top-down, the bottom-up, and the integrated comprehension models) (Storey et al., 1999). Furthermore, in a highly cited research article on program comprehension where six cognition models were analysed and discussed, the outcome of the study suggested that developers construct understanding by combining and building bigger chunks of code blocks (or artefacts) from smaller ones. The authors further stated that evidence and experimental results seem to go in favour of the top-down approach being the most effective in the comprehension process (Von Mayrhauser & Vans, 1995).

Software systems are inherently structural and as such it is unsurprising that developers tend to intuitively construct knowledge based on the actual structure of the system. This result is in strong alignment with what recent researchers have called 'spatial memory', which refers to how the brain utilises locality to retrieve and recall

information (Bragdon et al., 2010; Cockburn, 2004; Cockburn, 2004; Kot et al., 2005; Kuhn et al., 2010; Marcus et al., 2003; Teyseyre & Campo, 2009). In fact according to many of these research works, one of the motives and justifications behind software visualisation theory (particularly that related to visualising the static structure) is that it provides spatial representations that allow users to readily leverage their natural cognitive abilities.

For this particular reason, the research undertaken and reported in this thesis deliberately takes advantage of this phenomenon and presents the software development process within the context of the software structure. This natural affinity between software decomposition and constructing knowledge is not merely beneficial for people working individually, but it is also important for teams as a whole, aiding them to create a commonly shared mental map or picture of the system that is then used to relate to or discuss various issues about it (Kuhn et al., 2010).

2.3.4 Connecting the Dots

Although Petre and de Quincey have recognised the importance and value of representing the conceptual design in software visualisation, they did not propose or suggest any solution/approach to realise this concept practically. This is the opportunity that forms the core of the research reported in this thesis. Specifically, it utilises the increasingly prominent Scrum agile process as a promising means of introducing the software development process (in terms of original concepts and rationale) to the software visualisation domain, resulting in synchronisation of those processes with their associated product artefacts. As has been discussed in the introductory chapter (and is detailed in the Chapters that follow), various software tasks and activities (including some management tasks) are expected to benefit and be better-informed by this approach. Chapter 1 has revealed how the Scrum agile practice captures all the conceptual design elements that were called for by Petre and de Quincey (essentially capturing all user requirements) and how the Scrum data in itself represents further a valuable source of information that has the potential to inform many software management tasks, including most importantly artefact traceability and management. In this respect, the approach introduced in this work also shares many

commonalities with the Fasano and Oliveto approach (2009) in that it aims to unify the product with its related processes in order to inform several software management tasks. The present work, however, is conjectured to have the advantage of using a popular and commonly practiced approach for collecting artefact definitions and their processes (i.e., Scrum instead of the product-oriented WBS), and then using spatial visualisation (based on actual system decomposition) to represent those artefacts/processes in context. Lastly, as is also revealed in Chapter 5, because of the nature of the Scrum data that is being captured and represented, the approach developed and tested here will enable users to readily answer almost all of the 'awareness' questions highlighted by Storey et al. (2005) as essential elements for supporting activity awareness in SV tools.

2.4 Summary

This chapter has explored a range of prior research works deemed most relevant to the research initiative described in the chapters that follow. Some of the most influential compendia and literature reviews on software visualisation produced by recent researchers were introduced and then a selection of previous studies were discussed in some detail based on the criteria of them either incorporating a software static structure visualisation technique, a 3D visualisation technique, or a combination of both. The chapter then elaborated on three particular issues in contemporary software visualisation research, which are the potential of SV for artefact management and tracing, the lack of support to incorporate the development process (from the conceptual design perspective) in current SV research, and the essential and intrinsic role of software structural decomposition in the comprehension process (hence the importance of utilising it as a means of conveying and communicating other less-spatial aspects of software). Three prominent and particularly relevant past research studies (Fasano & Oliveto, 2009; Petre & de Quincey, 2006; Storey et al., 2005) were examined and discussed in some detail uncovering in the process the context for this research work in relation to past research and the motives behind it. A significant conclusion drawn from this review was that integrating *Scrum processes* (as currently practised in agile development) with *software structure visualisation* represents a

promising opportunity for realising and potentially solving some important issues in software engineering, due to the nature of the Scrum data that captures and accounts for the development process as well as informing the management of those processes. The *richness* of the Scrum data allows for valuable information and knowledge to be presented to software visualisation users thus enabling them (in principle at this stage) to achieve different goals, such as supporting artefact and requirement traceability, providing awareness of human activities, and presenting explicit projections and mappings of developers' conceptual design. The methodology used to develop and deliver this potentially valuable solution is now described in detail in the following Chapter.

3

Research Methodology and Design

This chapter comprises a description of the research methodology and design and reflects on the stated research objectives in that regard. It also sets out the evaluation approach in the light of the adopted research methodology. A brief discussion of common research paradigms in the IT/IS discipline is first presented, based on which the selection of research methodology for this work is highlighted and justified. The way in which the selected research approach serves to inform the different stages of the research project is then explained.

3.1 Research Paradigm

For any research to be robust and so receive appropriate recognition from the scientific community, it must follow a rigorous and well-defined methodology that can assure peer researchers of the validity of its outcomes.

Background

Early research in information science and/or information technology (IT/IS) was the subject of scepticism by members of the scientific community of other disciplines, specifically, by natural scientists. The legitimacy of its research nature as well as the validity of the research methodologies used were questioned (Nunamaker, Chen, & Purdin, 1991). As a reaction to this, in the last two decades the IT/IS discipline has witnessed the emergence of some prominent work that has aimed to address these issues. Some of the most notable and influential of those works are the Nunamaker et al. 1991 paper followed by the work of March and Smith in 1995, and most recently the Hevner et al. work in 2004. All of these papers aimed to make clear the intrinsic difference between the objectives of natural sciences as compared to the objectives of the sciences of the artificial, i.e., design sciences, and hence the legitimate variance of methodologies adopted in both fields.

'Design Science' has therefore become popular terminology to refer to the research approach adopted in many engineering disciplines including IS/IT, as opposed to other natural sciences. Design science research belongs to the philosophical constructivist paradigm and, along with Behavioural Science, constitutes the two most common paradigms of research adopted in IT/IS. Design science is a *developmental* approach that is commonly used for *creating artefacts* that serve a specific purpose or solve particular problems. Behavioural science, on the other hand, is an *evaluative* approach that in the IT/IS domain is typically employed for studying the *impact and effect* of software *artefacts* on users, organizations, and societies (March & Smith, 1995; Nunamaker et al., 1991).

Nunamaker et al. in their seminal 1991 paper introduced a multi-methodological approach to IS research which came to be known as the *systems development* methodology and was originally based on two basic processes, build and evaluate. The

authors further identified five stages to conducting robust systems development research which have been concisely summarised by (March & Smith, 1995) in their following statement:

> *"… Real Problems must be properly conceptualized and represented, appropriate techniques for their solution must be constructed, and solutions must be implemented and evaluated using appropriate criteria*"

More recently, Hevner et al. (2004) expanded on the work of Nunamaker et al. and other earlier works and introduced a more articulated framework for conducting research in the IT/IS discipline, combining elements from design as well as behavioural science. A set of seven guidelines for *conducting* and *evaluating* good design science research were introduced and extensively discussed. Those guidelines align very well with – and seem to have been originally drawn from – the five stages of system development research promoted by Nunamaker et al. Together the work of Nunamaker et al. and Hevner et al. have implicitly or explicitly underpinned research in this discipline. The annual conference specially focused on design science (the DESRIST conference) as well as the existence of special issues of research journals that are specifically dedicated to design science can attest to this.

The introductory chapter of this thesis has indicated the *constructivist* and *design science* nature of the research undertaken here. This work specifically adopts the Nunamaker et al. systems development methodology and carries it out in cognisance of the seven guidelines for conducting proper *design-science* research proposed by Hevner et al. Further justification for this approach, as well as details on how this research applies and adheres to the methodology and guidelines, are discussed in the following sections. Relevant details are also provided and stated in context when discussing the research objectives, design, and evaluation criteria. It should be noted, however, that the guidelines are not necessarily meant to be followed to the letter; as Hevner et al. themselves have specifically put it, "*we advise against mandatory or rote use of the guidelines*". While they contend that each guideline should be addressed in some manner, they advise researchers to use their own judgement for when and where each one is applied and to adapt each as necessary.

3.2 Design Science Research: Key Concepts

To better illuminate the reasons behind the selection of the design science methodology of Nunamaker et al. and the use of the Hevner et al. guidelines, it is necessary to highlight some of the primary concepts that underpinned the introduction of those key works on contemporary IS/IT research methodology. This will also help to bring the subsequent discussion in the next section into context.

It has been indicated in the previous section that there is an intrinsic distinction between natural and design sciences in terms of general research objectives. There are some important concepts in this regard that deserve particular emphasis. There exists a key difference in how progress is achieved in design science research compared to how it is achieved in the natural sciences. In the natural sciences, progress is made by the discovery of new knowledge or propositions of new theories; whereas in design science, progress is made by devising new purposeful artefacts or replacing a technology with a more effective one (March & Smith, 1995). Put another way, natural science strives to explain natural phenomena, hence producing knowledge; whereas design science is concerned with "*devising artifacts to attain goals*" – as aptly described by March and Smith (1995), hence exploiting knowledge to develop technology. Hevner et al. (2004) further highlight that the typical goal of design science is *utility*, whereas *truth* is the typical goal in natural sciences. Reflecting on this, it is easy to relate how the creation of new and innovative artefacts – the products of design science – has extended the boundaries of humans' as well as organizations' problem-solving capabilities.

Systems Development Uncovers New Knowledge

Another important but sometimes overlooked concept is the expectation that the actual process of conducting *design science* or *systems development* may be (and often should be) an important means of generating understanding as well as uncovering new knowledge about the problem at hand. Design science is inherently concerned with problem solving and thus each design science research is a *different* attempt to solve a particular problem (Hevner et al., 2004), or to create things that serves human purposes (March & Smith, 1995). Artefacts produced by those attempts are in fact

each "*an experiment*" that brings new questions to the domain (Newell & Simon (1976) as cited by Hevner et al. (2004)). Thus each of those attempts, assuming their sound design and conduct, should enable researchers to better understand the problem addressed by the designed artefact, much like field studies allow behavioural science researchers to understand organizational phenomena in context. Ensuing studies and independent evaluations of those artefacts can further lead to significant advancement in the field (March & Smith, 1995).

Systems Development as a Research Cycle

Furthermore, any good design science research will typically be founded on a well-studied base of prior research and discipline references and will hence exploit previous results, theories, instruments, and frameworks to solve and address a new or yet unexplored but important problem. Therefore, each sound design science research endeavour will utilise prior knowledge (assuming the needed knowledge exists) to solve particular problems, producing in the process new knowledge that adds to the overall archival knowledge in the domain, and thus contributing to incremental advancement of the field. Markus et al. (2002) remark, however, that in systems development research the requisite knowledge is often non-existent, requiring researchers to employ *creativity* and *trial-and-error* search to reach or obtain the desired results (Markus, Majchrzak, & Gasser, 2002). Emergent knowledge and results of such efforts that are found to have general utility or applicability will eventually find their way to the knowledge base and/or will become best practice.

Design Science vs. Routine Design

This brings to prominence another key point that sets apart *design research* and distinguishes it from *routine design* or *system building*. Hevner et al. stressed that the key distinction between those practices is the identification of a real and explicit novel contribution to the archival knowledge base of the domain. Routine design simply *applies* existing knowledge and best practices to create artefacts to fulfil the needs of organizations or users. Design *science*, however, is driven by the identification of important yet unsolved problems and the desire to present a novel solution, or the desire to solve a solved problem but in a more effective or efficient way. Design science thus *creates* best practices and contributes with new foundations or methods.

It is therefore important to realise that innovation is a key aspect of design science research. Further, design science has the potential to introduce new theories to the discipline. Gregor (2006) and (Wieringa, Daneva, & Condori-Fernandez, 2011; Wieringa, 2010) have particularly explored and studied this issue.

3.3 Revisiting the Research Objectives

In Chapter 1, various aspects of the motivation for this research were introduced and the problem statement exposed the specific direction for the course of this research. This section relates the objectives of this research to the adopted research paradigm taking into consideration the earlier highlighted aspects distinctive to design science research. This is intended to demonstrate how the various elements of this research relate to those aspects.

The primary aim of this research is to develop a software system that utilises 3D software visualisation technology to present aspects of software development processes – *specifically, processes of Scrum practice* – in context with their resultant software artefacts in the same visualisation scene. The tool or system devised serves as a proof of concept for the introduced technique of mapping software processes to their product artefacts in the 3D environment, which (based on prior literature) is anticipated to contribute to the SV body of knowledge. The motivation for this work is based on an identified real need in the domain as addressed in the previous chapters. In particular, the capture and presentation of the conceptual design of software systems along with the lack of attention and support afforded to different aspects of software processes in existing software visualisations were identified and presented with supporting arguments from the domain literature. In section 1.2 it was also emphasised how finding appropriate and effective mappings from different software aspects to graphical representations represents a fundamental challenge to research in SV (Caserta & Zendra, 2010; Gračanin et al., 2005; Teyseyre & Campo, 2009). Lastly, it was stated that the devised visualisation technique has promise in enabling rapid and direct traceability between the original user requirements and the implemented

system components. This research should therefore bring the cognitive advantage of software visualisation technology to a new aspect of software development.

The introduced mapping technique (see Chapter 4) is built on top of the recently proposed yet popular 3D City Metaphor (Wettel & Lanza, 2007b, 2008; Wettel, 2010) which is claimed to be a versatile metaphor that can accommodate numerous aspects of software systems. Hence this work further serves as yet another real-world verification for that claim of versatility.

Finally, there are few secondary goals of this research that are collectively intended to contribute in drawing more attention to software visualisation from industry, and to potentially facilitate its adoption. These include integration with the Eclipse IDE (in the interests of accessibility and availability), offering a useful set of task-oriented features, and facilitating better utilisation of the capabilities that 3D libraries have to offer. Many prior researchers have highlighted the absence of these aspects in existing tools and have called upon future researchers to attend to them, stating their importance in making SV tools more accessible to the SE community. (Further discussion on this is provided in Chapter 5 when introducing the evaluation criteria).

Needless to say, relevant and clear research objectives, and hence a successful research outcome, cannot be achieved without a thorough and complete understanding of the research domain (Nunamaker et al., 1991). As was presented in the previous chapter, this work is well-informed by prior relevant research and is built on top of earlier research findings, their analyses, as well as recommendations of previous researchers, many of which are individually highlighted as appropriate throughout this thesis.

The next section discusses the approach followed in carrying out this research and illustrates how the selected methodology serves to inform the different stages of its execution.

3.4 Research Design

Presenting the details of how research is conducted is essential in helping to convey that the work has sufficient rigor and robustness for the scientific community. It further demonstrates the credibility of the research to other researchers and enables appropriate peer validation of its outcomes.

It was stated previously that this research adopts the classic Nunamaker et al. Systems Development methodology while also taking into account the seven guidelines for conducting high-quality design science research promoted by Hevner et al. Here in this section the details of this research approach are stated and discussed.

The central principles of the Nunamaker et al. methodology revolve around four basic processes: theorising, building, experimenting, and observing, with building depicted as occupying the centre of these processes. The methodology encourages iterative cycles among these four processes where each consumes the outcomes of the others and feeds results back to all, with no strict sequence or starting point. This reflects the degree of freedom required in systems development research due to its explorative nature.

The subsequent analysis serves to highlight how these processes have occurred in this research. The research has been carried out through multiple systematic phases. These are organised and outlined below in a scheme resembling the Nunamaker et al. five stages of research mentioned previously.

3.4.1 Understanding and Defining the Problem Space

This research is founded on the recognition of an opportunity for a novel contribution in the area of software visualisation based on prior knowledge and experience. Extensive exploration of the relevant bodies of literature has been carried out in order to acquire a fuller understanding of the problem and to justify its relevance and the legitimate motives behind it as a gap or need in relation to prior research in the field. The problem addressed by this research has first been noted as an established concept

in the literature, and then described in detail. This has included identifying exactly what contributions are anticipated from this research, how the problems it addresses reflect a legitimate need and interest to the discipline, and how it attends to the recommendations and paths that have been called for by previous researchers. This has demanded careful and systematic investigation and exploration of the relevant literature. Both the introductory and literature review chapters are dedicated to covering and discussing these aspects of the research. They together serve to explain how this research is positioned in relation to relevant prior research in the area of software visualisation.

3.4.2 Building the Conceptual Framework

Since the primary objective of this research is to incorporate aspects of software processes in software visualisation, this has meant a need to gain sufficient insight about the relevant aspects of software management, software development, and software process management. Thus relevant literature in those particular topics has been investigated in order to identify *which* aspects can be supported and *how* they can be supported. It has already been noted that the software development process has largely been overlooked as almost all current SV research addresses the artefact only, with a few exceptions touching on team activities extracted from versioning repositories. Particularly, the Petre and de Quincey (2006) call for Conceptual Visualisation comprising the capture and representation of the originators' intentions, rationale, and activities, has been embraced here as a fundamental goal. For this to be realised, a framework or a scheme is needed that can account for such aspects of software processes and that is structurally suitable for projection onto contemporary software product artefacts. In other words, its structure has to be easily aligned or mapped to the structure of the software artefacts. Only a few such schemes have been identified, notably, the Work Breakdown Structure (WBS) and the Functional Decomposition scheme. Their unsuitability in terms of granularity, however, has been noted, as well as the fact that they are not commonly applied in everyday software development practice. The search effort has eventually led to the domain of Scrum agile development practice. Firstly, the data that are typically captured and found in "user stories" (also called features) account very well for the conceptual design called

for by Petre and de Quincey, and collectively, user stories and data of their enactment truly reflect the actual development process that is to be incorporated in software visualisation. Secondly, in terms of structure, user stories are very well suited for mapping to the software artefacts. Moreover, Scrum is an increasingly popular practice in the software development industry, in daily use by many organisations. Based on this, the Scrum data model has been chosen as a promising scheme for presenting the software development processes *contextually* in software structure visualisation.

To complete the conceptual model of this research's design, a suitable visualisation metaphor is needed such that it can accommodate the presentation of the Scrum data model. Furthermore, an appropriate technique has to be established for correctly mapping the different Scrum data elements onto the graphical and visual metaphors. On top of that, at a lower level, the Scrum data model as represented by releases, sprints, and user stories has to be correctly mapped to a software source code model in a manner that can be efficiently automated. These three elements together represent a crucial step for this work and their achievement has required a considerable amount of effort to develop a well-designed overall model. The intent is not to invent a new metaphor – as many researchers have noted, there is no shortage of metaphors – but rather, to find the best one that can be readily modified or extended to accommodate the new perspective of data to be presented.

Several trials and experiments have been performed in order to choose a final suitable metaphor and design an appropriate mapping of the Scrum data model for it. The basic criteria behind the metaphor and mapping search are: a mapping that is visually non-cluttering, having prior evidence of its cognitive advantage in aiding comprehension, providing simplicity in application, and very importantly, one that, with the Scrum data imposed on it, appears naturally expressive and not overreached or overloaded. The metaphor that has produced the best results during the exploration and mapping experiments is the *City Metaphor*, and specifically, the version introduced by (Wettel & Lanza, 2007b). Details of this chosen metaphor and the devised mapping scheme are disclosed and presented in Chapter 4 (specifically, section 4.3). Some of the other metaphors that have been considered include the Software Landscape and the recently introduced Evo-Streets approach – both were introduced

and described in the previous Chapter. The Software Landscape approach has a much less expressive visual metaphor, does not allow for global overviews of the system structure, and has not been supported with empirical evaluations of its effectiveness in supporting comprehension. The Evo-Streets approach has promise but it also lacks empirical evaluation of its effectiveness, and its layout has been found to result in considerably much larger city landscapes (than Wettel's City Metaphor) that might hinder navigability in the 3D environment (which is a serious issue in 3D SV tools, as reported and discussed throughout this thesis – see, for instance, Chapter 4, section 4.5.9 and Chapter 5, section 5.4.2).

While in the Scrum methodology user stories are classically captured and kept on sticky notes, recently, a few software applications have been introduced to manage the Scrum data electronically while retaining the methodology's essence and revered methods. Examples of such applications include OnTime[1] and ScrumDesk[2]. Since the approach introduced here needs to access the Scrum data in a certain automated mechanism, it has been decided that adopting a data model based on XML is sensible considering that this language is the current *de facto* data format for exchanging information in many software environments. However, even though the overall scheme of the Scrum data model is very much agreed upon in the agile community, still there does not exist a published standardised format for exchanging this data. Hence, a suitable XML schema for exchanging projects' Scrum data has had to be built. Consequently, to be able to use the prototype tool introduced here, one is assumed to have access to a project's Scrum data conforming to the introduced Scrum XML Schema. Again, Chapter 4 discusses all design aspects of the visualisation technique as well as the tool, and presents all the required details and wider contextual information on this matter.

3.4.3 Architecture Designing and System Construction

The next phase of this research project is to design the architecture of the envisioned visualisation technique. In accomplishing this, designs of earlier visualisation

[1] http://www.ontimenow.com/
[2] http://www.scrumdesk.com/

techniques and tools have been examined to gain relevant insights and take advantage of past experiences. Two particular tools have proved to be inspirational to this work, namely, X-Ray (Malnati, 2007) and Manhattan (Rigotti, 2011). The fundamental concept behind visualisation techniques of software structure is based on two key steps. The first is extracting or building a model of the source code of the system. In the case of object oriented development, the model usually reflects the actual hierarchical structure and containment of packages, sub-packages, classes and methods. For completeness it is important that the model makes available all information about individual components, their properties, and relations. In many recent SV tools (e.g., Evo-Streets, CodeCity, and Manhattan), it is typical that an external tool is utilised for building and providing this model (Rigotti, 2011; Steinbrückner & Lewerentz, 2010; Wettel, 2010). The second primary step lies in designing the mapping model to map each artefact of the source code to a corresponding visual metaphorical representation. In the case at hand, the Scrum data model must also be incorporated in this mapping architecture. Since the city metaphor of Wettel & Lanza (2007) is being utilised here, it has been modified and extended to fit it to the purpose of this research and particularly to accommodate the Scrum data model. Details are provided in Chapter 4.

For building and implementing the proof of concept tool, the popular Eclipse IDE has been chosen. This is a deliberate selection, given that one of the research objectives is to address the separation of visualisation tools from development environments which has hindered their practical usefulness as well as accessibility to the community. While this decision might impose some restrictions of capabilities and other limitations, the gained advantages far outweigh this concern as is disclosed later (see Chapter 4, section 4.1). Specifically, the tool is developed as an Eclipse plug-in.

Next, an appropriate 3D graphics library must be chosen. This research started with the intent of using the X3D language (Anslow, Marshall, Noble, & Biddle, 2006) – *an open source ISO standard based on XML* – for reasons of interoperability and the capability of being viewed in web browsers. However, after extensive experimentation and evaluation this option was deemed infeasible for this research. This is briefly discussed in Chapter 4, section 4.4. With the abandoning of X3D, attention has been

focused on full-fledged 3D engines and APIs. Ardor3D and jMonkeyEngine3 have surfaced as the best two candidates and after personal lab evaluation of their features and capabilities, jMonkeyEngine3 has been chosen to implement the prototype tool.

It has been mentioned above that the first step in visualisation is acquiring the architectural model of the source code of a system. While many standalone tools exist for accomplishing this task, in the situation at hand an API or framework is needed that can be integrated into the prototype tool. Since the tool is to be implemented as an Eclipse plug-in, one option is to utilise Eclipse's native APIs for this purpose. Eclipse provides two means of accessing a project's source code model, the JDT's Java Model and the AST's library. However both libraries are considered low level for the purpose of this research. After some search effort, the X-Ray[1] eclipse plug-in has been identified as a potential candidate. However, a very recently developed Eclipse plug-in called Vera[2] has also been encountered and has proved to be very suitable. Not only does it provide the source code model at the appropriate level required, the plug-in is specially created to readily host other software visualisation plug-ins on top of it. Furthermore, it is based on the FAMIX (Tichelaar, Ducasse, Demeyer, & Nierstrasz, 2000) language-independent modelling framework thus it can potentially allow visualisations to be language-independent. Consequently, Vera (Krebs, 2012) has been chosen as a *host* plug-in for the developed visualisation plug-in tool. Chapter 4 presents more details on this matter and how Vera specifically fits in the developed prototype.

To decide on the functionalities and features that should be implemented by the tool prior research has been considered in order to learn from their experiences. It was mentioned above that one of this study's research objectives is to address some of the shortcomings of earlier tools that, according to the literature, have to some extent impeded their adoption. Thus surveys and taxonomies of earlier tools have been carefully examined to identify these shortcomings and the aspects that need more attention. In fact, due to the relative youth of the field, some studies dedicated to

[1] Sincere appreciation goes to Jacopo Malnati, developer of X-Ray for providing advice and help.
[2] Sandro De Zanet, developer of Moose Brewer, has kindly brought attention to the newly published tool, Vera. Deep gratitude goes to Sandro too.

identifying and highlighting the shortcomings of existing tools and techniques and presenting the 'desired features' that future researchers should address (Petre & de Quincey, 2006; Sensalire et al., 2008; Storey et al., 2005) have been especially useful in this regard.

3.4.4 Evaluation and Communication

A crucial component of sound design science research is the appropriate evaluation and validation of the designed artefact. Hevner et al. (2004) emphasise the need to demonstrate the *utility* of the designed artefact as well as its quality and efficacy. The implementation of an instantiation of the artefact – the technique – represents "a proof by construction" (Nunamaker et al., 1991) that serves to demonstrate the feasibility of the devised technique. In addition, *evaluation* of the technique's *utility* is needed to practically demonstrate its success in addressing the stated research objectives. The evaluation criteria must hence specifically address the goals and objectives set above for the research. The main objective of this research is to address the absence of the presentation of software development processes in current SV techniques. Consequently, the first evaluation criterion is oriented to testing whether or not this idea is conceptually possible; in other words, testing its feasibility. This question, as is explained above, is answered by the actual construction of the tool. The other objectives of this research revolve primarily around claims of various potential contexts of use, applications, and real benefits to the SE community. Therefore, this leaves demonstration of the utility of the technique as the main focus of the evaluation process of this work.

Hevner et al. mention five generic categories of appropriate evaluation methods; under which they list *Simulation*, *Functional Testing*, *Scenarios*, as well as *Informed Arguments*. These different types of validation are intended to attest for various properties and aspects of the research artefact; for example, functional testing may attest for quality, informed arguments may attest for and defend the conceptual idea. While functional testing is addressed during artefact development in this thesis, and informed arguments appear in different places in support of the work, formally this research uses simulations and scenarios in the evaluation process. Simulations are

executed using real data being represented by the source code of selected open source projects, in order to lend credibility to the simulation results. The scenarios are intended to demonstrate the utility of the technique via various task-oriented features and potential contexts of use. Chapter 5 of this thesis is dedicated to presenting the details of the evaluation process and reporting the outcomes.

3.5 Summary

This chapter has presented a brief background on research paradigms in the IT/IS disciplines and has then described the adopted research methodology and justified its selection. Some important concepts of the selected methodology were then discussed in order to illustrate how this research and its objectives relate to and satisfy the different elements of the methodology. Lastly, the research design and approach were described in the light of the selected methodology, highlighting how the different stages of conducting this research have been informed by, and adhere to, the methodology.

4

System Design and Development

This chapter presents the details of the architecture, design, and development of the proof-of-concept tool, named *ScrumCity*, which is a major component of this research and is intended to demonstrate the feasibility of the proposed visualisation approach. The novel conceptual visualisation technique is also unveiled and presented in this chapter, along with detailed descriptions of the tool implementing the technique and its features.

4.1 Introduction

Building a prototype artefact is a significant element of design science as has been discussed extensively in Chapter 3. On the one hand, it serves as a basic and preliminary step to demonstrate the feasibility of the research artefact; or as Nunamaker et al. (1991) put it (p.98), it presents "a proof-by-demonstration" (Hevner et al. (2004) refer to it as "proof by construction" (p.14)). On the other hand, the building process itself has the potential to contribute to the body of knowledge in the field through the expected introduction of techniques and methods to solve the particular problems being addressed, as well as playing a role in the progressive advancement of the field. In this regard, this chapter serves to cover the design concepts and implementation details of the developed tool.

Based on the recommendations of several researchers who have carried out taxonomy and survey studies of existing SV work, there is a pressing need to build IDE-integrated visualisation tools as the vast majority of past attempts produced primarily stand-alone systems (Caserta & Zendra, 2010; Ghanam & Carpendale, 2008; Lemieux & Salois, 2006; Maletic, Marcus, & Collard, 2002; Storey et al., 2005; Teyseyre & Campo, 2009). These and other researchers have discussed different incentives and reasons behind this need, and in sum it can be stated that IDE-integrated visualisation tools present greater potential to bring this technology closer to developers and potential stakeholders, thus establishing a better chance to make available its advantages to the field's practitioners. Even though a stand-alone application will usually permit greater functional freedom and higher capabilities in terms of processing power and memory for a 3D application, it has been concluded here, due to the aforementioned reasons, that the advantages of IDE integration outweigh the benefits of developing the tool as a stand-alone application. Hence, ScrumCity has been developed as an Eclipse plug-in, making it easily available to install and explore for potential stakeholders.

However, as is well known, reinventing the wheel goes against advancement. In developing ScrumCity, then, extensive effort has been expended in studying existing and prior work to learn from past attempts as well as to look for suitable tools to utilise or augment. Particularly, as has been indicated above, ScrumCity requires a means of providing a ready and well-structured model of the source code of a project

before it can be visualised. Parsing source code to build and provide such a model is a major work in itself and is beyond the scope of the intended research. The explorative effort has proved particularly fruitful given the recently released tool named Vera (Krebs, 2012) has been found.

ScrumCity has therefore been built on top of Vera which itself is an Eclipse plug-in. The following sections present the architecture, design, and implementation details of ScrumCity and also describe how Vera specifically fits in the design.

4.2 System Architecture

Three primary stages of processing can be identified in almost all software visualisation systems for representing code static structure (Alam & Dugerdil, 2007a; Churcher & Irwin, 2005; Greevy, Lanza, & Wysseier, 2006; Rigotti, 2011), which can be generalised as: source code processing, graphical metaphor and layout processing, and scene rendering. Considering the nature of software static structure visualisation, the three identified stages in fact simply reflect the essential concept behind software visualisation, which is taking as input textual source code and outputting a graphical representation of it that has lower cognitive overhead and is easier to understand and process by the human brain. Source code of the system to be visualised must hence be first processed to extract all required information, which typically results in the production of a model of that system's source code conforming to the requirement of the visualisation system. That model is typically then processed by another module of the visualisation system which generates the mappings from source code entities (product artefacts) and their attributes to abstract graphical representations (spatial metaphors or simply, abstract glyphs). This phase also includes the layout processing that is responsible for giving each abstract glyph its proper place in the visualisation scene. The third and final stage is then concerned with rendering those abstract glyphs into visual representations (turning them into real and visible glyphs) using a graphical rendering library. Whether it is a 2D or 3D visualisation approach, those three main stages (illustrated in Figure 4.1) will always be essential parts of it.

These three general processes of SV systems represent thus the backbone of the generic and shared architecture of software visualisation systems. Needless to say, the actual architecture of any particular system will have its own variations depending on the specific technologies used and the nature of the visualisation technique employed. The specific architecture of ScrumCity is now introduced.

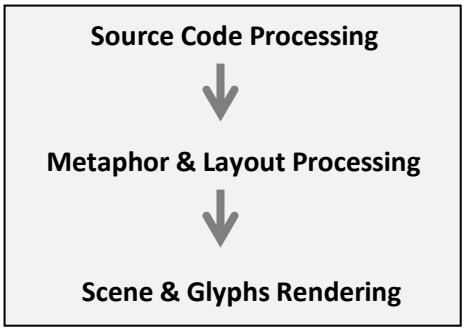


Figure 4.1: The Common Main Processing Stages of SV Systems

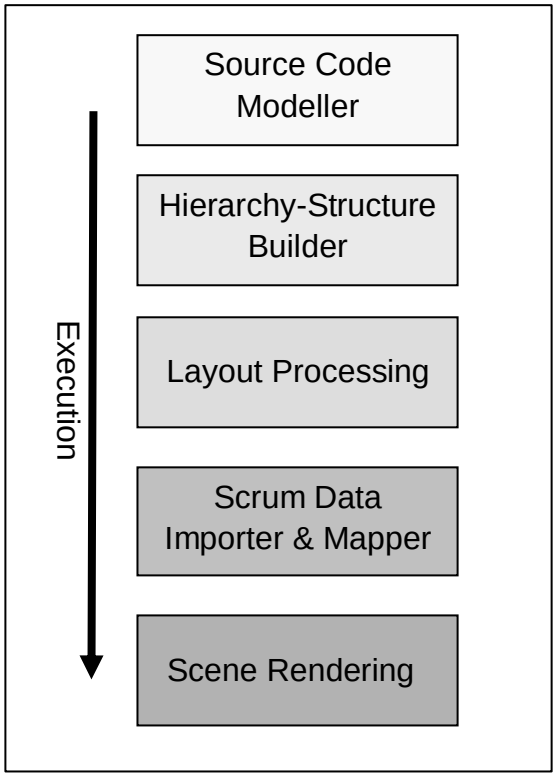


Figure 4.2: Layered Architecture of ScrumCity

Figure 4.2 shows the general architecture of ScrumCity from a process-oriented perspective. It consists of five abstract processing layers that map to real modules of the developed system. In brief, the source code of the system to be visualised is input to the top module and execution continues until 3D scenery is produced and displayed by the lower module. Explanation and brief discussion of each module is presented here and the reader is referred to sections 4.3 and 4.4 for a more elaborated discussion on the inner workings of each module. Since each layer maps to a physical module of the system, they are referred to as processes and modules interchangeably.

It is relevant here to note that as the ScrumCity tool has been developed as an Eclipse plug-in, it has been specifically designed to visualise Java Systems. This also means that

projects to be visualised must be imported into the Eclipse project format so they can be properly opened by the Eclipse IDE. Nevertheless, ScrumCity is merely a proof-of-concept tool that demonstrates the introduced conceptual visualisation technique. Thus, even though Java systems are generally assumed throughout this thesis, generalisation to other object-oriented languages should be relatively straightforward.

4.2.1 Main System Modules (Process-Oriented Perspective)

Source Code Modeller

In this module, the source code of the target system (i.e., the one to be visualised) is parsed and processed in order to build an *object model* containing all the required information about the target system. The object model can be thought of as a logical and abstract decomposition of a system to its constituting components (also referred to as *artefacts* in this thesis). In an object-oriented system those artefacts specifically refer to packages, classes (including abstract classes and interfaces), and methods. Those artefacts collectively represent the low-level architecture of a system (as compared to a top module-level one), which is the level of system visualisation on which this research is focused. This module hence captures the model of a target system into a single 'object model' that is then fed to the next module.

Hierarchy-Structure Builder

The adopted City Metaphor is based on a hierarchically-structured containment approach where child components are nested inside their parent components. The model captured in the previous process does not, however, exhibit this hierarchical and containment structure. Packages, for example, are provided in a flat structure instead of being nested. For this reason, and also due to the nature of the layout algorithm used (discussed in section 4.4), it is necessary to build a secondary model of the system to be visualised where this hierarchical structure is embodied in the inner structure of the model. This module is thus specifically responsible to achieve this requirement by consuming the model of the previous process and producing a new secondary model conforming to this requirement.

Layout Processing

Since the City Metaphor uses glyphs (geometrical shapes) to visually represent the system artefacts, a mechanism is needed to calculate the proper dimensions and position of each glyph. This module hence performs all layout calculations for the entire 'city'; producing specific dimensions and 3D-coordinate values for each glyph. Since this work is based on the city metaphor version of Wettel and Lanza (2007), this module thus uses Wettel's (2010) layout algorithm. More details and description are provided in the implementation section (i.e., section 4.4).

Scrum Data Importer and Mapper

A special 'Scrum' object data model has been specifically designed (introduced in the design section of this chapter) based on observation of common usage of the Scrum methodology in the agile community, since there does not exist a formal scheme for representing Scrum data. In this module, XML files of Scrum data are loaded from a designated file directory, validated, parsed, and the data is then stored into instances of the defined data model. Those data objects are then mapped to the corresponding related objects of the secondary system object model (the one created by the Hierarchy-Structure Builder module). In other words, Scrum data artefacts are mapped to their related system artefacts that are being represented in the secondary object model in a hierarchical structure. Again, details of the mapping technique appear in the design section of this chapter. A similar approach is also used to load and map documentation data to the system artefacts, which is handled by a similar module.

Scene Rendering

This module is responsible for building the scene graph of the virtual 3D environment where each glyph representing a corresponding system artefact is finally visually rendered on the screen. This module comprises a sub-module that uses the 3D graphics library of jME3 to construct the meshes for each glyph (called *Geometry* in jME3) and sets its specific attributes such as colour, dimensions, and location coordinates. It also sets special interactive *controls* for each glyph based on its type.

4.2.2 GUI Module

In line with its visual nature, ScrumCity requires a significant number of GUI components to facilitate proper interaction with the visualisation. So in keeping with the modular approach taken for the system architecture, ScrumCity has a sixth module specifically designated to handle all GUI creation and functionality issues. As the functionality of this module does not relate to the general 'processes' of the system, it is not represented in the main architecture diagram introduced in the beginning of this section. As Figure 4.3 shows, this module consists of two sub-modules, a GUI builder component, and a GUI interaction controller. To help minimise user distraction and lend a greater sense of immersion, all GUI components are specifically designed within the 3D environment allowing users to work with the visualisation in a full-screen-like mode. From a usability point of view, this has a certain implied advantage that is discussed and demonstrated throughout this thesis (particularly, see sections 4.3.2 *(Scrum Presentation Layer)*, 4.4.6 and 4.5.3).

4.2.3 Summary

Now that each module has been briefly presented, the diagram in Figure 4.3 is provided to give an overall picture of the complete architecture of the system, highlighting inputs, outputs, and paths of interaction between the modules.

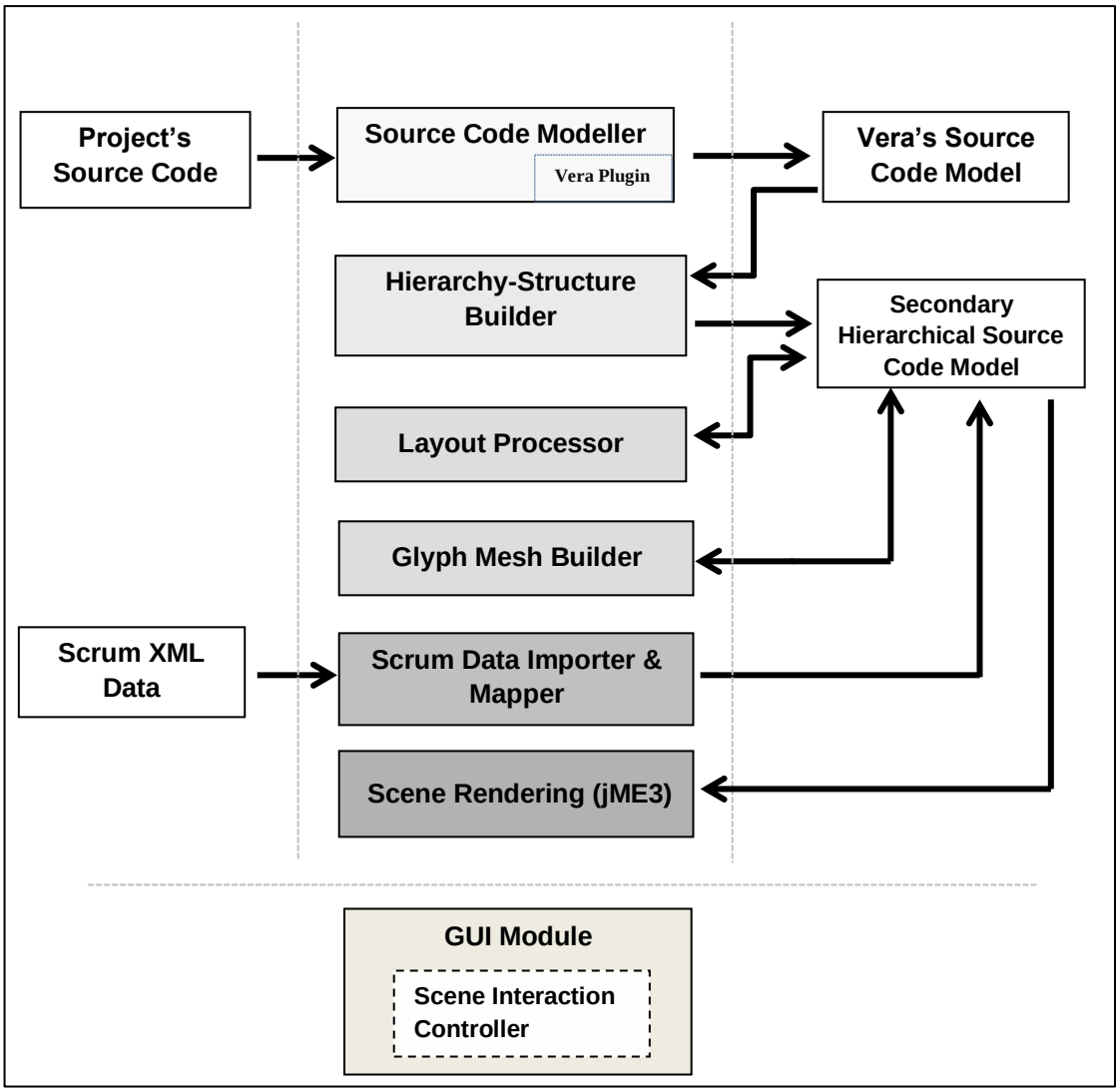


Figure 4.3: ScrumCity's Overall Architecture Model

4.3 System Design

4.3.1 Description of the Visualisation Technique

This section describes the details of the *conceptual visualisation* technique that is the main subject and a primary outcome of this research.

As has been introduced in Chapter 1, the main goal of the conceptual visualisation is to present the software development process (represented in this research by Scrum artefacts and activities) in the context of software structure in a synchronised mechanism. Scrum artefacts are thus to be mapped to their related software artefacts and then presented in the visualisation scene.

The Case for Scrum

To understand this technique it is necessary to elaborate further on how the Scrum methodology is used in contemporary systems development. The intent is not to describe the Scrum practice *per se*, but rather, to highlight how it fits in the introduced visualisation technique.

In the Scrum methodology, system development is carried out through the rapid implementation and delivery of self-contained chunks of user requirements known in the agile community as user stories or features (both terms are used interchangeably in this thesis). This occurs in short repeating cycles known as *Sprints*, where each Sprint contains a pre-determined number of user stories. Each user story normally describes a small functional component of the system (hence the other name, *feature*) that requires a day or less of working effort to develop. A sequence of Sprints represents a Release (normally fewer than 15 Sprints), which is intended to provide a coherent set of working functionality – a deliverable. A complete system is realised and sustained over multiple iterations of Releases.

With the brief Scrum description just provided, it should be evident that user stories are the smallest units of user requirements, whose implementation results in the creation of different system artefacts; which on a similar level of scale, map to classes and methods. Packages are simply logical groupings of classes that do not represent immediate manifestations of user stories. Since a user story by definition captures a

small and specific feature of the system, it is expected to contribute to the system with a set of new classes or methods, or simply with additions to existing classes. In other words, the different system artefacts created are nothing more and nothing less than manifestations of user stories. This is the main concept underpinning the introduced visualisation technique.

QNames. In systems development, each artefact (a method, a class, or a package) has a unique identifier that can be used to refer to that specific artefact. This identifier is commonly called a *QName* and for Java systems, its format is a *de facto* standard amongst all development environments. So taking advantage of this QName, it becomes possible to link and map each user story to the specific method(s) or class(es) that it created, or to which it significantly contributed.

Since each developer is assigned (or selects) a set of user stories to implement, they then know exactly what classes or methods they have created or significantly modified to implement a particular user story. It is thus presumed in this research that each developer plays a key role in realising this mapping. In the traditional manner of Scrum practice where user stories are jotted down on sticky paper notes, this mapping would not be feasible due to the absence of electronic record of the user stories. However, as has been mentioned above, the agile community has recently seen the emergence of several Scrum tools[1] that automate aspects of the Scrum practice. These tools are becoming popular and many high-profile organisations that utilise Scrum practices have migrated to such automated applications to manage their system development[2]. A prominent feature of such tools is that they enable developers to manage their assigned user stories; marking the current status of each one, performing daily updates of remaining work hours, setting a priority level for each story, and various other operations. In such situations it becomes safe and practical to presume such tools have the capability to allow developers to specify the system artefact(s) that were created (or significantly modified) as a direct result of implementing a particular user story. This is simply achieved by having the developer add the QName of the related system artefact(s) as an attribute to that user story. In a real world scenario, a given user story

[1] Examples include OnTime, AgileBuddy, ScrumDesk, and ScrumNinja

[2] www.ontimenow.com claims that more than 10,000 organisations are using their product, including high profile organizations such as NASA, Microsoft, IBM, and others.

will typically manifest to either a single or a very limited number of system artefacts; given the small size and short-time requirement nature of user stories.

While an informal survey of some Scrum tools reveals that such a feature is not currently provided, the advantages and benefits gained by adding such simple data elements to user stories are in fact far-reaching and are of significant value to various stakeholders. One significant advantage would be enabling traceability from original user requirements to actual implemented features, *in a bidirectional manner*. This, along with other implications, is presented in detail in Chapter 5, section 5.4.1.

Work Hours. In Scrum practice, each user story is designated a specific estimate of *work-hours* (also called man-hours or, more appropriately, person-hours), representing the *effort* that is predicted to be required for implementing that user story. During development, it is conventional that each developer makes daily updates of that estimate to reflect the 'remaining' work-hours until it reaches zero, which marks the completion of that user story. The remaining work-hours estimate can decrease or increase according to the actual progress made during implementation. Many current Scrum management tools provide the capability for developers to record such information, which is then used to inform different statistical project data reports and predictions such as burn-down charts – *a key element of the Scrum methodology*.

These important data elements are used in the ScrumCity visualisation technique to present a visual cue for each system artefact – *particularly for classes* – giving users an immediate visual impression of how much work has been completed and how much is left for a particular user story.

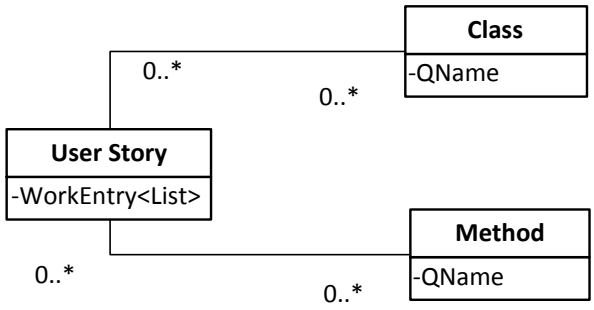


Figure 4.4: Simple UML diagram showing Class and Method Relationships to a User Story

To complete the picture, this research thus assumes that the Scrum data of a system to be visualised is available. Most importantly is that the QNames of system artefacts for each user story have to be available according to the scheme explained above. Figure 4.4 illustrates the conceived relationships between user stories and system artefacts. The next section explains the mechanism used to import the Scrum data.

Scrum Merged with the City Metaphor

Another key aim of the proposed visualisation technique is that the Scrum artefacts are to be presented visually in the context of the software structure. As introduced and discussed above, the City Metaphor has been chosen as a suitable approach among other software static structure visualisation techniques. In the city metaphor, packages, classes, and methods are represented by special glyphs (geometries) in a layout that closely reflects the real structure of software. Furthermore, as also mentioned above, the specific version of the city metaphor that is adopted here has been demonstrated and empirically validated to aid comprehension of, and support learning about, software systems. Cognitive overhead is significantly reduced when exploring a software system in such a visual representation as compared to conventional textual scanning and probing.

Hence, the city metaphor is adopted here to present the software structure and then the Scrum artefacts are mapped and presented in a synchronised manner on top of it. Glyphs of classes and methods are mapped to the specific user stories to which they relate. Users of the visualisation can select a particular user story to see the system artefact(s) that have resulted from its development (or significant modification). In a similar manner, a particular system artefact can be selected so that the user can see the user stories that were involved in its creation. This overall representation is expected to support the user in exploring, inspecting, and reasoning about software systems and places development activities into context. Development processes become seamlessly integrated and unified with the individual components of the product. The next section describes how this mapping is achieved.

Inspired by the work of Petre and de Quincey (2006), the term '***Conceptual Visualisation***' is used to describe the proposed visualisation technique.

4.3.2 Mapping Technique

To accomplish the mapping between the system artefacts and the Scrum artefacts, an object model for each is required. For the *system* artefacts, the model is already provided (albeit it needs some remodelling) by external tools as mentioned above (discussed further in section 4.4), created by parsing a system's source code. For the *Scrum* artefacts, however, an object model has been specially designed. As noted above, there is currently no standard in the agile community for a data model of Scrum. The data elements of Scrum are, however, widely known and represent a conventional *de facto* scheme among agile developers when implementing the Scrum methodology.

For the purpose of this research an object data model for Scrum, as well as an XML schema, have been designed based on that convention. The XML schema is necessary for acquiring and importing the Scrum data of a software project into the visualisation. The data model, on the other hand, is required to properly handle the different Scrum artefacts inside the visualisation.

The Scrum Data Model

The UML diagram in Figure 4.5 describes the newly developed Scrum Data Model. For those with a close affinity to agile methods, the model structure is straightforward. A Release stands as a main object consisting of some properties that include a list of Sprint objects, which each in turn have some conventional properties that include a list of *Features* (user stories). Features in turn have their own properties. The properties that are of particular interest are the lists of Class and Method references, and the lists of Work Entries.

Class and Method Reference lists. Each *user story* object has its own list in which the QNames of system artefacts that are directly related to it are kept. A user story, as has been explained above, can be immediately related to either a class or a method. For simplicity reasons, the model keeps the QNames of each type in a different, separate list. QNames are stored in those lists as simple String objects.

Work Entries lists. Records of the work-hours daily updates for each user story are similarly kept in a list. Each update record has a number of data items that are of interest and thus a special data object has been created to capture those data elements. The *date* and *hours* elements are self-explanatory. With regard to the *QName* element, since a single user story can relate to multiple system artefacts, it is desirable to be able to link each update record of remaining work hours to a specific system artefact (as this enables the implementation of the remaining/completed work depiction that is discussed in section 4.4.4). As for the work entry *type*, since estimates of remaining work hours can decrease as well as increase during the course of development, it is impossible to determine a correct value of how much has been actually done at a given point of time only by inspecting the recorded *remaining* hours. For this reason, it is again postulated that an ability to record the '*completed*' work-hours value is desired, and hence there are the two types of Work Entries. While most surveyed Scrum tools only provide a mechanism to record the estimated *remaining* work-hours, in some organisations employees are asked to record their actual spent hours of work which is then used for determining employee payment as well as billing to the customer. While expecting the availability of the *completed* work hours remains in the realm of theory, its practicality is however not overreached.

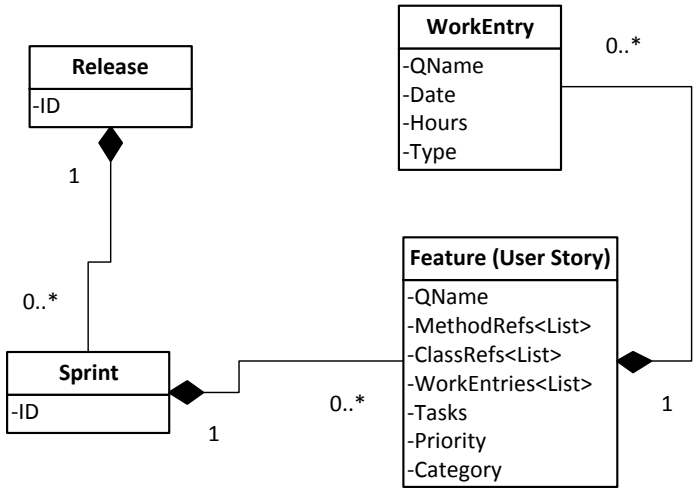


Figure 4.5: UML diagram of the Scrum Data Model (showing partial attributes only)

The Scrum XML Schema

The other component needed to achieve the mapping between Scrum and system artefacts is the XML file representation. XML is a popular and widely used standard of the World Wide Web Consortium for exchanging data. An XML schema for Scrum has been created to reflect the exact data model described above; which, as explained, was conceived based on the conventional application of Scrum practice as found in the agile community. Design of the schema is provided in Figures 4.6 through to 4.9, which are self-explanatory. The complete schema document is provided in Appendix A along with an example of an instance file.

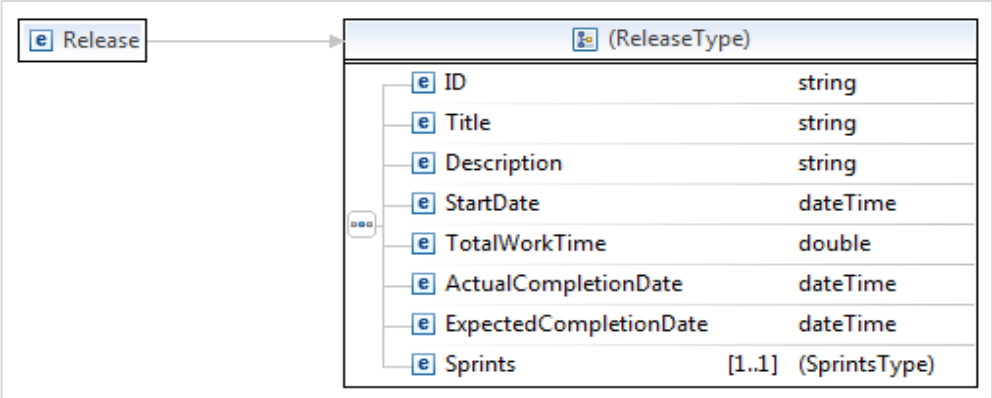


Figure 4.6: Scrum XML Schema Design (Release-Type Details)

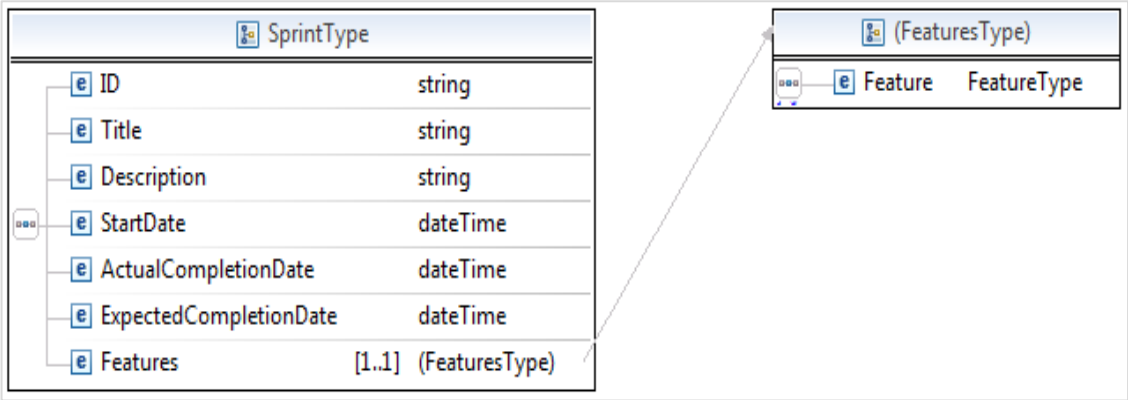


Figure 4.7: Scrum XML Schema Design (Sprint-Type Details)

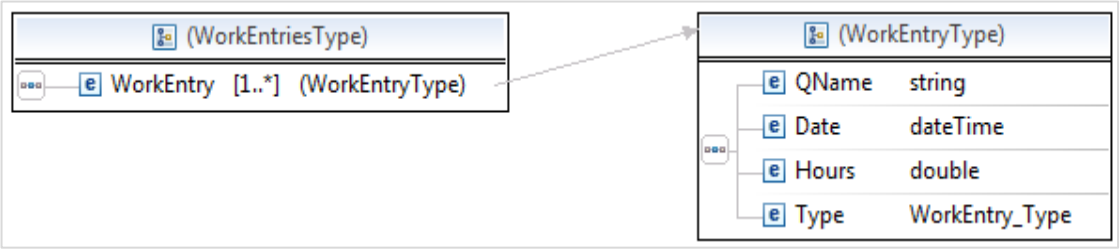


Figure 4.8: Scrum XML Schema Design (WorkEntry-Type Details)

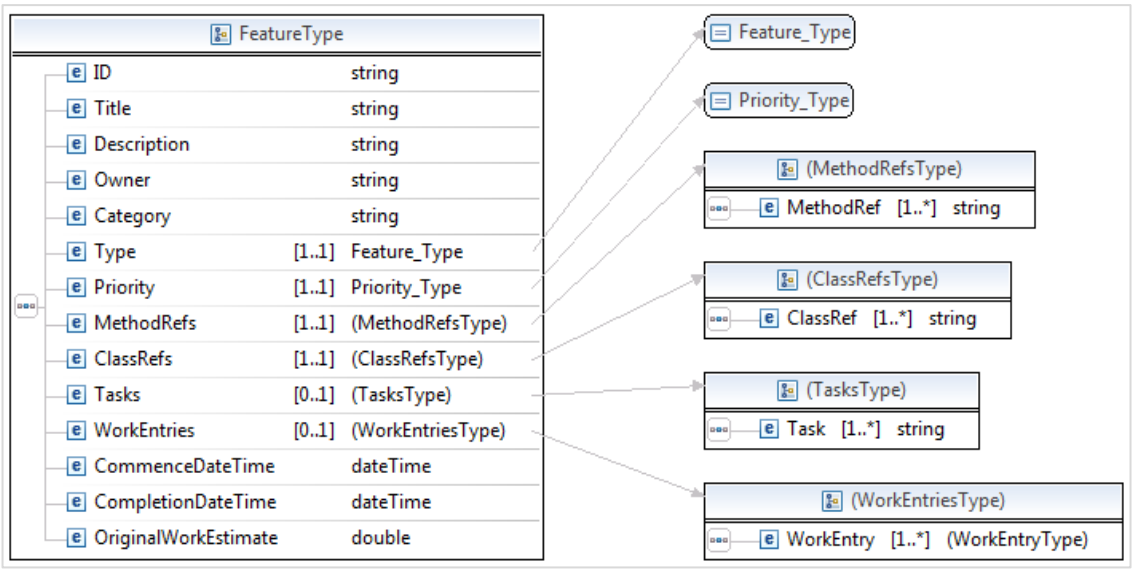


Figure 4.9: Scrum XML Schema Design (Feature-Type Details)

In addition to the Scrum XML Schema, a similar Schema for system artefact documentation has also been designed with the purpose of illustrating the advantage of having *in situ* and in-context ability for inspecting, exploring, and learning about a system. The schema is depicted in Figure 4.10 while some discussion on this is found in Chapter 5 (the complete schema document is provided in Appendix A).

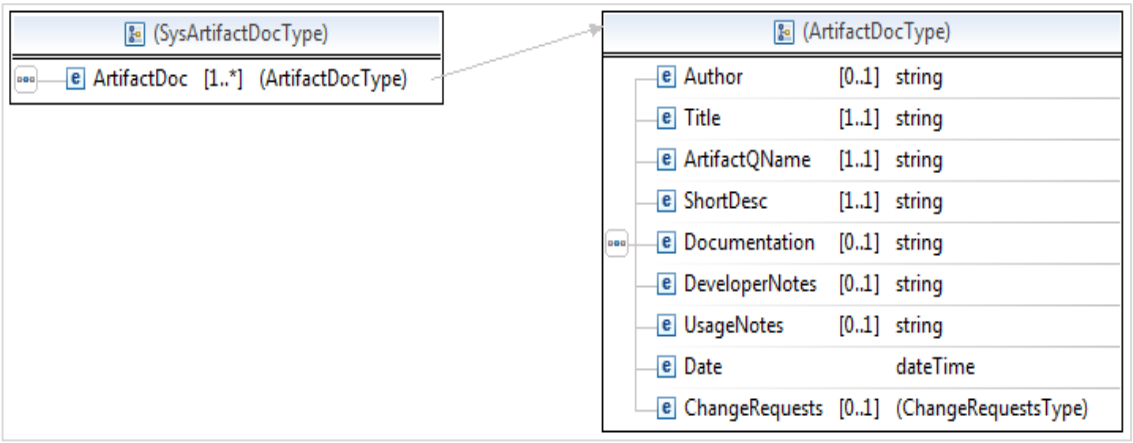


Figure 4.10: XML Schema Design of System Artefact Documentation

Scrum Presentation Layer

The Scrum data model, as presented above, has a hierarchical and tree-like structure. This naturally implies that a graphical user interface in which elements are represented in a tree list would be a good choice to present the Scrum elements to the user. Since the tool being developed in this work is an Eclipse plug-in, a common first choice might also be to use the native Eclipse GUI libraries to build the interface. However, this

means the interface would take a precious amount of space of the available screen area. Given the user-navigable nature of virtual 3D environments, available screen area plays a significant role, and full-screen working mode is commonly preferred. Visualisations of software structure also tend to be of large scale and size because of the typically large number of constituting system components, hence requiring a large display area. (Small and/or simple systems do not need to be visualised in order to be understood.) For this reason and for other usability and aesthetic purposes (discussed in Chapter 5), an *inside-the-3D-scene* interface is preferred.

Hence, to present the Scrum model over the 3D city metaphor representation of software structure, a special non-intrusive graphical user interface has been designed to provide easy interaction right inside the scene graph of the 3D environment. The interface is designed as an overlay layer that can be instantly shown or hidden with a single keyboard press. It presents the Scrum data *in situ* via an integrated non-intrusive mechanism; allowing users to interact with the visualisation without bearing the overhead of switching back and forth between the scene graph window and external GUIs. Even though the tool is an Eclipse plug-in, users can work uninterrupted in full-screen mode to inspect and investigate their system.

Scrum-To-Glyph Synched Mapping. The interface provides a bidirectional mapping between the Scrum artefacts and the city metaphor glyphs (i.e., the system artefacts). A user can select a specific Scrum element from the user interface and the related system artefact glyphs are then highlighted. If the selected Scrum element is a user story, the user is automatically 'transported' using an animated transition to the specific system artefact glyph(s) that this user story has created or modified. Justification for the animated transition can be found in section 4.5.3 (*Scrum-To-Glyph Mapping*). Selecting a Sprint results in all system artefacts involved in that Sprint being highlighted; giving an immediate visual cue as to where in the overall software structure that specific Sprint has contributed.

In a similar manner, when the user selects a particular system artefact glyph (specifically, a method or a class), all of the related user story records in the interface are then selected, guiding the user to the original user stories that were responsible

for creating that artefact. Users can then choose to view the details of each user story, which are displayed in a similar non-cluttering overlay GUI. The user can also interact with each glyph and can choose to read its documentation, view Scrum details related to it, or view the source code. The source code view opens the native Eclipse Java editor window which inevitably interrupts the full-screen mode. Figure 4.11 shows a sample view of ScrumCity visualising itself with some simulated Scrum data being shown.

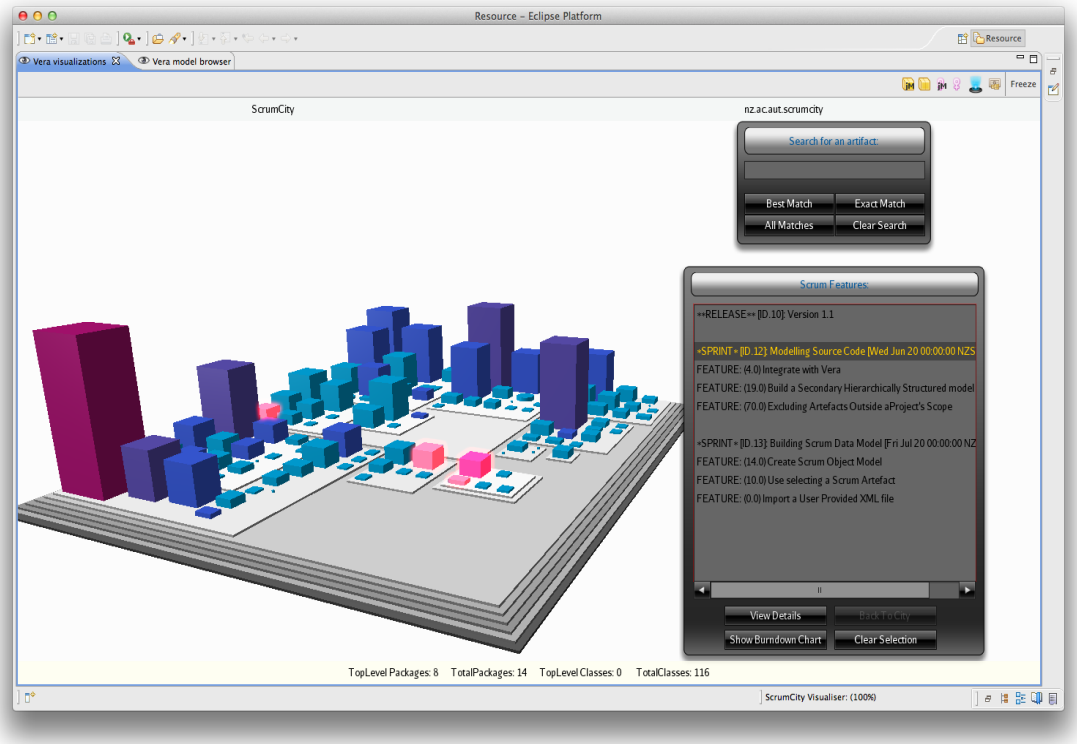


Figure 4.11: ScrumCity visualising itself

4.4 System Implementation

This section sheds some light on important implementation aspects of the system. Whereas the previous two sections were focused primarily on providing a higher level picture that conveys the general outline and main features of the proposed visualisation *technique*, this section discusses details that are more specific to the *development* of the proof-of-concept tool.

4.4.1 Vera

Vera was introduced above as an Eclipse plug-in on top of which ScrumCity was to be built, and which provides the source code model of systems to be visualised. During the course of this research, Vera[1] was released as part of Krebs' Master's thesis. It was immediately found to be particularly suitable for the purposes of this research because, on top of modelling projects' source code, it also provides a ready and convenient special GUI support to host software visualisations. Plug-ins that hook to Vera get their own icon and toolbar command as well as a contextual right-click menu command where that plug-in can be invoked and launched. A shared canvas pane is then used to display the resulting visualisation view after the invoked guest plug-in completes processing. Figures 4.12 and 4.13 show the Vera plug-in running (showing a native example of a visualisation view). The reader is referred to the original thesis for more details about Vera.

Source Code Modelling

In order to visualise the structure of a system, detailed information about each of its individual components needs to be extracted and provided in some accessible manner. Each artefact of the target system, e.g., a package, a class, a method, or an attribute, will have different properties that are important to the visualisation system and must thus be extracted from the original source code. For example, the parent of each package and a list of its sub-packages must be known. For classes and methods, all of the basic and conventional software metrics such as lines of code (LOC), number of methods (NOM), and number of attributes (NOA) are recorded in addition to other information such as method invocations and inheritance relationships.

In early software visualisation systems, those data were extracted and recorded in tabular formats in a database for later access. However, to take full advantage of the intrinsic modular nature of object-oriented development languages, it would be more sensible and practical to build real 'object models' that capture all the required information about each constituting code artefact. As open-source frameworks and libraries for automatically building such *object models* of systems started to appear, software visualisation tools quickly started to adopt them.

[1] http://scg.unibe.ch/download/Vera/

Vera is one such recent tool that provides this capability. It parses the system to be visualised and outputs a single Java object model that can be considered as a logical and abstract representation of that system. The model contains child objects, i.e., *variables*, portraying each single code artefact (package, class, method) of the original system. Each child object keeps the information about the corresponding code artefact. That single Java object is then input to ScrumCity where it is processed to build the visualisation.

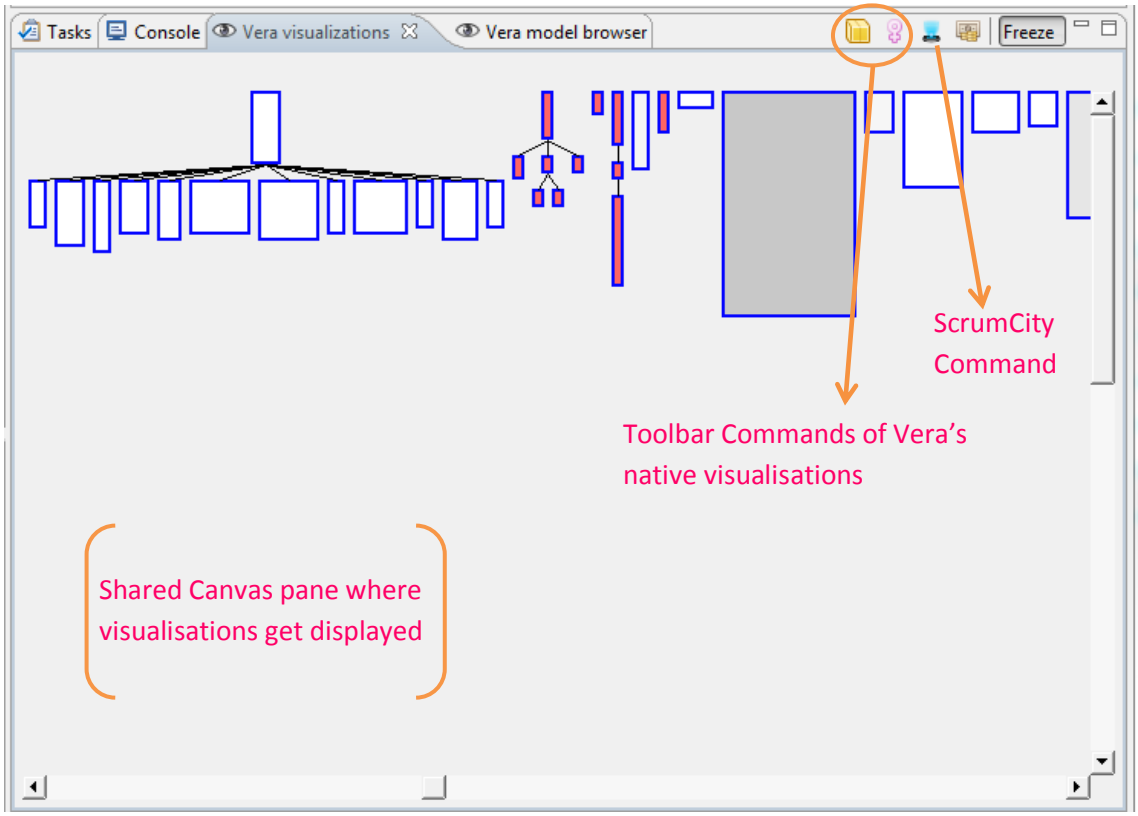


Figure 4.12: Vera's Eclipse Plugin showing ScrumCity's Toolbar command (Vera's native visualisation shown in canvas)

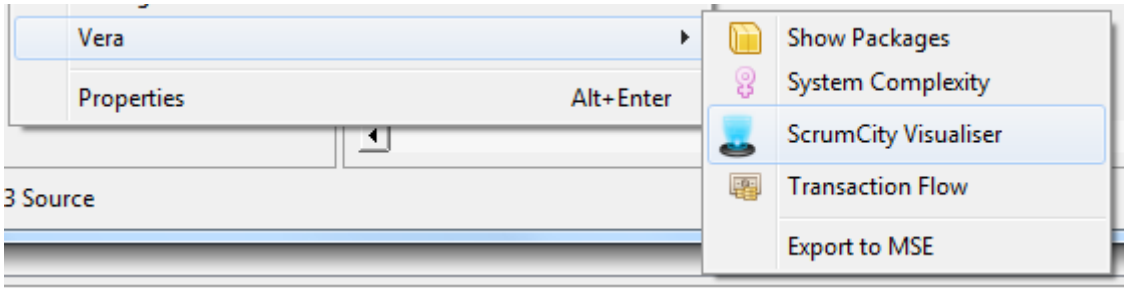


Figure 4.13: Vera's Contextual Menu (displayed when a *Java* Project is selected in Eclipse's *'Package Explorer'* view)

4.4.2 Building a Hierarchically-Structured Model

As briefly mentioned above, the model produced by Vera cannot be used 'out of the box' for the purposes of the city metaphor visualisation. To understand why this is the case, it is helpful to reflect on the layout employed in the city metaphor. Classes in the city metaphor are visually represented on top of their parent packages, and sub-packages in the same way are also represented on top of their parent packages. This results in a nested structure in which each system artefact is placed on top of its parent one. As a result, and due also to the nature of the layout algorithm (discussed next), it is necessary to have a secondary logical model to overlay the original model of the target system. That secondary model needs to reflect and exhibit the internal hierarchical structure of the original system. It needs to be in a tree-structure scheme where a root object is set to represent the project, and then that object holds child objects representing each root package, which in turn each have their own child objects representing sub-packages. This continues until methods of each class are represented by child objects of their parent classes. Figure 4.14 illustrates this hierarchical and containment structure. This model is necessary because it can eventually be utilised as a blue-print when creating the glyphs of the visualisation scene, and which will result in obtaining a visual replica of the system's actual containment structure.

While the Java object model produced by Vera captures all the required information about the system entities in a single and easily accessible model, the internal structure of that model object does not reflect the real hierarchical structure described above[1]. For this reason, a secondary model conforming to the described scheme has to be built. To accomplish that, Vera's object model is traversed and the full names (QNames) of packages are used to recursively build the hierarchical containment structure of packages. A similar approach is also used to build the class and method containment relationship. This process finally results in a secondary logical object model that overlays Vera's original source code model (see Figure 4.14). As has been

[1] The Vera Java Object Model uses a HashMap internally to store the child objects so in reality the structure of the object model is entirely different than the original system's structure. However, this object model provides access methods that make the whole model appear structurally similar to the original system's structure, except for packages which can only be accessed in a flat way.

introduced in the architecture section, this resultant model is processed by the layout algorithm first and then it is used to create and render the visualisation glyphs.

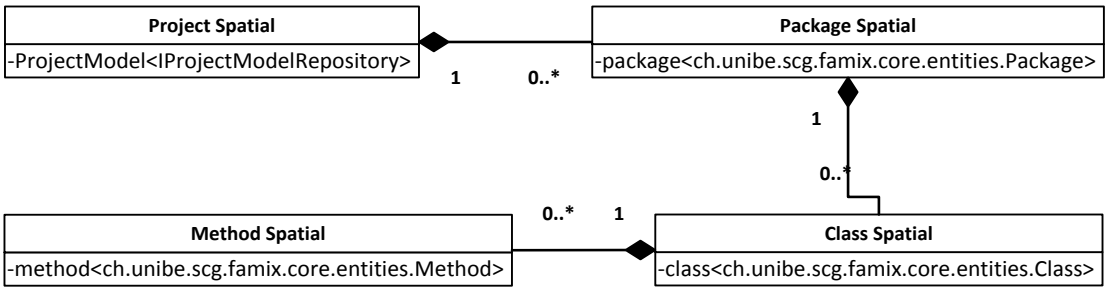


Figure 4.14: A class diagram illustrating the hierarchical structure of the secondary logical model and showing how the different elements of Vera's original source code model are eventually mapped into an all-encompassing single object model that embodies the true structure of the system.

4.4.3 City Metaphor Layout Algorithm

Before discussing the layout algorithm, it is relevant to briefly review (see Figure 4.11) the city metaphor graphical representation technique. In the specific city metaphor version adopted here, a main platform represented by a thin cuboid is used to simulate a city real-estate landscape. Root packages are then represented by other thin layers of cuboids residing on top of the city platform. To obtain the containment structure view, sub-packages (also represented as thin cuboids) are further placed on top of their parent packages, each sharing a portion of the total surface space of their parent package, giving a district-like appearance. Classes are then represented by buildings placed on top of their corresponding parent package. The dimensions of each Class glyph are determined by two metrics: NOM for height and NOA for width (and further discussion on this is provided in Chapter 5, section 5.4.2 (*Normalised City Metaphor*)).

To understand the layout mechanism, it is important to note that the size of each container, i.e., a package or class cuboid, is directly related to the size of its constituting child containers. For example, to calculate the surface dimensions of a container representing a given package that contains three sub-packages and two classes, the surface areas of the two class cuboids as well as that of the three sub-package cuboids need to be calculated first. In other words, to determine the size of a

parent glyph, the sizes of its constituting child glyphs must be first determined. Since parent-to-child hierarchies can go down to unknown numbers of levels, this suggests a recursive approach is needed where the size of each glyph is determined by traversing bottom-up into the hierarchical structure of the system model. This is in fact exactly how the City Metaphor layout algorithm introduced by Wettel and Lanza (2007) behaves. This also explains now the need for building the secondary hierarchical structured model discussed above. Each object in that model corresponds in reality to a glyph object which is eventually assigned a geometrical mesh and is visually rendered.

The interested reader is referred to Richard Wettel's PhD thesis for a full and detailed description of the layout algorithm. That same algorithm was in fact also implemented by Rigotti in Manhattan (2011). ScrumCity adapts and uses the same algorithm, albeit with slight variations. The main variation relates particularly to presenting the methods *inside* their classes using the same layout technique as is used for presenting the classes *on top of* their parent packages in Wettel's original city metaphor. In their original version of the City Metaphor, Wettel and Lanza (2007) did not go so far as to represent methods, although later in his PhD (2010) Wettel provided a fine-grained variation where glyphs of classes were entirely replaced by buildings of bricks where each brick corresponded to a method.

4.4.4 Implementation of Remaining and Completed Work

As part of the proposed conceptual visualisation technique a special mechanism has been devised to enable users to monitor the progress of development for a given Release. On a mapping scale between Class glyphs and user stories, users are able to see a visual depiction indicating how much user story work has been completed for a particular Class. When this is extended to a collection of classes of a particular package (which from a software architecture perspective usually maps to a system module or sub-module) a potentially useful picture emerges providing a visual impression of how much work has been completed and how much is remaining for a particular system component or module.

Similarly, a User Story or a Sprint can be selected and the remaining/completed work depiction is then displayed for those classes involved in that user story or Sprint. This provides another potentially informative visual view of remaining/completed work from the perspective of a user story or a Sprint, where various classes, not necessarily in the same package, will be involved in the depiction.

To implement this mechanism, for each given class, the remaining work-hours of all user stories related to this class are added up to get the total remaining hours standing for this class. The same is done for the completed work-hours. The ratio of these two values is then computed and the Class glyph is turned partially transparent, where the height of the transparent portion is determined by the percentage of remaining work. Furthermore, the other portion, representing the completed work, is colour-coded (as per section 4.5.6) to provide a richer visual sensation of development progress. For example, classes with less than 20% completed work (relative to the remaining work) are assigned a red colour.

If a specific user story or a specific Sprint is selected, then data obtained from that selected Scrum artefact only (as opposed to all related user stories in the previous case) are involved in the completed/remaining work depiction.

This mechanism has been implemented mainly for Classes at this stage of development, but a preliminary experimental implementation has also been instituted for a Method-to-user story mapping scale.

4.4.5 Implementation of the Burn-down Chart

The burn-down chart is a core feature of Scrum practice as it provides project managers with various valuable statistics regarding progress and projections of estimated delivery dates. For this reason, it is contended that adding this feature is desirable for its inherent management value, rather than any conceived advantage from a software visualisation perspective.

Thus for each Sprint, a user can choose to display a 3D burn-down chart. The time span of the chart for each Sprint is determined based on either the *start date* and *expected completion date* attributes of a Sprint, or determined by finding the oldest and newest

dates of work entries. Furthermore, Work Entry records represent the major source of data for implementing this feature and thus they are heavily involved in various computations.

After determining the timespan of a Sprint, column bars are used to represent each day of a Sprint's duration where its height indicates the total amount of remaining work-hours up to that day. Future days of a Sprint (where no work entries yet exist) are represented by place-marks which allow the overall chart to give a visual cue of the days to go before a Sprint's expected completion date is reached.

Section 4.5.11 provides illustrations of this feature along with brief descriptions.

4.4.6 Implementation of a Custom Tool Tip

An implicit requirement of conceptual visualisation is to enable *contextual* and *in situ* exploration, learning, and reasoning about system artefacts and their design, and this has meant that textual information must be presented to the user in an integrated fashion within the visualisation and with minimal possible distractions. For ScrumCity, three categories of information are displayed to users: system artefact information, Scrum artefact information, and system artefact documentation. While the 3D graphics library (jME3) provides a well-supported third-party GUI (called Nifty GUI) which integrates well into the 3D environment, unfortunately the GUI library was not yet fully-fledged at the time of ScrumCity's development. Many GUI components have a very simplistic implementation and a few of them are hampered by bugs. As a result, a special Tool Tip GUI component has been custom-built so that it conforms to the desired behaviour.

The implemented tool tip can be triggered on and off with a simple key press, and different modes of operation can be further chosen with other keys. The custom implementation makes the tool tip behaves much like an information centre, with other modes of operation available to be configured.

Section 4.5.5 presents an example view of this tooltip in action.

4.4.7 Implementation of Automatic Transparency

It has been mentioned above that methods in ScrumCity are represented as glyphs contained inside their parent classes. Because visualisation of software structure down to the method level might not be desired by some stakeholders (e.g., due to potential clutter), it was decided that their representation should be made available only *on-demand*. To achieve this, class glyphs are by default rendered as completely opaque. However, during navigation when the user comes 'close' to a Class glyph (as measured by a specific distance), it is made transparent so that method glyphs inside become visible. When the user moves away, the glyph returns back to its default opaque status. The user can also select one or more different glyphs and invoke the transparency mode from a contextual right click menu, irrespective of their proximity.

This transparency mechanism was inspired by a similar approach introduced by Balzer et al. (2004, 2007) in their 'software landscape' metaphor (introduced in Chapter 2) and was called dynamic transparency in that work. In their metaphor, spheres of multiple nesting levels were turned transparent, with changing scale of transparency based on the viewer's distance from the spheres.

Turning glyphs transparent, however, enables users to only *view* the method glyphs contained inside, but not interact with them. To solve the latter need, another technique has been implemented to automatically detach and restore the parent class glyph automatically. After a class glyph turns into transparent mode, when the user moves even closer, the class glyph is completely detached allowing users to interact freely with the methods. The Class glyph is restored once the user moves away from those Method glyphs.

Illustration of this feature in operation can be found in section 4.5.2.

4.5 System Features

This section presents the main features implemented in the ScrumCity tool along with brief descriptions of each[1]. While some features are Scrum-specific and are intended to support the main theme of the tool, other features are more general and so are intended to enhance the *utility* of software visualisation tools in general. Many of those features have been informed by the literature and so address specific areas that have previously received minimal or no attention (see section 2 of Chapter 5). Thus, collectively they attend to and aim to fulfil the secondary objectives of this research that were disclosed in the introductory chapter of this thesis.

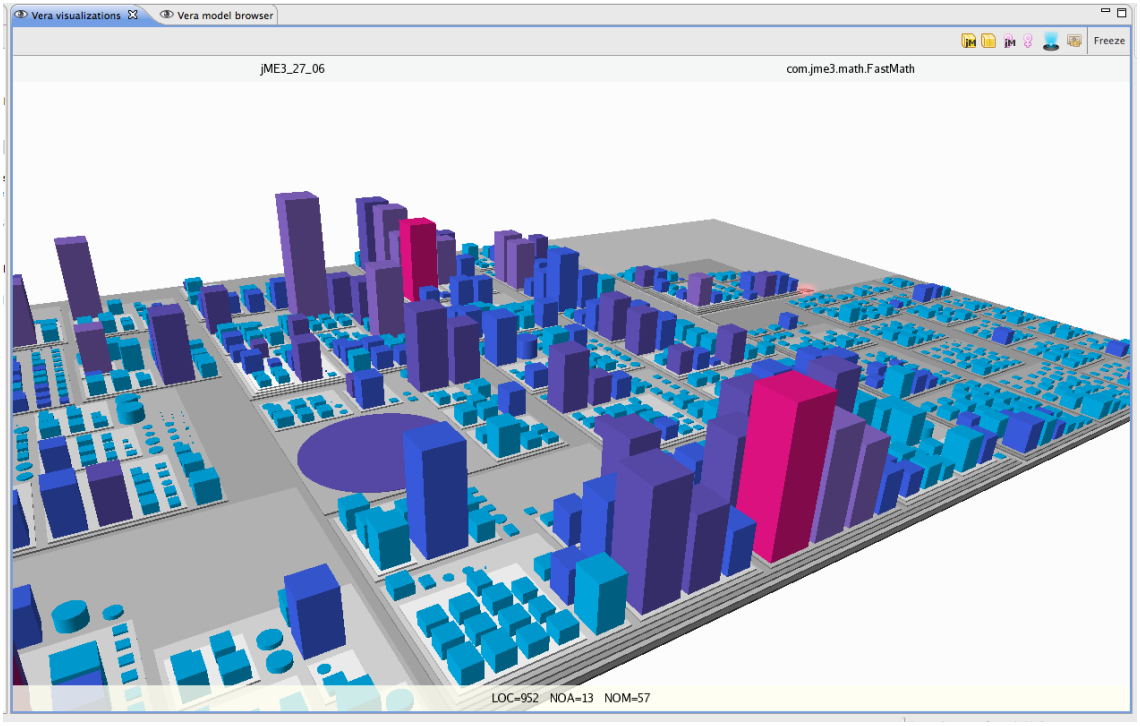


Figure 4.15: Example view of the City Metaphor Layout as implemented in ScrumCity

4.5.1 The City Metaphor Layout

Figure 4.15 shows an example of the initial view produced by ScrumCity when a project is first visualised. The view shows a system's static artefacts (the system is

[1] The reader is referred at this point to the accompanied video demo where all the features and functionalities described here are illustrated. The demo was created on a different machine (MacBookPro OS X 10.7.5, 2.4 GHz) that is of lower performance than the machine used to create the majority of the figures in this thesis (see Table 5.2 for that machine's specifications). The video demo is also available on YouTube: http://youtu.be/XEEcXOk-KW0

jME3 itself in this case) being visualised using an enhanced version of the Wettel et al. city metaphor. Cylindrical glyphs are used to distinguish interfaces from concrete classes[1].

4.5.2 Method Representations (On-demand Transparency and Detachment)

Figure 4.16 shows views of two different scenarios where Method glyphs that are rendered inside Class glyphs can be viewed and interacted with. Depending on the user's distance from a Class glyph, the glyph is automatically turned transparent or is detached completely to allow interaction with methods. Upon moving away by a certain distance, glyphs are restored to normal appearance.

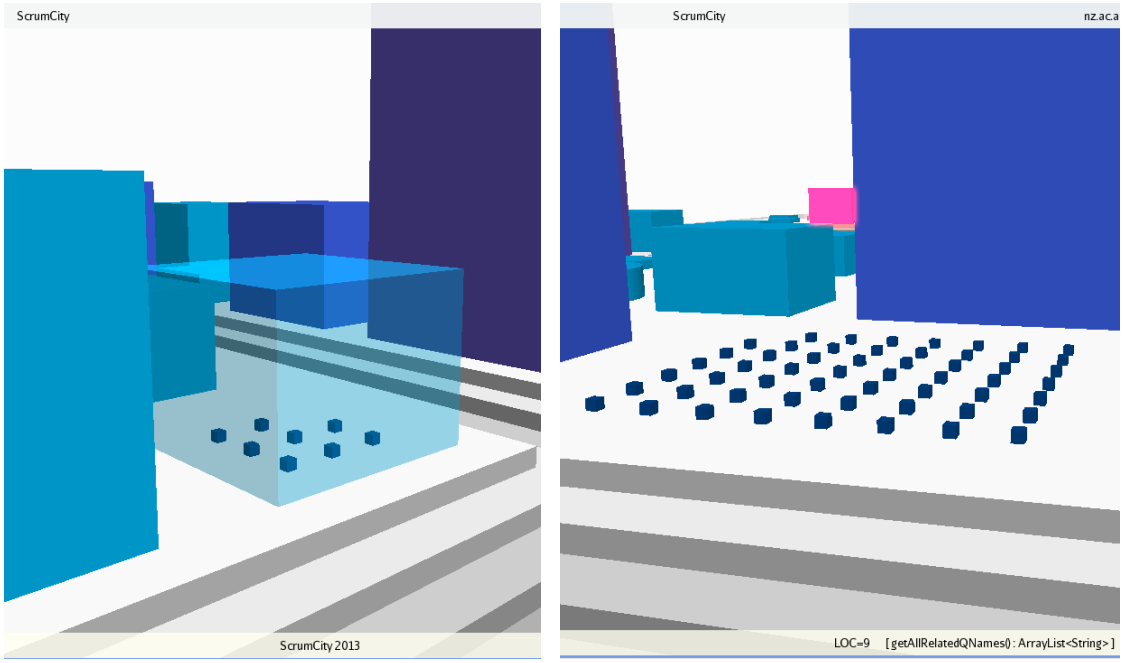


Figure 4.16: Class glyph is turned transparent due to user navigating closely (Left). As user approaches further, the Class glyph is detached allowing interaction with methods (Right).

4.5.3 Presentation of Software Processes (Scrum Artefacts and Activities)

ScrumCity presents a unified view of software processes and software products. This presentation is achieved via various mechanisms illustrated as follows.

[1] Although not part of Wettel's original city metaphor, cylinders and cuboids have been commonly used in 3D software visualisation metaphors since the field's early days. Cylindrical shapes were also used in Manhattan (2011) to represent interfaces.

Scrum Artefact List

By pressing the '**1**' key on the keyboard, a special graphical interface showing a list of Scrum artefacts, namely *Releases*, *Sprints*, and *Features* (user stories), is displayed. Those Scrum data are loaded from a user-provided XML file. Ideally, a tree-list based GUI would be used in such a scenario for the presentation. Unfortunately, at the time of ScrumCity development, the available version of Nifty GUI (v1.3.1), which is a third-party GUI library integrated into jME3, had a very primitive implementation of a *TreeBox* GUI control that was plagued by bugs hindering its usage[1]. Hence, to demonstrate the concept being introduced here, a simple *ListBox* control has been used instead (more information is revealed in Chapter 6, section 6.4). Figure 4.17 shows this control with simulated Scrum data being displayed. To work around the fact that hierarchical tree representation is not possible at this stage, simple empty records have been used to separate release, sprint, and feature records with special prefixes used for further distinction.

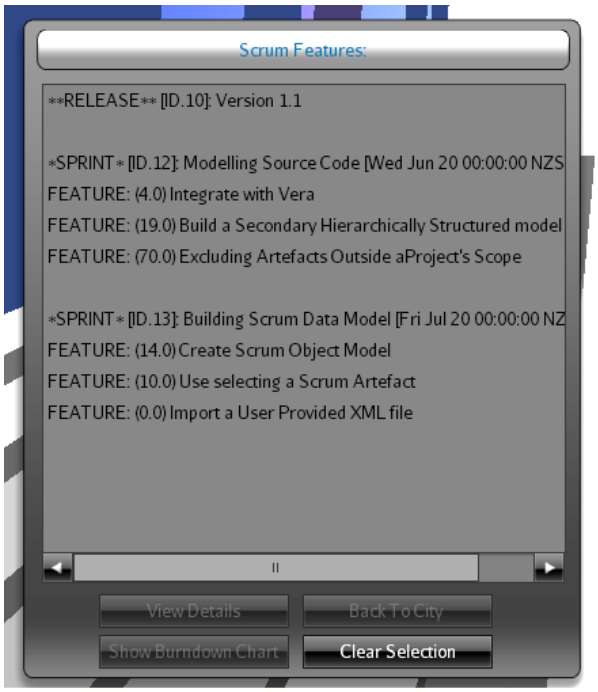


Figure 4.17: Scrum Artefact List

The same keyboard key is used to hide or show the Scrum List GUI (toggle on/off).

[1] Some discussion of the technical problems encountered can be found on a jME3 forum post at: http://jmonkeyengine.org/forum/topic/treeitemselectedevent-is-not-being-published/#post-182530

Depending on the type of the Scrum artefact selected, the Scrum List GUI control provides various other functions accessible via the buttons at the bottom. These functions are presented below.

Scrum-To-Glyph Mapping

As has been introduced above, the Scrum-To-Glyph mapping must be implemented to work in a bi-directional mechanism in order to realise the conceived advantage of the introduced conceptual visualisation.

When a feature is selected from the Scrum List GUI, the user is automatically 'transported' using animated transition to the specific Class or Method Glyph with which the selected feature is involved. Once the target glyph is positioned at the centre of the view scene, the glyph is highlighted with a blinking effect in place for a few seconds. The reader is referred to the accompanying video demo to see the feature at work. Figure 4.18 shows the final scene where the target glyph is highlighted and presented at the centre of the view scene.

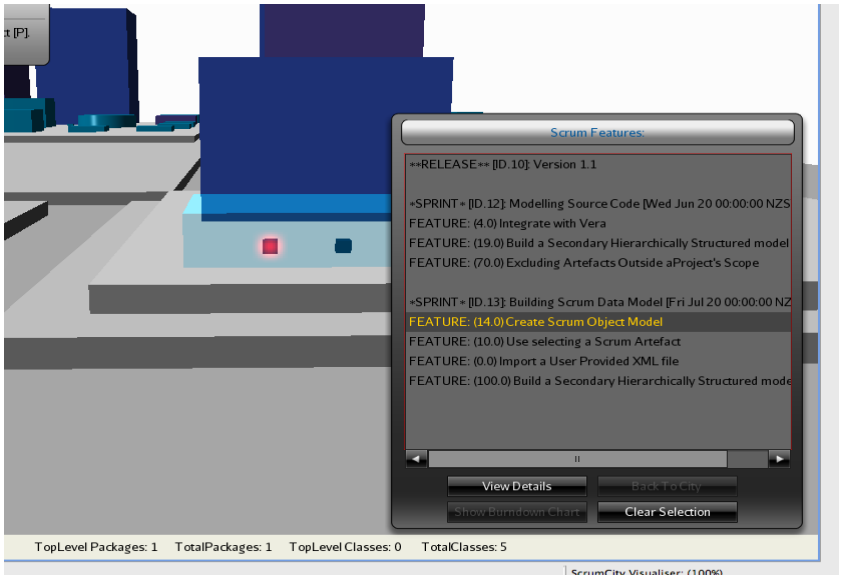


Figure 4.18: User is automatically transported using animated transition to the related system artefact after a feature is selected (final scene is shown)

Figure 4.19 shows the visual effect after selecting a Sprint from the Scrum List user interface.

Working the other way, by selecting a glyph from the scene (i.e., a Class or a Method glyph), all related user stories are highlighted in the Scrum List user control, allowing the user to identify the related user stories and giving them the opportunity to inspect and examine the details of those user stories as needed.

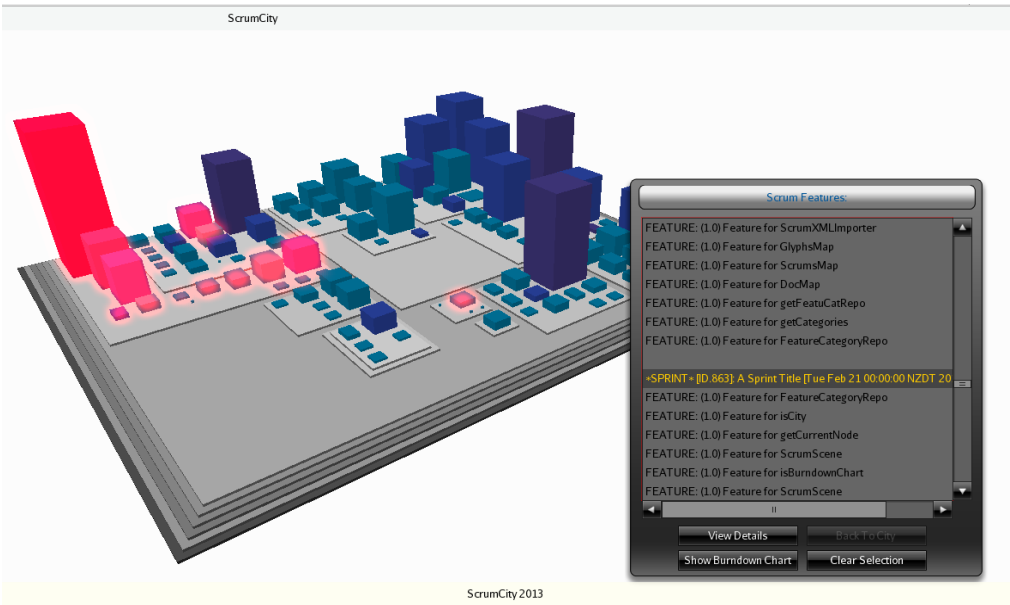


Figure 4.19: Selecting a Sprint reveals the locality of its contribution within the system's structure

In situ Information (Overlay Popup Screens)

The Scrum List GUI control provides access to view the details of a selected Scrum artefact. Figures 4.20 shows different detail views (based on the type of a selected Scrum artefact) accessed using the 'View Details' button. Special overlay popup screens have been designed to present the data in context and in a potentially non-distracting manner while the user is exploring or examining the system components.

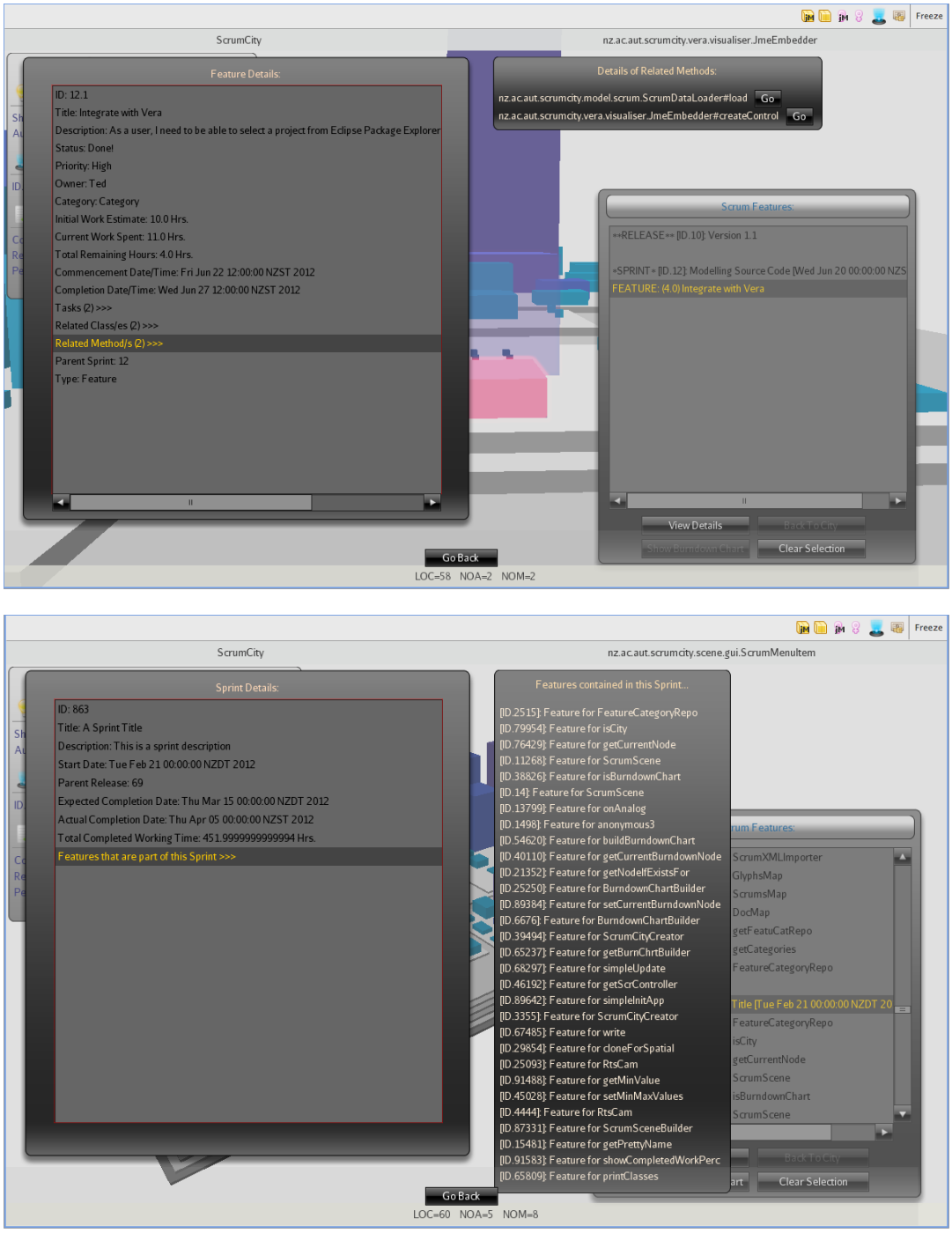


Figure 4.20: Two examples of information being presented in context.

4.5.4 System Artefact Search

Since real world software tends to consist of large numbers of components, trying to manually find an artefact of particular interest in the city visualisation would likely be a tedious and impractical task. Hence an artefact search feature has been implemented with various search strategies. The search functionality can be instantly displayed or hidden using the '**2**' numerical key. Figure 4.21 shows the search mechanism displayed.

Once an artefact is found, the user is automatically 'transported' to it using animated transition and the artefact is then blinked and highlighted in the same manner described above. The prefix '#' can be used to selectively search for methods only. If the hash prefix is not used then only class and package names are searched. In all cases, the search term is compared only to the last part of a QName to give proper search results. The simple justification for this strategy is that comparison to the whole QName string would mean that artefacts whose parents have a matching term would be returned and this is most often not the desired behaviour.

The *'All Matches'* button only searches for and highlights artefacts found with matching terms, without performing any transition. The reader is again referred to the accompanying video demo to see this feature in action.

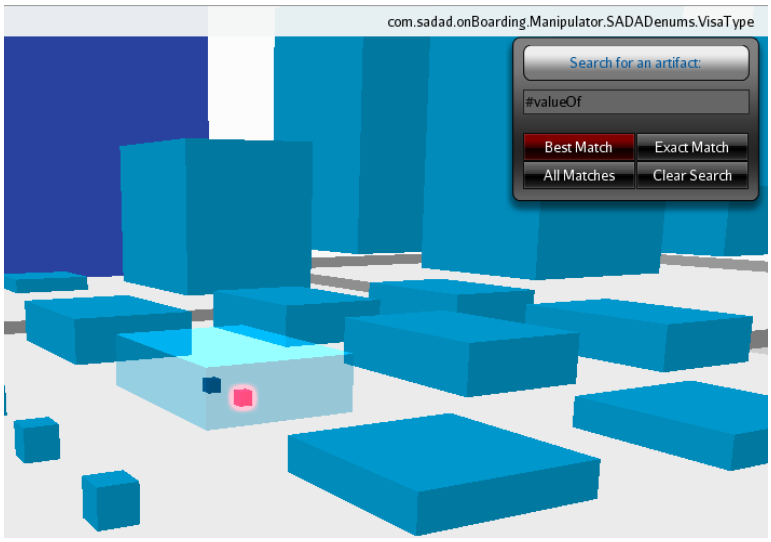


Figure 4.21: The Search functionality

4.5.5 Custom-Built Tool Tip

The reasons behind the creation of the custom tool tip have been discussed above. The tool tip has three mode of operation: Scrum Mode, Documentation Mode, and Combined Mode. The modes can be easily selected by pressing the '**3**', '**4**', or '**5**' numerical keyboard keys, respectively. The tool tip can be disabled using the '**9**' keyboard key. The default operation mode for the tool tip is the combined mode, which is shown in Figure 4.22.

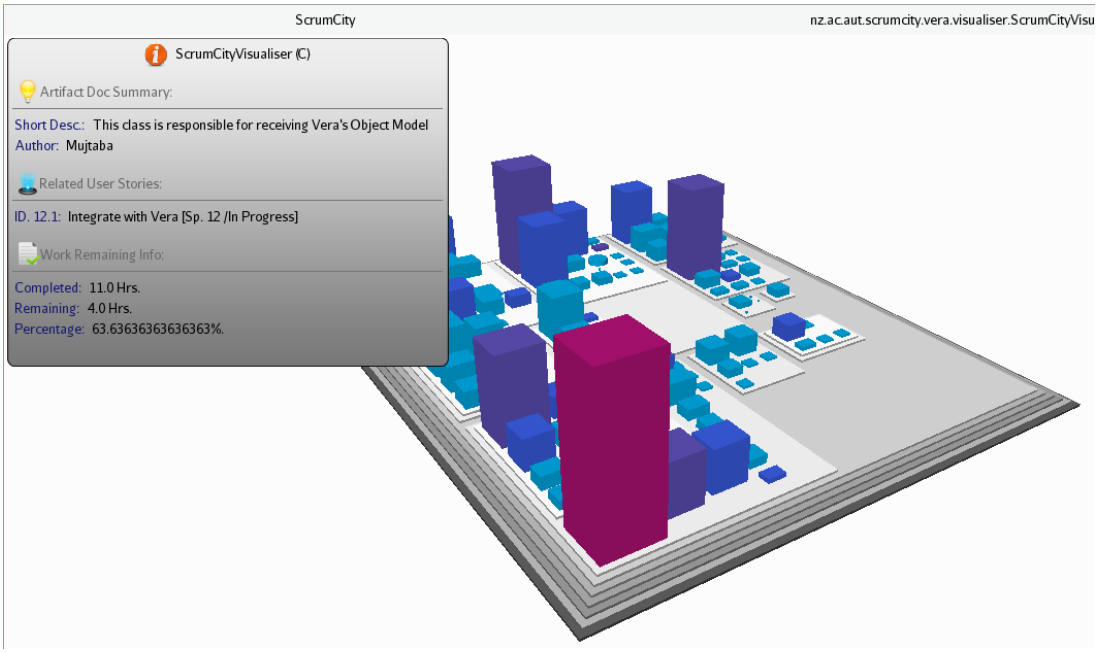


Figure 4.22: Tool tip in combined mode

4.5.6 Remaining and Completed Work View

Figure 4.23 depicts an example scenario of the completed/remaining work view discussed in the previous section. By right clicking on a class glyph, the user can choose to view a visual depiction of the ratio of completed work to that which is remaining. The numerical key '**6**' of the keyboard can be used to toggle the completed/remaining work view for all classes. Figure 4.24 shows the 5 colour-code scheme used to give a further visual cue to the percentage of remaining work.

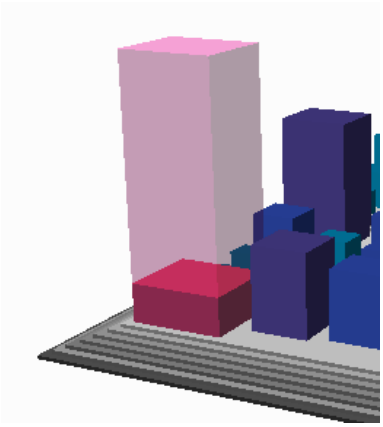


Figure 4.23: Example Scenario showing a depiction of completed to remaining work-hours ratio.

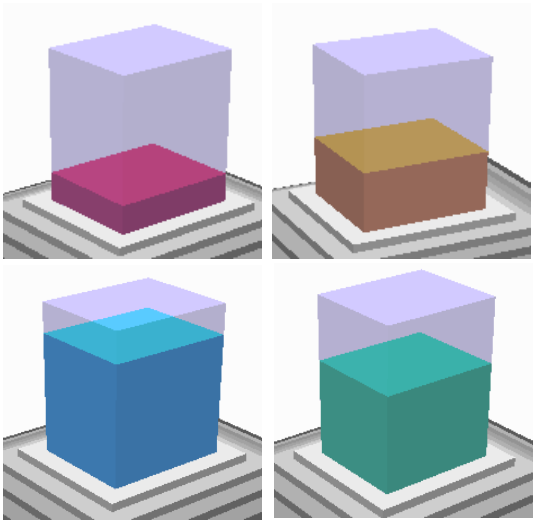


Figure 4.24: Colour-coding for completed work-hours percentage. From top left, clockwise: <20%, <40%, <70%, and >70%. Percentage is calculated in relation to remaining work-hours.

4.5.7 Contextual User Interaction

By right clicking on a glyph, a contextual menu is displayed from which specific functions can be performed based on the type of the glyph. For a Class type glyph, those functions include viewing full details of related features (user stories), viewing artefact documentation, showing contained artefacts (methods), showing the completed/remaining work depiction, and detaching the class glyph to interact with inner components. Figure 4.25(a) shows an example view of these menu commands.

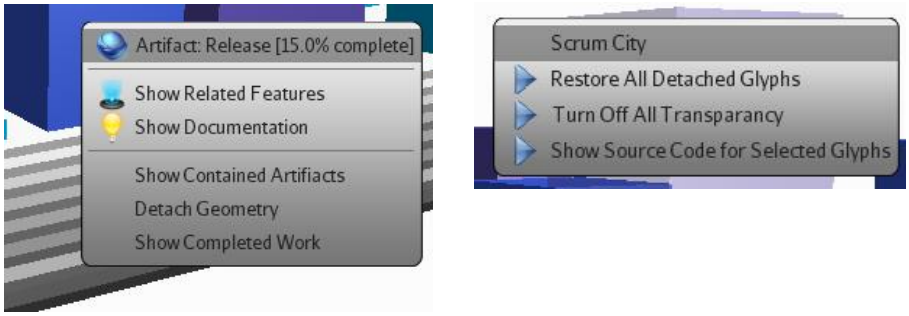


Figure 4.25: Left (a): Contextual right-click menu for glyphs. Right (b): Example of the general contextual menu.

4.5.8 Glyph Selection

Any glyph in the city landscape can be selected by clicking the mouse's left-button while holding down the ALT key. When selected, a glyph glows in reddish colour. This enables the user to highlight an area of interest. Moreover, depending on the type of the selected glyph(s) and the current status of the scene, right-clicking on an empty area of the scene while some glyphs are selected will reveal a general pop-up menu with various commands according to the situation in place. Figure 4.25(b) depicts an example of this contextual menu

4.5.9 Enhanced Navigation

Navigation in a 3D environment is a major issue in software visualisation as has been indicated in the literature review chapter of this thesis (and see also Chapter 5, section 5.2), with many researchers highlighting it as a core issue in need of suitable treatment before 3D software visualisation can enjoy practical use in the SE industry. As a result,

special attention and effort have been expended in this regard in the hope of attaining an acceptable resolution.

The standard navigation mode commonly found in 3D software visualisation tools is the conventional WASD key combination. In this standard WASD navigation mode, the 'W' and 'S' keys are conventionally used for zooming-in and zooming-out, respectively, while the 'A' and 'D' keys are used for left and right panning, respectively. Many of the 3D software visualisation tools surveyed during the course of this research offered only this conventional WASD navigation mode, with a few also offering rotation around axes using the arrow keys. As a result, the navigation experience offered by these tools is hampered and laden with limitations making it far from ideal. ScrumCity, in contrast, uses a variety of navigation mechanisms in order to enhance the user experience.

Combined Fly-Camera Mode

ScrumCity implements a special navigation configuration that combines the WASD mode with a *fly-camera* mode allowing the user to use both the *mouse* and the conventional WASD keys *simultaneously,* resulting in a smooth navigation experience. Furthermore, another two keys, 'Q' and 'Z', can be used in the same mode for upward or downward elevation. With this configuration, the user can virtually walk-around and explore the neighbourhoods of the city. Compared with other configurations, this is considered to provide an improved navigation experience. The combined fly-camera mode can be toggled on using the CTRL key, which captures and locks the mouse cursor in the 3D scene, allowing the user to start to use the mouse and WASD keys together. When the CTRL key is pressed again, the mouse is released and the navigation mode returns back to the default WASD mode.

Enhanced WASD Mode

Some enhancements have also been added to the default WASD mode. In this mode, the four arrow keys can be used to rotate the city around its own axes, enabling the user to move to the desired side of the city or tot tilt the city to a desired angle. In addition, this mode has also been augmented with mouse interactivity. The mouse is not locked in this mode; rather, the user can use it to hold and drag the city to a desired location, by pressing and holding the left mouse button during the interaction.

This produces a pan-like effect in the four directions. The mouse wheel can also be used to zoom in or out at a speed slower than that offered with the WD keys.
The accompanying video demo demonstrates these various enhancements to the navigation experience.

4.5.10 Source Code Integration

Class and Method glyphs are tightly integrated with the actual source code artefacts that they represent. By clicking the mouse's left button on such a glyph while holding the SHIFT key down, the source code of that artefact is opened in Eclipse's native Java editor. If the artefact is a method, the editor opens to the exact location of that method in the file.

To open the source code of more than one glyph, the glyphs of interest can be first selected and then right-clicking over an empty area of the scene graph reveals a contextual menu with a command for opening the source code of the selected glyphs. This can also be achieved by holding the SHIFT key and then pressing the '**O**' key.

4.5.11 Burn-down Chart

By selecting a Sprint from the Scrum List GUI control, a burn-down chart can be displayed showing a graph of the remaining work-hours per day across the Sprint's timespan. When placing the mouse cursor over a particular bar column (representing a day) or over the chart base, different statistical details are displayed in the tool tip. Figure 4.26 shows an example view of the burn-down chart functionality. The 'back to city' button returns the user back to the main city visualisation.

4.5.12 Colour-Coding for LOC

In ScrumCity, Class glyphs are colour coded according to their LOC metric value. The colour scheme should enable viewers to easily identify where in a system the various

levels of responsibility occur[1]. Figure 4.27 illustrates this colour-coding scheme and the criteria used within it.

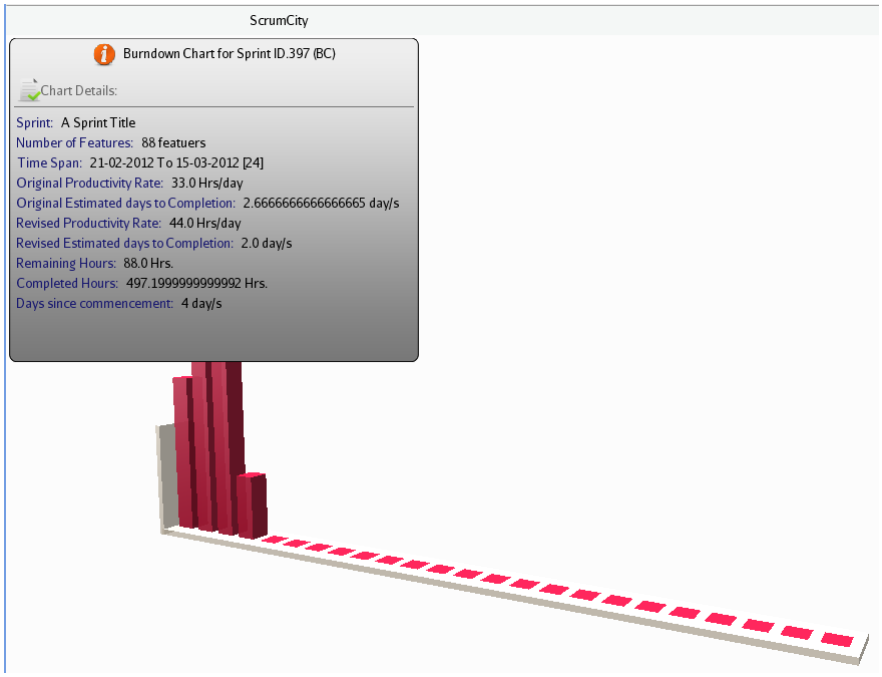


Figure 4.26: Burn-down chart for a simulated Sprint data

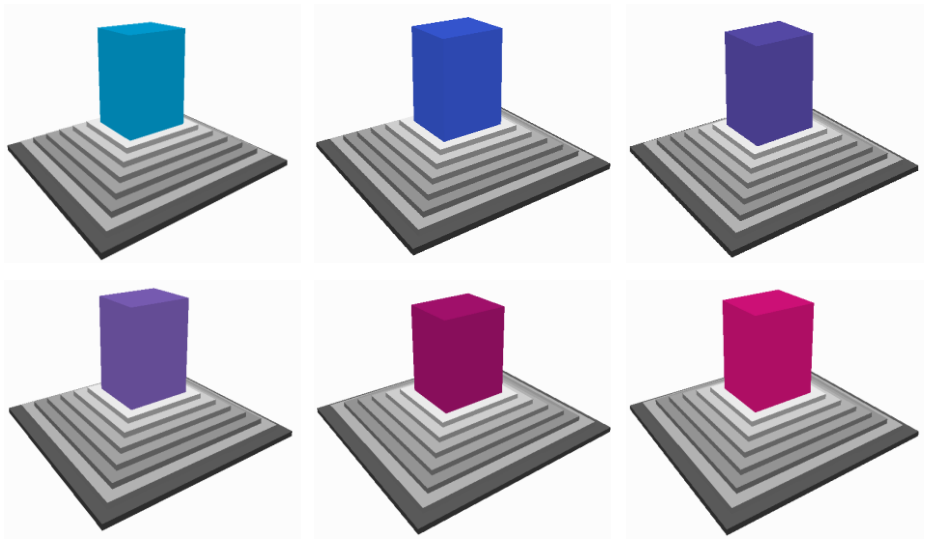


Figure 4.27: LOC colour mapping. From left to right clockwise: <200, <500, <1000, <1500, and >2000

[1] Mapping the LOC metric to particular colours is part of Wettel's original City Metaphor but since ScrumCity uses a different colour-coding scheme, it is necessary to illustrate this scheme here.

4.5.13 Colour-Coding for Package Nesting Level

Similar to the approach just described for mapping Class colours to the LOC value, the nesting level of packages is also indicated in ScrumCity using a similar colour-coding scheme technique. Root packages are given a dark grey colour, and as the nesting level increases, the colour becomes lighter. This technique is part of Wettel's original city metaphor. Figure 4.28 demonstrates this characteristic.

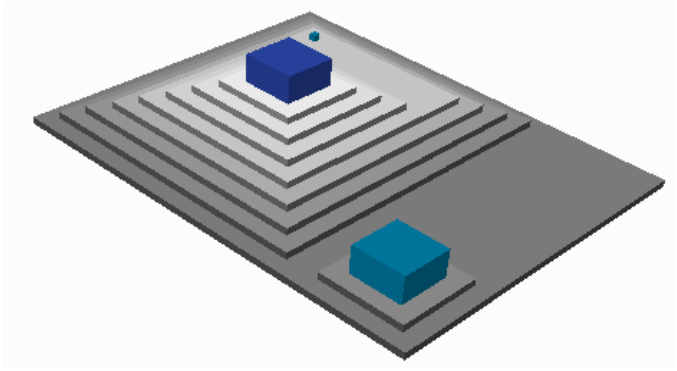


Figure 4.28: Package nesting level indicated using colour mappings.

4.5.14 Top-down and Side Views

Top-down and side views of the city can be displayed by pressing the '**U**', or the '**I**' key respectively, while holding down the SHIFT key. Pressing the '**R**' key while holding the SHIFT key down resets the visualisation scene to the default isometric view. Figure 4.29 shows examples of the side and top-down views.

4.5.15 Keyboard functions Map

ScrumCity makes extensive use of keyboard commands, a common feature in 3D virtual environments. To help guide new users to the available functions, a keyboard function map can be found in Appendix B.

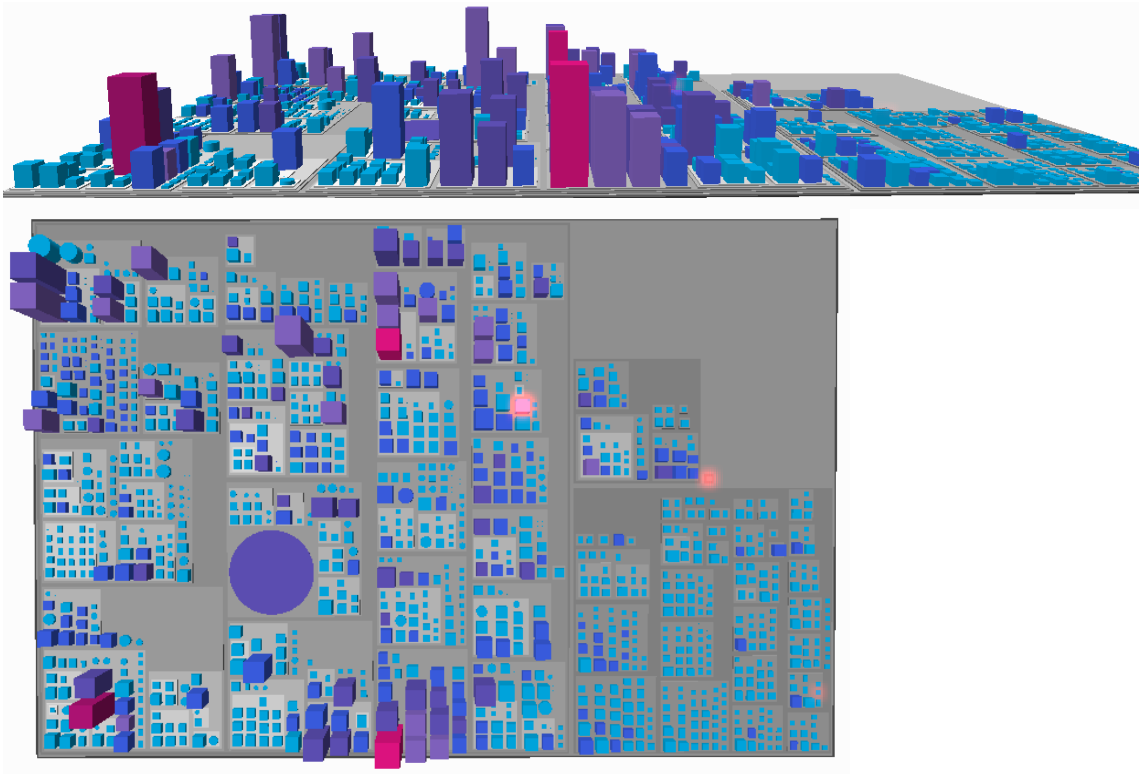


Figure 4.29: Side View (Top) and Top-down View (bottom) of JMonkeyEngine3

4.6 Summary

In all design-science research, communicating the development process is a major activity and one through which the continuous gradual advancement of a field can occur. Hence, in accordance with the guidelines of design science methodologies, this chapter set out to communicate the details of the development process as instantiated in this research project. It began with a discussion of the overall architectural design of the developed proof of concept tool, highlighting the major elements and components behind it. Each of those major components was also presented separately in detail, explaining its purpose and working mechanisms. Furthermore, the commonality of design shared by most other software visualisation tools was highlighted, indicating where ScrumCity differs and why.

Most importantly, this chapter has introduced and elaborately described the design of the proposed visualisation technique which is the main anticipated contribution of this research. Some important implementation details were then discussed, and the chapter was concluded with a description of the tool's features and functionalities.

5

System Evaluation

Validation of the proposed *Conceptual Visualisation* technique and assessment of its capabilities and potential applications are presented and demonstrated in this chapter. Evaluation criteria drawn from the literature along with recommendations of previous researchers are first presented, and these then form the basis against which the visualisation technique and the prototype tool are assessed.

5.1 Introduction

As stated, the primary objective of this research is the introduction and manifestation of the novel visualisation technique referred to here as *Conceptual Visualisation.* According to the discussion in Chapter 3, the feasibility of this technique is asserted by the actual construction (and then demonstration) of the prototype tool, ScrumCity. However, this work also has other important secondary objectives that are intended to address issues in software visualisation that, according to previous researchers, need attention.

Two strategies are therefore adopted in the evaluation procedure for this research. Initially, the utility of the new concept is demonstrated in case studies by applying the visualisation to several real-world open-source systems using simulated Scrum data. Potential real-world applications are then revealed and discussed in light of prior literature and past development work in order to further demonstrate the utility of the technique and its relevance. In addition, some of the issues that have been deemed by previous researchers to hinder the practicality of 3D software visualisation in the SE community are specifically discussed, showing how they have been addressed in ScrumCity – in the hope of taking this relatively new technology closer to the life of the everyday developer as well as other potential software stakeholders. Those issues constitute the secondary objectives of this work and their achievement is collectively anticipated to help in mitigating the low rate of adoption of '*3D'* software visualisation, in particular.

5.2 Issues in 3D Software Visualisation

To initially guide this research in an appropriate direction, some effort was expended on identifying the most prominent issues that stood in the face of SV use in general and 3D SV use in particular. Such efforts are also important in order to avoid 're-inventing the wheel'. With an apparent increase of interest by academia in software visualisation research during the past decade, the field has recently witnessed a proliferation of reviews, surveys, and taxonomies. As was indicated in Chapter 2, some of those surveys were specifically oriented to identifying '*desirable features*' or '*features that make for an effective software visualisation*'. Sensalire et al. (Sensalire et al., 2008, 2009; Sensalire & Ogao, 2007a, 2007b) in particular have published a series of four papers to identify such issues based on empirical work. In total, 21 survey studies and literature reviews were carefully examined and studied as part of this research effort in order to identify the most prominent and/or enduring issues in SV research. Table 5.1 summarises those issues that are particularly relevant to 3D visualisation. These important elements, drawn from the literature, may be considered to act as broad evaluation criteria for the secondary objectives of this research. They are presented here to enable the reader to relate the anticipated contributions of this work to the literature and also to help in assessing the ScrumCity tool. Some of these issues have been tackled before with variable degrees of attention and success, but are still being identified as in need of further work. Those marked with a *star* suffix have, on the other hand, been identified as having received minimal attention from previous researchers.

Even from a brief, initial glance at the table, it is evident that most of the issues identified pertain to *tool design* and only a few relate to *visualisation techniques*. This apparently suggests that tool design and the technologies involved should receive more attention from future researchers. While most of these identified issues have been given some consideration during the design and development of the conceptual visualisation technique and during the design and implementation of the ScrumCity tool, this research has been focused primarily on the first seven issues. Hence those seven issues receive particular attention during the evaluation process in order to show how they render a visualisation more usable and therefore more effective and practical. Further general discussion is presented in section 4 of this chapter.

Table 5.1: Major Issues facing 3D software visualisation as drawn from literature

#	Feature	Citation
1	**IDE-integration***	(Sensalire et al., 2008), (Sensalire & Ogao, 2007a), (Kuhn et al., 2010)
2	**Good Searching Mechanism/ Query Support (including at visualisation level) ***	(Sensalire et al., 2008), (Sensalire & Ogao, 2007a), (Sensalire & Ogao, 2007a), (Kienle & Muller, 2007), (Gallagher et al., 2008), (Bassil & Keller, 2001)
3	**Navigation in the 3D environment***	(Sensalire et al., 2008), (Petre & de Quincey, 2006), (Gračanin et al., 2005), (Ghanam & Carpendale, 2008), (Young & Munro, 1998), (Gallagher et al., 2008)
4	**Non-distracting and approachable user interface***	(Petre & de Quincey, 2006), (Kienle & Muller, 2007), (Bassil & Keller, 2001)
5	**Utilisation of Animation***	(Sensalire et al., 2008),
6	**On demand display of details and meta-data***	(Sensalire & Ogao, 2007a), (Sensalire & Ogao, 2007a), (Petre & de Quincey, 2006), (Beck & Diehl, 2010)
7	**Scalability (in terms of the visual metaphor when visualising large-scale systems)***	(Sensalire et al., 2008), (F. Steinbrückner & Lewerentz, 2010), (Bassil & Keller, 2001)
8	Simplicity (of use & installation)	(Sensalire & Ogao, 2007a), (Bassil & Keller, 2001)
9	Responsiveness	(Sensalire & Ogao, 2007a), (Bassil & Keller, 2001)
10	Varying Level Of Detail (Metaphor) – a.k.a. Elision	(Petre & de Quincey, 2006), (Beck & Diehl, 2010), (Gračanin et al., 2005)
11	Source Code Integration	(Petre & de Quincey, 2006),
12	A metaphor that is resilient to change	(Petre & de Quincey, 2006), (Gračanin et al., 2005), (F. Steinbrückner & Lewerentz, 2010)
13	Good use of visual metaphors	(Petre & de Quincey, 2006), (Ghanam & Carpendale, 2008), (Young & Munro, 1998)
14	User interactivity (in the 3D environment)	(Petre & de Quincey, 2006), (Young & Munro, 1998), (Ghanam & Carpendale, 2008), (Gallagher et al., 2008), (Kienle & Muller, 2007)
15	Integration of documentation and other informal sources of information (e.g. email communications)	(Storey et al., 2005)
16	Level of Automation (e.g. mechanism of importing the source code to be visualised)	(Sensalire & Ogao, 2007a), (Gračanin et al., 2005)

5.3 Laboratory Validation

The aim of this section is to demonstrate the utility of the principal concepts behind the introduced visualisation technique and to verify that it achieves the objectives of this research. As introduced and described in Chapter 4, the conceptual visualisation technique relies on the availability of real Scrum data (reflecting Scrum artefacts and activities) for the system to be visualised. Scrum data, as has also been explained, intrinsically represents and captures the software development *processes* – which through their enactment produce the various system artefacts. Since obtaining such data for a real-world system is beyond the scope of this research, a simple mechanism has been employed in ScrumCity to optionally allow for the generation of *simulated* Scrum data specific to the system being visualised when no real XML Scrum data is available. In this way, it becomes possible to test and demonstrate the visualisation technique for any system for which only the source code is available.

So for demonstration and validation purposes, six real-world open-source systems have been chosen taking into account that they collectively cover a reasonable range of system sizes (and working from the stance that very small systems do not benefit significantly from visualisation). The smallest system of those chosen consists of 66 classes while the largest has a total of 1315 classes. More importantly, the selected systems have various characteristics that help to highlight particular features of the developed tool, and that are disclosed in context as each system's visualisation is discussed. The chosen systems are: AntViz, Apache IvyDE, jEdit, jMonkeyEngine3, Shrimp Suite, and ScrumCity itself. A brief description of each system is provided in context in the following sections.

5.3.1 Environment Specification

ScrumCity was developed on a MacBook Pro machine with a 15-inch display. However, to take advantage of a bigger screen size, the validation process was carried out on a Windows 7 Desktop machine equipped with a 22-inch screen. Table 5.2 shows the hardware specification details of this machine.

Table 5.2: Specification details of the machine used in the validation process

Processor	2 GHz Intel Core 2
Memory	4 GB
Graphics	Radeon X1600

5.3.2 Case Studies

This section reports a sequence of case studies using the six mentioned subject systems, in order to demonstrate the main functionalities of ScrumCity across multiple real-world systems[1]. A brief description and comments are provided here as appropriate for each case, while the discussion section that follows deals with other important general aspects.

AntViz

Description. AntViz[2] is a small application designed as a plug-in for the jEdit programming environment. It serves to provide graphical representations of dependencies in Ant scripts with some debugging capabilities.

Initial Visualisation view. Figure 5.1 shows the city metaphor representation of AntViz as produced in ScrumCity. As can be seen in the diagram, the system is relatively small and simple consisting of only 66 classes, most of which comprise fewer than 200 LOC. It took ScrumCity 5 seconds to produce the visualisation for AntViz.

[1] Higher resolution screenshots are available at : http://scrumcitytool.wordpress.com/
[2] http://sourceforge.net/projects/antviz/

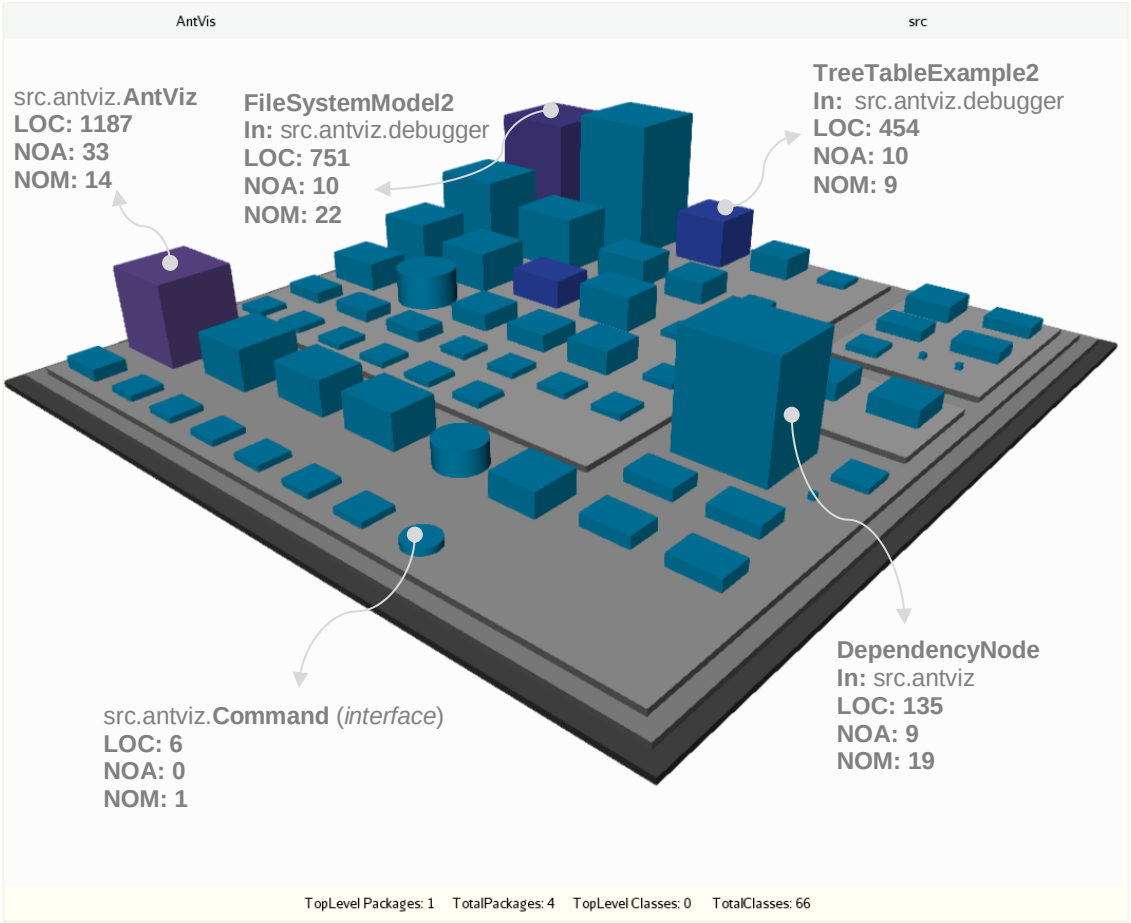


Figure 5.1: AntViz system as visualised by ScrumCity (initial view)

Some artefacts of particular interest have been annotated to give the reader a sense of how glyphs' colour and size mappings should be interpreted. In the actual visualisation environment, upon the mouse hovering on a certain glyph, relevant metric data as well as the artefact's QName are displayed accordingly in information bars (shown top and bottom).

Remaining/Completed Work Depiction. During the visualisation process, ScrumCity was set to generate simulated Scrum data for AntViz, as explained above. This included generation of random WorkEntry records (see Chapter 4, sections 4.3.1 and 4.4.4) for each user story. Figure 5.2(a) shows the *Remaining/Completed Work* depiction for all the buildings of AntViz city as visualised using those randomly generated work entries. The depiction shows the proportion of remaining work as *unfilled* space for each building, as compared to that of completed work. The exact percentage figure can be read from the tooltip upon placing the mouse over a glyph.

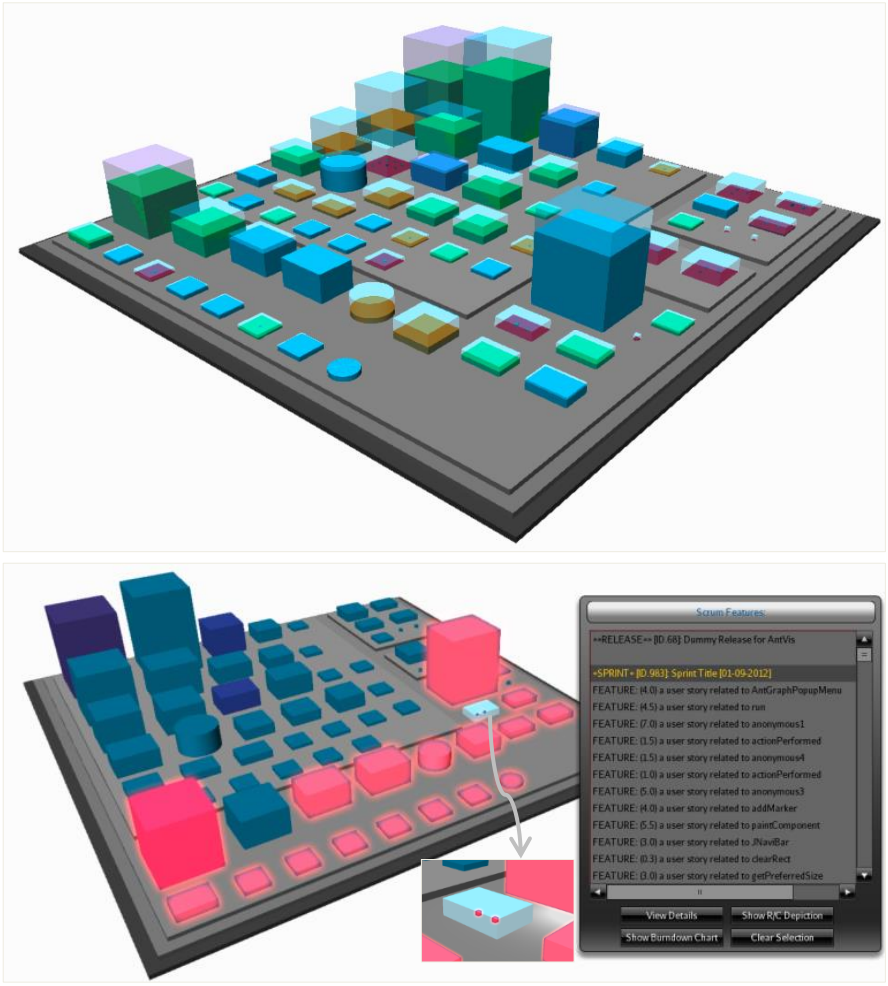


Figure 5.2: Top (a): Remaining Work Depiction of AntViz City. Bottom (b): Feature locality view for a selected user story.

Remaining work can be displayed for the entire city at once, for a particular building, or for a group of buildings related to a currently selected Sprint (illustrated later).

Feature Locality. As explained in Chapter 4, the main novel characteristic of the conceptual visualisation is the seamless and tight integration between the Scrum processes and their manifested product artefacts. Figure 5.2b (bottom) shows the effect of selecting a particular *Sprint* from the *Scrum Features* list box. All system artefacts in relation to the selected Sprint are instantly highlighted. One particular building is seen to have been turned to transparent mode. This is because, while this building is not directly related to the selected Sprint, a method inside it has a direct relation to that Sprint, and hence it is made transparent so the highlighted method(s) inside can be properly noted. (In this case, due to the random generation of simulated data, both methods happen to be in direct relation to the Sprint but not their parent Class.) A single user story (feature) can also be selected from the list box to find out the

buildings related to that feature in particular. The Scrum artefacts in the list box are tightly integrated with the city in the sense that a manual selection of a building in the scene graph will reveal the related Scrum features in the list box, i.e., the integration works in both directions.

Burn-down Chart. Figure 5.3 shows a burn-down chart being displayed for the selected Sprint. Hovering over a particular day-column displays the exact figures of remaining and completed work-hours for that day. The flat pads represent the days left until the expected completion date of the Sprint. The tooltip shown displays more statistical information which includes: time span, original and revised productivity rates[1], estimated days to completion based on both original and revised productivity rates, total remaining and total completed work-hours, days elapsed since commencement of the sprint, and total number of features. Extra details about the Sprint can be displayed via the 'View Details' button.

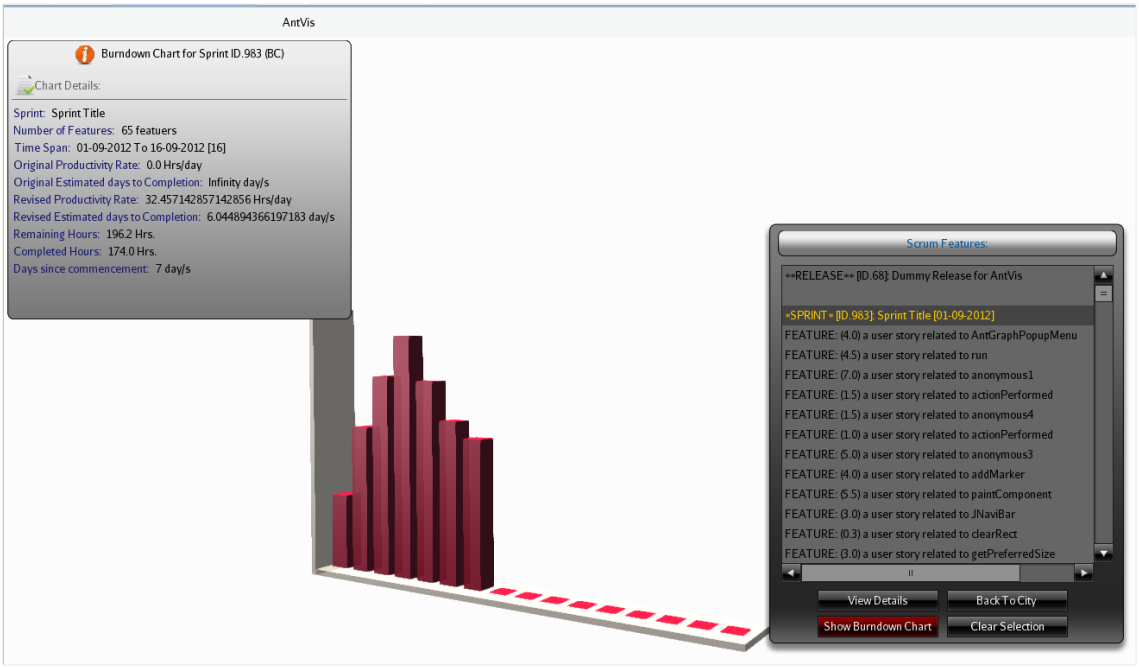


Figure 5.3: Burn-down chart displayed for a selected AntViz Sprint

[1] 'Original Productivity Rate' refers to the average value of completed work-hours per day obtained by dividing the difference in remaining work-hours between the '*first*' and the '*lowest*' days of a Sprint's time span by the number of elapsed days. In the particular situation observed in Figure 5.3, the 'first' and the 'lowest' days happen to be the same day and hence this results in the '0' value seen in the tooltip. On the other hand, the 'Revised Productivity Rate' is calculated by taking the difference between the 'highest' and the 'lowest' days, irrespective of the sequence they occur in.

ScrumCity

Description. It has become conventional in the research community of software visualisation to have the designed visualisation tool visualising itself, providing some insights into the tool's development in retrospect while serving as a validation measure at the same time.

Main City View. The city landscape[1] of ScrumCity can be seen in Figure 5.4 containing a total of 115 buildings and with execution time in this case of 8 seconds. The system has a degree of variance in terms of components' complexities (LOC and NOM) which is also dispersed across the different modules (packages). A single building stands out distinctively in the middle, which happens to be the 'ScrumScreenController' class – responsible for handling user interactivity in the virtual environment.

Other Scrum Views. Figures 5.5 and 5.6 both provide different scenarios for reasoning about and inspecting the system in terms of locality of Scrum artefacts and the status of development activities. Figure 5.5b shows particularly the added utility of depicting the '*completeness'* level of a selected Sprint.

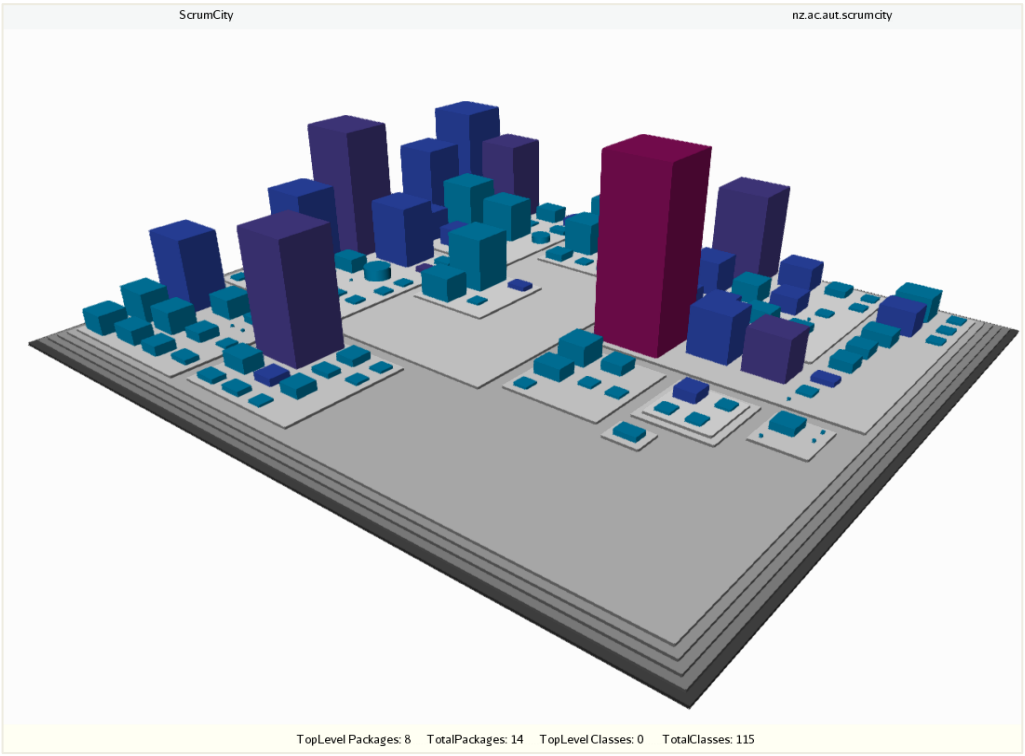


Figure 5.4: Main City View of ScrumCity

[1] The landscape of ScrumCity as it appears in the video demo varies slightly from what is depicted here. This is due to the fact that since the time of creating the figures for this Chapter, the package structure of ScrumCity code has been modified slightly as a result of final fine-tuning process.

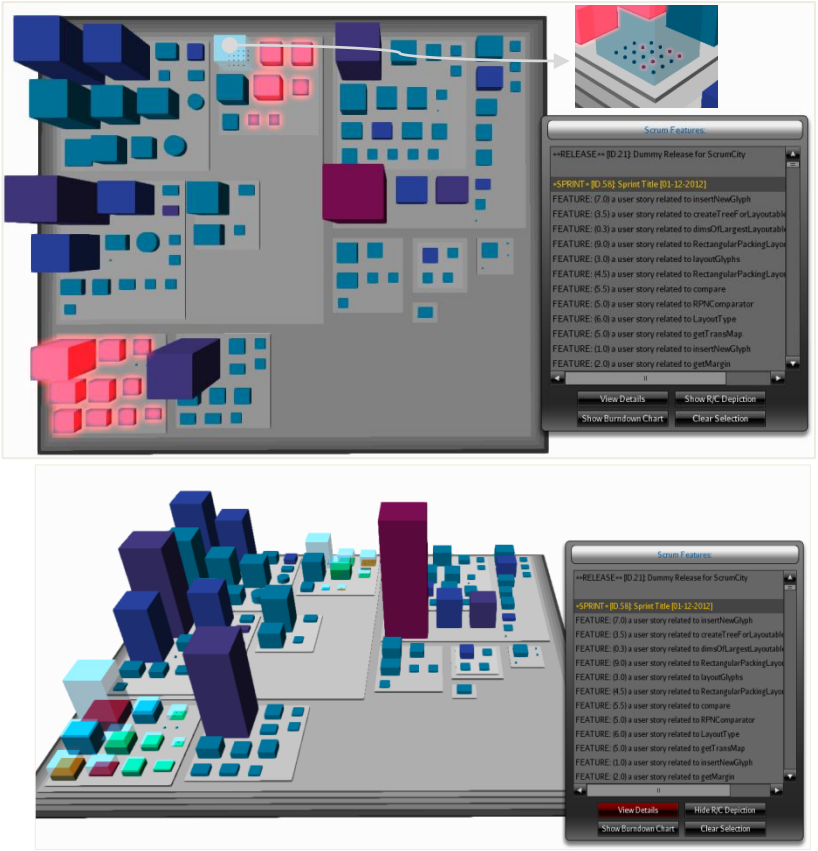


Figure 5.5: Top (a): Feature locality in ScrumCity for a selected Sprint. Bottom (b): Remaining work depiction view for the same selected Sprint.

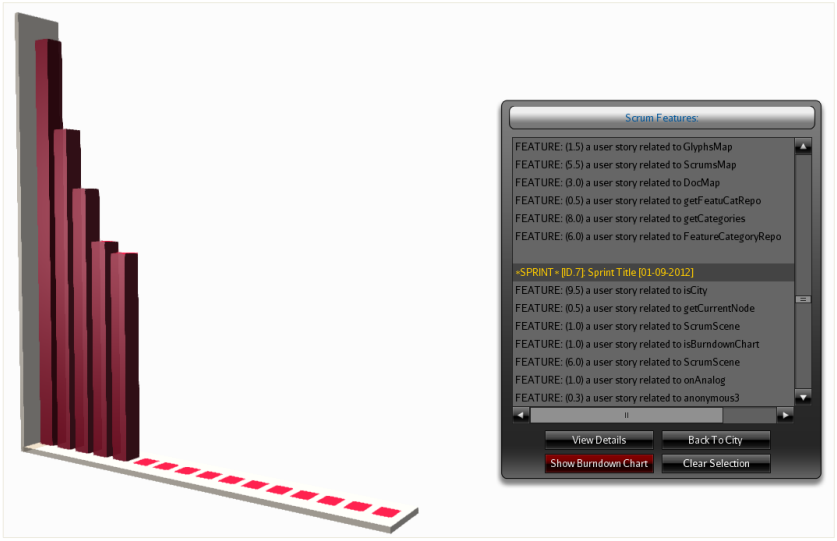


Figure 5.6: A burn-down chart of a selected Sprint

Apache's IvyDE

Description. IvyDE[1] is an open-source Eclipse plug-in developed by the Apache foundation which brings the popular Apache Ivy dependency management tool to the Eclipse community.

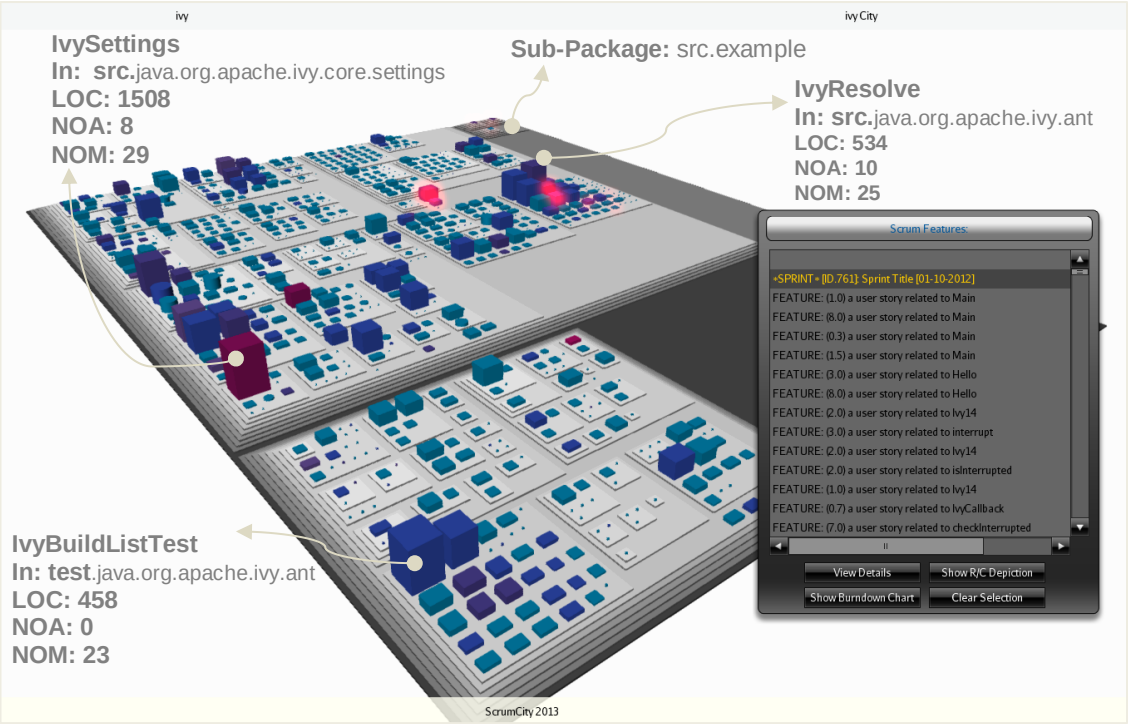


Figure 5.7: Main City Landscape of Apache IvyDE Eclipse Plugin

Visualisation. The bird's eye view of IvyDE's city landscape shown in Figure 5.7 gives an immediate visual cue about the system's structure as well as its complexity. The key feature evident in the city's visualised landscape is that it consists of two main districts (a very small third module can in fact be spotted at the furthest corner). The smaller district in the front of the visualisation is a relatively well-sized test module. The whole system consists of a total of 793 classes dispersed across 130 packages. Another interesting characteristic of this city is that most of the system artefacts are of low complexity, having low numbers of methods and lines of code counts fewer than 200. The 'IvySettings' class stands out as a central class having a LOC count over 1500 and a NOM measure of 29.

Scrum. The glowing buildings in Figure 5.7 demonstrate the advantage of identifying the exact location, in such a relatively large system, of where in the landscape a Sprint or a User Story is involved or is contributing to the system.

[1] http://ant.apache.org/ivy/ivyde/

Shrimp Suite

Description. SHriMP has already been introduced in the literature review as one of the earliest and most popular 2D software visualisation tools. Shrimp Suite[1] comprises in one system a collection of different visualisation features that were developed over time as SHriMP add-ons.

Visualisation. Figure 5.8(a) shows the structure of this system as visualised by ScrumCity (in the remaining/completed work mode). A peculiar, large rectangular patch (aptly dubbed as a parking lot by Richard Wettel in similar findings of his work) claims the city's main attraction. This feature, unseen in the landscape of the previously visualised systems, happens to be a '*Constants*' class with no methods. Four other smaller patches can also be seen in the city, which unsurprisingly, all happen to be similar '*Constants*' classes as well. Another distinctive feature of the city is the number of large skyscraper buildings that dominate its skyline. The largest of these, *PShrimpNode*, comprises a total of 157 methods and 2367 lines of code. It is also worth noting that, as would naturally be expected, buildings' colours (see Figure 5.8b) are observed to generally move up the scale as building size increases.

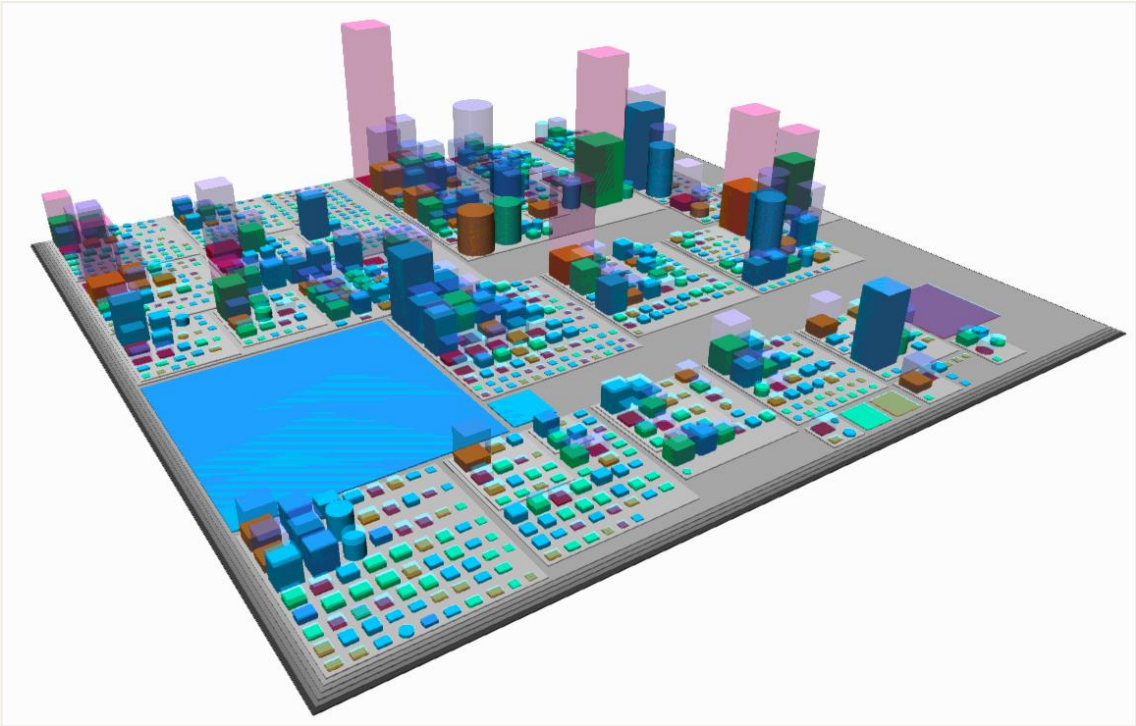


Figure 5.8 (a): a simulated view showing the progress status of a Release projected over the affected system artefacts (in this case, all artefacts are involved).

[1] http://thechiselgroup.org/shrimp-user-manual/. Code obtained from: http://sourceforge.net/projects/chiselgroup/

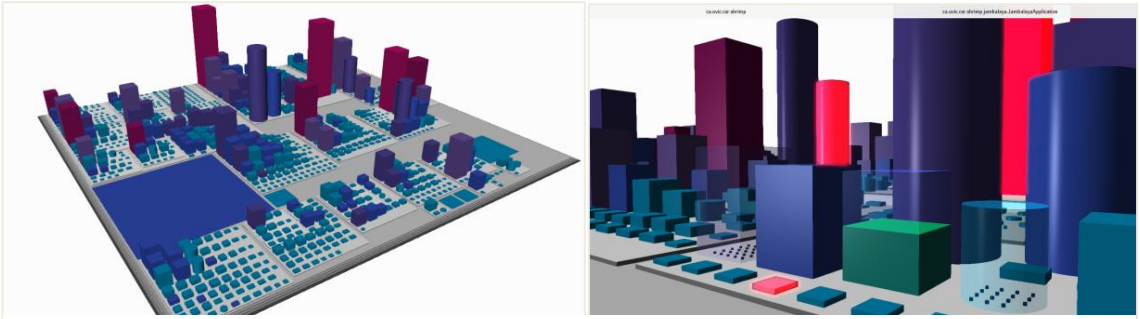


Figure 5.8: Left (b): Normal city view of Shrimp Suite showing original colour mappings. Right (c): A close-up view of Shrimp Suite city.

Scrum. Apart from providing a visual perspective of the system's structure, Figure 5.8(a) shows also at the same time a bird's eye view of the progress status of an entire release. The depiction of the *remaining-to-completed* work proportions allows the user to perceive a localised completeness level for each artefact, as well as providing a broader overall awareness of work progress over the whole system. It is important to mention that in a real-world situation, the contribution of a certain release would most probably not be dispersed over the whole system as is seen in this case of simulated data. In other words, it is unexpected that *all* system artefacts would be involved in a single release – an exception of that would be in the case of the first release being projected over the first version of the system.

In addition to the unfilled space representation, the four-colour coding scale (as defined in Chapter 4, section 4.5.6) serves to depict the *range* of completeness ratio of each artefact, particularly when viewing the whole system at once. All four colours can be seen with different distributions in Figure 5.8(a) as a result of the randomly generated data. A prevailing sky-blue colour, however, would have indicated a complete or near-complete status (>70%) of a current release. Furthermore, the exact percentage figure is displayed in the tooltip when the user places the mouse over a building (see Figure 5.9).

Figure 5.8(c) is provided to show an example of what a close-up view looks like as a user is interacting with and 'walking through' the city.

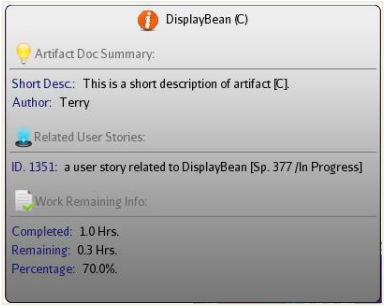


Figure 5.9: A ToolTip showing some details of a class from Shrimp Suite city.

jMonkeyEngine3 (jME3)

jMonkeyEngine3 has also been introduced above, as the 3D graphics library used in ScrumCity to render the virtual environment. It is an open-source 3D gaming library that has been witnessing an increase in popularity for the past two years due to an active community of developers. The main city view of jME3 can be seen in Figure 5.10 with 3 main districts characterising its landscape. The jme3test module stands out with an interesting appearance of structure featuring a uniform distribution of classes, each having a limited number of methods and a low LOC count. This module alone comprises 36 packages and a total of 430 classes. The main 'jme3' module, on the other hand, features a round flat patch along with a few skyscrapers of varying sizes.

As with the parking lots exhibited in Shrimp Suite, the flat round patch seen here (*KeyInput*) correspond to a '*Constants*' keeping place, except this time a Java Interface is used instead of a Class for this purpose. Other interpretations and conclusions can by drawn by a software engineer by examining the various city buildings, their sizes, colours, and distributions. Overall, the jME3 city consists of 138 districts and a total of 1249 buildings (inner methods, i.e., rooms, are exempt from this count). Execution time taken by ScrumCity to process and render jME3 city was 1 minute and 9 seconds.

Figure 5.11 shows two different views of jME3 city in Remaining/Completed work mode while Figure 5.12 shows a burn-down chart of a selected Sprint. This particular example of a burn-down chart is presented here to show that, in this case, the date range of work entry records exceeded the expected Sprint completion time set originally; hence no empty place-marks can be observed since the last column corresponds to the most recent day found in the work entries.

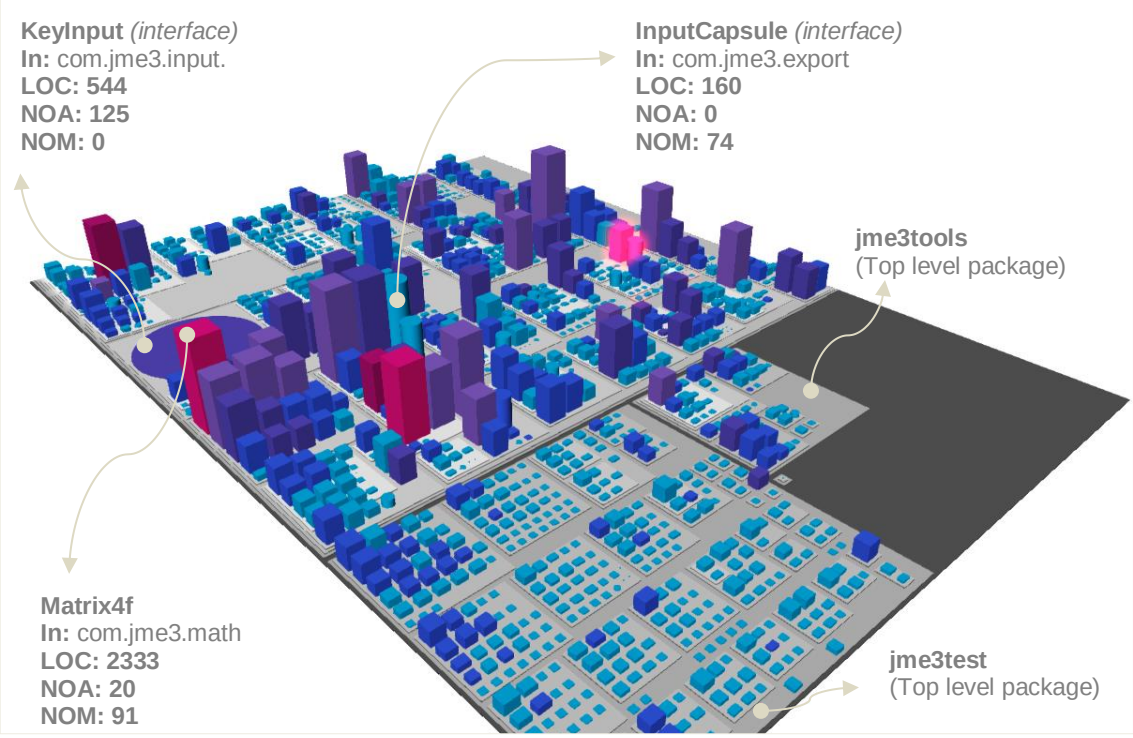


Figure 5.10: A view of jMonkeyEngine3 city landscape as produced by ScrumCity

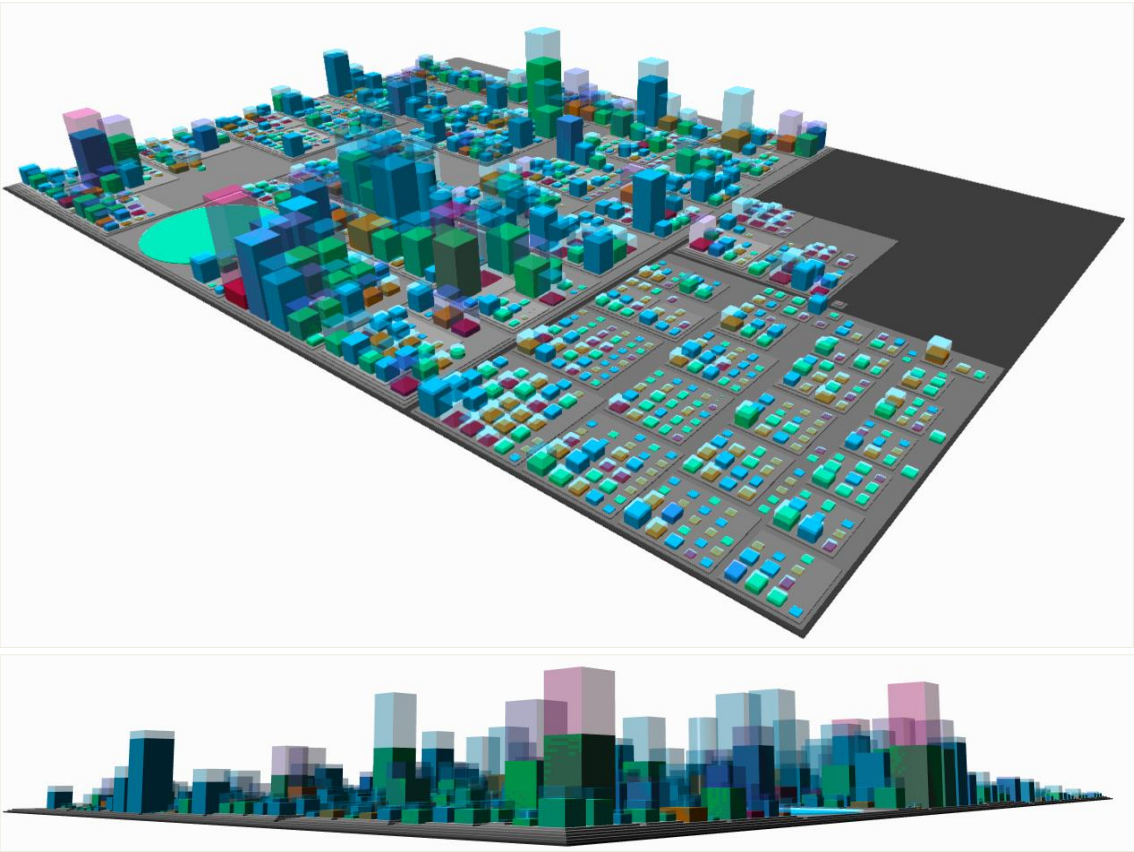


Figure 5.11: Top (a): An isometric view of jMonkeyEngine3 city in remaining/completed work mode, showing work progress of a simulated Release. Bottom (b): a side view of the same scene.

Figure 5.12: A burn-down chart from jME3 City.

jEdit

During the validation process of ScrumCity, the jEdit[1] city landscape was found to be among the most interesting as it exhibited an abnormal structure – making it hard to resist the temptation of presenting it. jEdit is a very popular open-source text editor for programmers supporting a wide range of programing languages and file formats. It consists of 58 packages and a total of 1315 classes, and took 54 seconds to be rendered by ScrumCity. Its skyline boasts a number of extremely large skyscrapers situated among mostly small and short buildings (see Figure 5.13). Interestingly, many districts feature a single very large skyscraper surrounded by other very small buildings.

This has many software engineering interpretations and most certainly would lead a manager or a software engineer to investigate the reasons behind the abnormal concentrations of functionalities in particular classes. Many of those classes are likely to exhibit the *god* or *blob* design anti-patterns and refactoring might need to be considered. Two classes particularly show LOC and NOM values on the extreme side, namely *TextArea* and *Parser*, having lines of code of 6711 and 5816, respectively. The two round patches correspond to '*Constants*' Java Interfaces, just as in the previous observed cases. Figure 5.14 shows the jEdit city from a different perspective with several buildings in different modes.

[1] http://jedit.org/

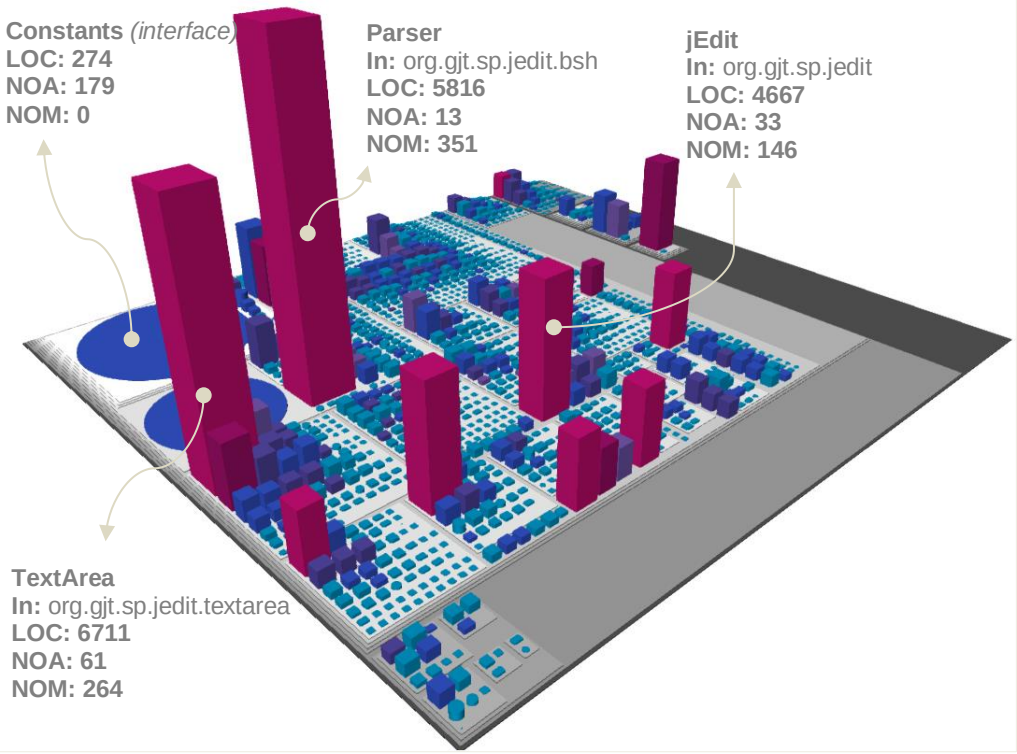


Figure 5.13: City Landscape of jEdit

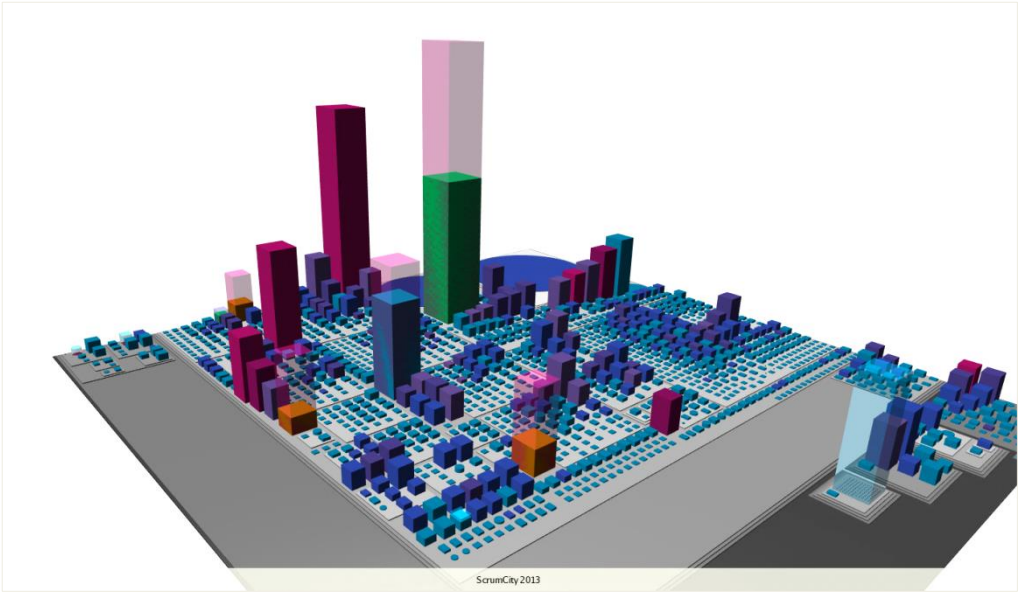


Figure 5.14: A view of jEdit City from a different perspective showing several buildings in different modes.

5.3.3 Summary of Case Studies

To conclude this section, a table summarising the subject systems' sizes and the execution time taken by ScrumCity to generate the visualisation for each is presented (Table 5.3). While performance was not formally one of the concerns of this research, presenting this information may be of interest. The execution time presented here is as experienced on the test machine described above. The figures are generally consistent over multiple runs with a noticed difference not exceeding 8 seconds. A general pattern of increases in execution time as the system sizes increase is evident; however, IvyDE was found to violate this convention (although investigating the reason behind this was deemed out of scope at this time). One last particular note with respect to performance is that due the extensive graphic processing demanded by 3D graphic tools, a level of degradation in responsiveness is encountered as system size increases. This is a common issue intrinsic to and shared by almost all existing 3D SV tools. However, the general consensus of the SV community is that machine performance as well as 3D graphics processing power are rapidly improving, both of which will consequently mitigate this intrinsic problem.

Table 5.3: Summary of subject system's sizes and execution time as experienced on the validation machine.

	System	Packages	Classes	Execution Time
1	AntViz	5	66	05 seconds
2	ScrumCity	18	115	08 seconds
3	IvyDE	130	793	3 minutes, 09 seconds
4	Shrimp Suite	43	1054	47 seconds
5	Jme	138	1249	1 min, 10 seconds
6	jEdit	58	1315	54 seconds

5.4 Discussion

This section presents and discusses various aspects of the introduced visualisation technique including the features and functionalities provided in the prototype tool. The purpose is to highlight the anticipated advantages, the enhancements to existing techniques, and most importantly, the foreseen potential applications of use. Some discussion is also presented on how particular concerns of previous researchers have been addressed by this research.

5.4.1 Potential Applications

During the conceptualisation phase of this research many potential applications were envisioned to benefit (to varying degrees) from the introduced conceptual visualisation as well as from the specific functionalities implemented in the tool. A collection of the most important potential applications is presented and briefly discussed here to highlight the tool's prospective utility.

Requirements Traceability

The major realised advantage that ScrumCity brings is the immediate visual tracing of various system components (down to class and method level) back to their original user requirements. Losing the original developer's intentions and concepts behind the code artefacts is a well-known issue in software engineering, and Petre and de Quincey (2006) expressed particular thoughts in favour of capturing these original design concepts into software visualisation. The way that ScrumCity integrates and synchronises the Scrum data with the produced code artefacts make this traceability particularly straightforward and visually evident. Different software stakeholders are anticipated to benefit from this gained advantage. A project manager, for example, can look through the list of the Scrum User Stories and select one of interest to find exactly where in the system that user story has manifested (see the previous section for examples). For a software engineer, the ability to display and read the details of the Scrum artefacts (user stories, sprints, and releases) at the scene makes it feasible to trace the different user stories and determine whether or not the intended user requirements have been correctly implemented. It can support engineers in reasoning

about and studying the system's design and architecture, since they have the opportunity to visually relate the concepts to the architecture. For a new developer joining the team, or a maintainer, they can select a particular class or a method from the scene and readily identify all the user stories that have contributed to that class or method. Some of those user stories may be bug fixes, enhancements, or simply new functionality being added. For each user story, they can then display and read its description, identify its original author, in addition to other details right on the spot, with no need to shift attention to other textual documents or to switch applications; hence avoiding mental distractions and ensuring the developer's focus remains in context. This can enable a maintainer or newcomer to be readily informed about the purpose of the artefact in question and the original concepts behind it, before they commence maintaining it or adding enhancements to it.

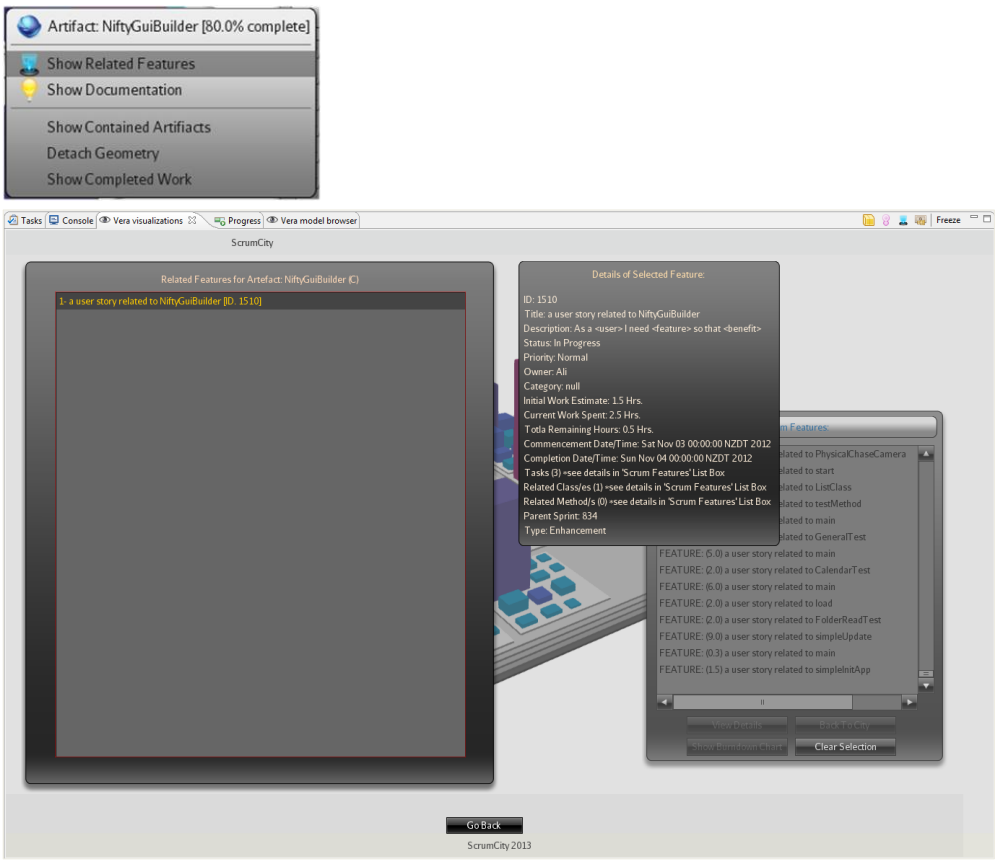


Figure 5.15: Top (a): Right Contextual menu. Bottom (b): Overlay GUIs displaying a list of related features on the left pertaining to the selected artefact, and details of the selected feature on the right.

The introduced visualisation technique makes possible two perspectives for looking at a system. An inspector can choose to focus on the Scrum artefacts (e.g., Sprints and User Stories) and has the ability to locate their exact manifestations within the system

structure. Another perspective, which is more beneficial to developers and maintainers, is to focus on particular system artefacts, and then be able to identify the user stories involved in that specific artefact. The first perspective has already been demonstrated with several examples in the previously presented case studies (the red glowing highlight of glyphs upon selection from the Scrum list). Figure 5.15 demonstrates an example scenario of the other perspective. The user right-clicks on a building (a class or a method) and chooses to display the related features (Figure 5.15a). An overlay GUI is then displayed with a list of associated user stories on the left (Figure 5.15b). Selecting a user story displays its textual details on the right. For convenience, Figure 5.9 shows that brief information of each involved user story is also displayed in the tooltip as a user points the mouse to a building. Working from this perspective, a software engineer can draw different reasoning and conclusions about the current design. For example, a class found to be involved with many features could mean that it is being continuously updated or is starting to develop into a *god* class. The *nature* of the involved features in a certain class could also enable an engineer to deduce other undesired design anti-patterns. Those two important perspectives are contended to be of real value aiding different stakeholders in their tasks.

Lastly, the *search* functionality available in the tool is considered to be an especially supportive feature for the different scenarios of the 'requirements traceability' task discussed here. In fact, Kienle and Muller (2007) reported that search functionality for textual and graphical elements was rated as the most useful functional aspect of software visualisation tools with 74% of participants in a survey rating it as 'absolutely essential'. The Kienle and Muller report was based on the electronically-conducted study of Bassil and Keller (2001) that involved 107 participants (two thirds of whom were industrial practitioners and 41 were specifically 'expert users'). Sensalire et al. (2007a & 2007b) also highlighted the need for more adequate searching and querying capabilities in SV tools.

Feature Locality

Locating specific features of interest within the structure of a system's source code (also called *Concept Location*) is a well-known challenge in software engineering. In fact, it represents one of the most carried out tasks by software developers in their day

to day activities (Kuhn et al., 2010; Xie et al., 2006). Given a particular piece of functionality (a feature), a developer needs to identify and locate the specific code artefacts involved in that functionality before they can maintain, enhance, or debug it. Kuhn et al. (2010), in an empirical pilot study of a visualisation tool called CODEMAP, found that participants actually made the most frequent (and '*more interesting*') use of their visualisation tool to complete the '*feature location*' task, more than any other task. Their pilot study involved 3 professional developers and 4 graduate students that carried out 5 program comprehension tasks (of which one was '*feature location'*) in addition to a sixth bug-fix task. In that experiment participants used nouns and verbs found in the feature description as search terms, and then used the visualisation to assess the search result dispersion. ScrumCity makes the *actual* list of features readily available and tightly integrated into the visualisation scene. It is contended that this form of presentation will be more advantageous, and should give more precise results, since selecting a feature highlights only those artefacts that were deemed relevant by the original implementer of that feature. Even though not currently implemented, adding a search mechanism to search the textual description of the features in the Scrum list is expected to further increase the value and convenience of using the Scrum list.

Monitoring Development Progress

A particularly evident potential application of use is the opportunity for managers to visually watch over and monitor the progress of development. Given the Scrum-centric nature of the tool at present, this could be specifically valuable for Scrum Masters. The visual projection of work progress over the individual system code artefacts is believed to be unprecedented. The ability to monitor the remaining work of user stories at the individual artefact level as well as across an entire system is expected to be of real value to Scrum-practicing software teams, providing a tool-supported means of viewing individual as well as team performance. The burn-down chart also contributes to this task. Since all details of features (i.e., user stories) are synchronised with the code artefacts and are made available, a manager can also access more information about any system artefact that is of special concern (e.g., displaying a low-complete

work portion) such as finding which developer is involved or reading the feature description.

This facility to pay focused as well as broad attention to the progress status of development has actually been discussed by (Ghanam & Carpendale, 2008), who emphasized that it is important to developers, maintainers and managers, and that software visualisation tools are particularly suited to providing this sort of functionality. While managers need a high level overview to determine the completion of development goals, developers and maintainers need to know the most recent status of particular artefacts so they can continue the development process.

Projecting Scrum Data over Different System Versions

Another conceived application of potential interest to software project managers and engineers is the result that can be observed by having a *specific* Scrum release projected on *different* versions of the system. Taking an early release and projecting it on *various* late versions of the system would reveal potentially insightful views exposing the change of locality of Scrum artefacts as the system has grown. Taking it from the opposite perspective, *multiple* Scrum releases can be projected (one at a time) over a specific late version of the system exposing visually how the different releases have contributed to the system. This can be valuable, for example, in terms of understanding the effect and impact of the different releases – and their parameters (e.g., tools used, practices employed, developers involved) – on the evolving structure of the system. This potentially interesting observation unfortunately could not be demonstrated here due to the fact that it requires real Scrum data which was not available within the scope of this research.

Studying, Exploring, and Discovering

The presentation of all three aspects of software – the product's visual structure, the development processes (Scrum), and the code artefact documentation – synchronised and integrated together in one place, is expected to provide multiple benefits for different groups of stakeholders. It brings into one place, and in an aesthetically appealing and accessible way, all the information required for a potential user who needs to gain some knowledge about the system. It is asserted here that stakeholders

new to the system, particularly new developers, would find the tool valuable in terms of being 'introduced' to the system and in exploring its structure and artefacts. Focusing on a specific artefact, they are able to view and read the *features* that this artefact implements as well as the *documentation* (see Figure 5.16) about this artefact. Again, the artefact search mechanism, as well as the non-distracting *in situ* GUIs for displaying textual information, are conceived to also be helpful for such tasks.

One of the main drives behind software visualisation, as discussed very early in this thesis, is to depart from the textual(-only) dimension and take full advantage of new visualisation technologies. Instead of being presented with large textual documents to learn a new system, a newcomer can be instead presented with a fully interactive visualisation environment with documentation, user requirements, and software structure all integrated and unified in one place. In addition to making the learning experience more interesting, it should also significantly reduce the cognitive load faced in the traditional way of learning about new systems.

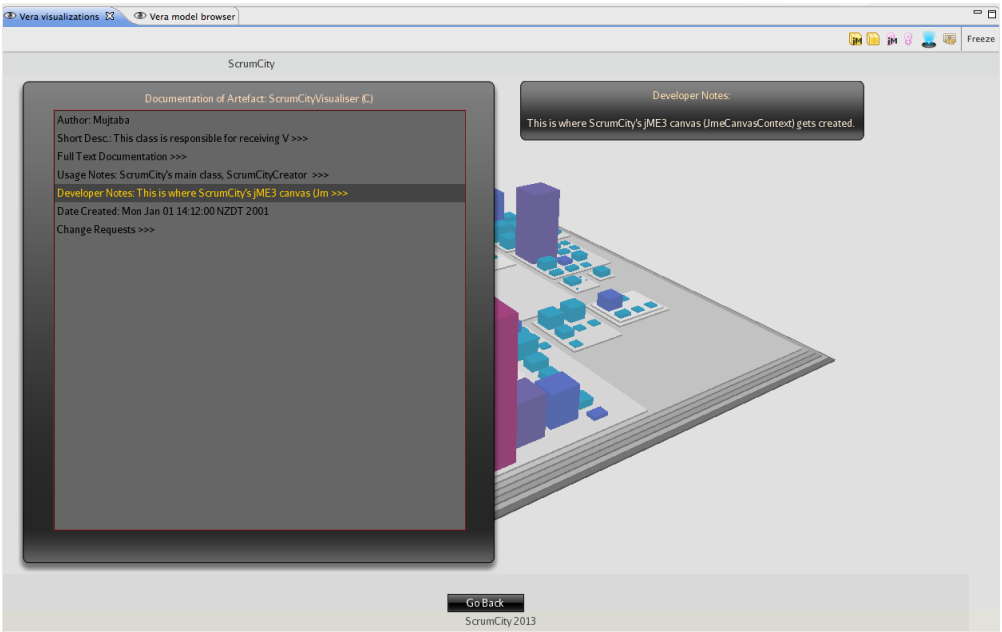


Figure 5.16: Example of the in-situ presentation of artefact documentation.

This on-demand presentation of meta-data and details has been highlighted by many researchers as an important feature that should be incorporated in software visualisation tools to render them more useful (Beck & Diehl, 2010; Petre & de Quincey, 2006; Sensalire & Ogao, 2007a, 2007b; Storey et al., 2005). Storey et al. and Sensalire et al. in particular reported that requirements documentation and informal

code comments were found to be very important to software stakeholders (based on studies aimed to find requirements that lead to supportive visualisation tools), yet almost all recent software visualisation tools have failed to incorporate these valuable information sources. (Ironically, one of the earliest visualisation tools, SHriMP, provided contextual integration for traditional HTML documentation.).

Stakeholder Communication

Another potential application of the introduced conceptual visualisation technique is in supporting information *communication* between different software stakeholders. The fact that this visualisation brings all three major aspects of software together in one place, as discussed above, is thought to make it particularly suitable for this purpose. Since this aggregation of information comprises various forms of knowledge important to different stakeholders, it deems the visualisation tool as being potentially useful in meetings and group sessions for reasoning and analysis. It can bring managers, engineers, designers, architects, and developers 'under one roof'. The 3D visual representation of the software structure could even prove to be of particular marketing benefit to potential customers. Further, the remaining/completed work view may be suitable to visually demonstrate the current status of a project to the software owners. The potential of software visualisation as being a suitable mechanism for communication between and among this widely varying range of audiences has already been conceived by other researchers (Boccuzzo & Gall, 2008; Ghanam & Carpendale, 2008; Parnin & Görg, 2007) and it is anticipated that the techniques introduced in ScrumCity make this particularly feasible.

5.4.2 Enhancements

Apart from introducing and then implementing the conceptual visualisation technique, this research was also focused on introducing some specific enhancements to 3D software visualisation and to addressing some concerns and issues highlighted by previous researchers. Table 5.1, presented at the beginning of this chapter, summarised the concerns and/or recommendations deemed to be most prominent in the software visualisation literature (or 'continuing issues' as Petre and de Quincey

(2006) referred to them). As mentioned above, this set of 'desired features' has been given significant consideration during the course of this research. The *system features* section of the previous chapter (section 4.5) presented how ScrumCity has addressed some of those issues. This section elaborates on some of these aspects, particularly in order to relate them to literature and previous developmental research.

Normalised City Metaphor

Table 5.1 shows that 'good use of visual metaphor' and 'metaphors that are resilient to change' are among the major concerns in software visualisation. The city metaphor version introduced by Wettel and Lanza has been discussed as a very promising 3D metaphor that has been empirically validated to support software comprehension and to reduce cognitive load, hence it was specifically chosen in this research. Moreover, it was also mentioned in Chapter 4 that some specific enhancements were introduced to the metaphor to make it more suitable for the purpose of this research. A major issue with the Wettel and Lanza metaphor, which was criticised by some researchers (Caserta & Zendra, 2010), is the fact that it produces software cities with unrealistic appearance. In the Wettel and Lanza metaphor, the NOM metric is mapped to a building's height while the NOA is mapped to a building's width. This can lead particularly to two irregularly shaped buildings that dominate the city rendering it unrealistic. Classes with high NOM but very low NOA appear extremely thin or '*needle-like'*. On the other hand, buildings with low NOM and low NOA appear as '*dot-like',* making them hard to distinguish individually. The end result is a city with '*very diverse building shapes*' that, according to Caserta & Zandra, works against the gestalt principle, which states that humans can often distinguish 4 to 6 different shapes efficiently at one time. Figure 5.17 shows a view of the argoUML system as visualised by Wettel's CodeCity tool.

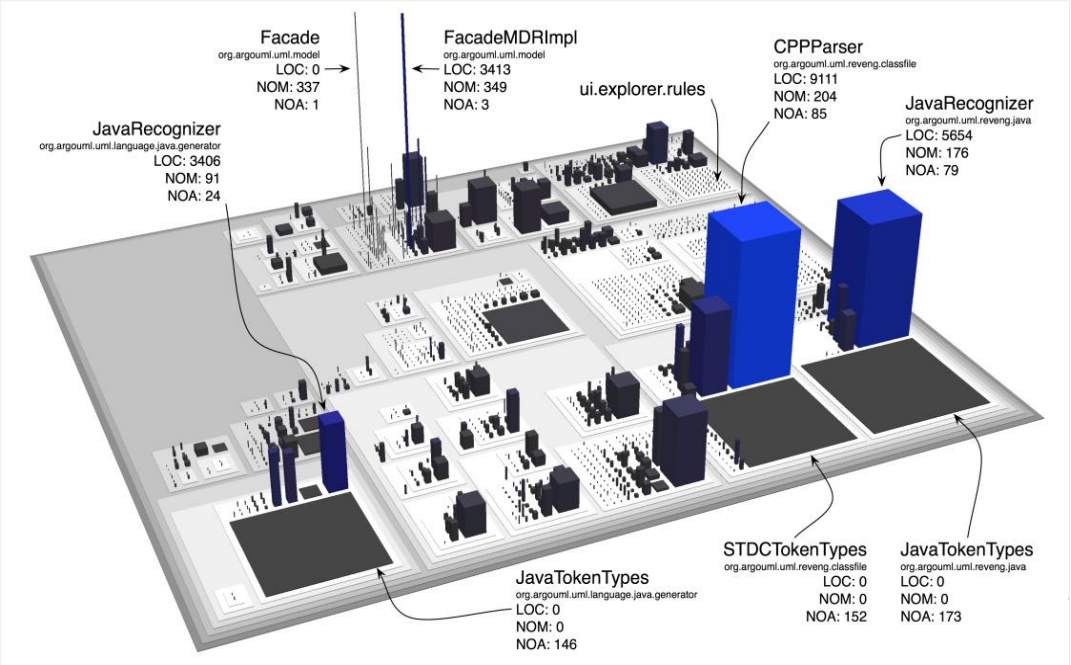


Figure 5.17: A visualisation of ArgoUML as visualised by Wettel's CodeCity tool, obtained from (Wettel, 2010).

Wettel actually recognised this issue in his PhD thesis and tried two different strategies to restrict the buildings' sizes, but after some evaluations concluded that leaving the metaphor as is produced the best results.

Improvement

In this work, the metaphor has been slightly modified in two aspects. First, methods had to be visualised to meet the purpose of this work and so they were rendered in the naturally expected way, i.e., as small rooms *inside* the building of their parent class. The same layout mechanism used to render classes on top of their parent packages was used to render the methods, but instead they were rendered inside their classes. Second, even though the width of a class' building was initially mapped to the class' NOA metric, the class' building width was later allowed to expand to accommodate the methods (rooms) that were rendered inside of it.

Unexpectedly, this strategy has proved to consistently result in a more uniform city that has a more realistic landscape and appearance than Wettel's original city metaphor. The number and diversity of visualised systems presented in the case studies were chosen in part to demonstrate and attest to how the enhanced metaphor employed in ScrumCity produces neither the 'dot-like' buildings nor the 'needle-like'

buildings; making the overall appearance of the resulting city being much closer to 'reality' with realistic building sizes dominating the landscape.

In addition to the advantage of representing methods, this strategy also has some other benefits, as follows.

Drawing attention to what is significant

For software engineers or any software stakeholder, a low NOA is not normally considered to be a characteristic of concern for a class. Hence, allowing a building to be tall but extremely thin just to reflect the fact that it has a very limited number of attributes, not only falsely draws viewers' attention to a non-significant attribute, but has the drawback of not communicating the actual *largeness* of a class. A class with over 100 methods is a large class that stakeholders would normally be concerned about, irrespective of it having few attributes. So by visualising it as a needle-like building, its importance is falsely downplayed and belittled. Even worse, if the class does not happen to be especially tall compared to its surroundings, it becomes hard to spot and distinguish. Interestingly, and in stark contrast to Wettel's original metaphor, the recently released Sonar plug-in 'City Model' that was introduced briefly in Chapter 2, completely dropped the NOA metric from their metaphor mappings, thus discarding the opportunity to bring to viewers' attention some important characteristics of software, such as the parking lots that correspond to large constant classes.

The modified city metaphor introduced and used in ScrumCity is considered to not only make the city look more uniform and realistic, but it makes sure only those artefacts that have significant or peculiar characteristics stand out among the other artefacts. As the number of methods in a class grows, the class' building not only increases in height, but also correspondingly increases in width – giving the building a more realistic volume reflecting the class' actual large size. Yet, if the number of attributes were significantly high, the class' building will in that case have an abnormally and immediately noticeable width, creating the rectangular or round flat patches demonstrated previously. Thus in this way, this strategy avoids the unrealistic and irregular shapes, creates a much more uniform landscape, and viewers' immediate attention is drawn to what is truly significant or to what is commonly considered to be of particular importance.

Another advantage of this strategy – albeit a more mechanical one – is that the needle-like and the dot-like buildings are difficult to interact with practically inside the 3D virtual environment (e.g., by right-clicking or hovering the mouse to view a tooltip). Thus by avoiding those irregular shapes in ScrumCity, all buildings in the city visualisation become both noticeable and practical to interact with.

Since this enhanced version of the city metaphor avoids the irregularities found in Wettel's original city metaphor, it was hence dubbed as a 'normalised city metaphor'.

Addressing Navigation Issues

As discussed in the literature review chapter of this thesis, navigation in 3D virtual environments is a major issue in software visualisation, with many researchers highlighting it as a core concern requiring proper attention before 3D software visualisation can enjoy practical use in the SE industry. As a result, special attention and effort was paid to navigation while developing ScrumCity, in the hope of attaining an acceptable result.

The standard navigation mechanism commonly found in 3D software visualisation tools is the conventional WASD key combinations as described in Chapter 4. The 3D software visualisation tools surveyed during the course of this research principally offered this conventional WASD navigation mode; with very few offering other mechanisms such as rotation around axes. The Manhattan Eclipse plug-in, for example, offers an '*orbital*' mode in addition to the WASD mode, which is a positive enhancement, but the tool still suffers other navigational limitations that hamper users' freedom to 'walk through' and explore the virtual city. Enhancements were still needed in order to provide users with a better navigational experience.

In order to support navigation in ScrumCity, different solutions were initially experimented with to find out those that gave the best results. This led eventually to a *fly-camera* mode being combined with the standard WASD navigation mode. This special configuration allows the user to optionally use both the *mouse* and the conventional WASD keys *simultaneously to walk-through and explore the city,* resulting

in what is believed to be a smooth navigation experience (particularly if a modern multi-touch desktop screen was in use, see Chapter 6, section 6.5).

As presented above in Chapter 4 section 4.5.9, some other enhancements were also added to the default WASD mode which included the ability to rotate the city around the y and x axes, upward and downward elevations, and drag-&-pan and slow zooming using the mouse device. The accompanying video demo demonstrates this enhanced navigation experience.

Utilising Animation

Animation in the 3D virtual environment is a potentially very supportive feature that should be utilised appropriately in software visualisations. Caserta and Zendra in their 2010 literature survey *'Visualisation of the Static Aspects of Software'* noted that utilisation of animation in 3D software visualisation not only makes the visualisation enjoyable – which is a characteristic that many researchers believe to be strongly desirable (Bassil & Keller, 2001; Kienle & Muller, 2007) – but it also has cognitive and perceptual benefits if used well. In spite of this, they concluded that animation has rarely been utilised in visualisation tools. Other researchers that highlighted the lack of animation use in software visualisation tools include Sensalire et al. in their highly-cited 2008 paper titled '*Classifying desirable features of software visualization tools for corrective maintenance*' and Storey et al. in their 1997 paper – demonstrating that this absence is long-standing. In ScrumCity, as introduced in Chapter 4, a feature to automatically move the viewer to a target building (in the case of a successful search result or a relationship match to a selected feature) has been implemented using smooth animated transition. According to the findings of early researchers, such an approach is believed to significantly help the user to mentally relate to the part of city landscape (which resembles the actual software structure) in which the artefact in question is located. Instantly showing the target glyph, on the other hand, would leave the viewer completely unaware of the relative artefact location until they zoom out of view, putting unnecessary overhead on their perception. Such utilisation of animation, as far as the literature review in this research could ascertain, has not been witnessed in previous 3D visualisation tools.

5.5 Summary

This chapter has presented some of the contributions to the software visualisation field anticipated from this research and has specifically highlighted how different aspects of the research relate to prior literature and attend to the calls and recommendations of previous researchers. Case studies were also presented and discussed using real-world subject systems in order to demonstrate and evaluate the utility of the devised tool.

6

Summary and Conclusion

This chapter comes as the end and culmination point of this research journey. It presents a summary of the main milestones and highlights major accomplishments as well as the anticipated contributions to the body of knowledge of the Software Visualisation domain. Some encountered research difficulties and limitations are also acknowledged, with recommendations and directions to future research being finally suggested.

6.1 Summary

Before a conclusion to this research and its key contributions are presented, this section provides a concise account of the previous chapters in this work.

The introductory chapter opened with a brief account of the software visualisation field, what it is, the nature of problems it deals with, and how it is deemed useful to various stakeholders in the software development industry. The lack of attention directed towards development processes in current software visualisation techniques was then identified and highlighted, and it was pointed out that most contemporary techniques focus merely on re-presenting a system's source code (when considering software structure visualisation specifically). Similar concerns raised by previous researchers, emphasising the field's failure to address the '*process'* aspects of software in SV techniques, were also discussed. In particular, work by Petre and de Quincey (2006) that called for the representation of developers' original intentions, rationale, design concepts, and activities underlying individual software artefacts was given special attention, laying the ground work for the introduced *Conceptual Visualisation*. The popular and widely adopted agile development method of Scrum was then introduced as being particularly promising for integration into software structure visualisation –given excellent alignment with the requirements of the conceived Conceptual Visualisation. The novel integration and synchronisation between the Scrum artefacts and the software artefacts, both contextually visualised within a single software structure, has been deemed to potentially benefit and inform a range of software tasks and activities.

In Chapter 2, prior research in software visualisation was introduced, focusing particularly on 3D software visualisation of system structure. The importance of finding effective visual metaphors to employ in visualisation techniques was discussed, highlighting its role and significance in leveraging human perceptual skills and, therefore, amplifying cognition and aiding comprehension. The major early metaphors used in software structure visualisation were also presented along with reviews of some tools that had implemented them. Several previous works that were of particular relevance to this research were discussed, emphasising the software tasks and activities that those approaches were intended to support, and what elements/aspects

of software were involved in those approaches. From that basis, the stance and position of this work in the context of other research in the field were explained.

Chapter 3 focused on the research methodology adopted in this research - the *Design Science* methodology. A detailed discussion and analysis was provided setting out the distinctive characteristics, requirements, and expectations that shape the design science (or systems development) paradigm of research and that set it apart from other behavioural and natural science research paradigms. Key papers related to research methodologies in IT/IS were reviewed in this regard, including Hevner et al. (2004), Markus et al. (2002), March & Smith (1995), and Nunamaker et al. (1991). The problem space of this work was then outlined in the context of earlier research and the major objectives and goals were highlighted. The chapter was concluded with detailed descriptions of the specific design steps as well as the broader methodological approach within which this work was conducted.

The development process and any resulting prototype tools are considered to be important aspects of design science research that each contribute to the knowledge base of the field and its ongoing progress. Chapter 4 therefore presented detailed descriptions and technical explanations of the novel system architecture as well as the design of the proof of concept tool, ScrumCity. Most importantly, the design discussion included details of the introduced *conceptual visualisation* technique, the *Scrum Artefact* to *Software Artefact* mapping mechanism, and the data model that enabled this mapping. Relevant details of the important aspects of the implementation were also introduced in this chapter. This technical information was provided not only to provide better understanding of the visualisation technique and how it was implemented in ScrumCity, but also in the interests of prospective researchers who might wish to build on top of the introduced technique or take it into further development and enhancement. The chapter concluded with pictorial representations of the tool's major functionalities and features.

Finally, Chapter 5 began with a tabular summary of the major issues and concerns that face present day software visualisation research, drawn and extracted from recent key literature and survey studies. Those identified '*desired features*' (as they were called by some researchers) stood as significant goals and ambitions driving the tool and the

technique development strategy in this work, and thus could be viewed as broad criteria against which this work could be assessed. ScrumCity was then validated using case studies of six real-world systems of varying sizes in order to demonstrate the introduced visualisation technique 'in action' across those different real systems. Main usage scenarios were illustrated and described using those case studies. The chapter then presented some potential applications of use, highlighting a range of software tasks and activities that are anticipated to benefit from the devised technique; the most significant of which are requirements traceability, feature (or concept) location, design reasoning, and stakeholder communication. Additionally, some of the key contributions of this work pertaining to improvements to the visual metaphor adopted and some user experience enhancement techniques were also revealed in this chapter and were informally evaluated and contrasted against existing approaches.

6.2 Conclusions and Contributions

This research work began with a number of motivations that shared the common ambition to contribute to the advancement of the field of software visualisation. Specifically, this work has focused on a particular category of software visualisation, which is the 3D visualisation of the static structure of software in virtual environments. Software visualisation was identified from the outset as a domain possessing great potential for research and industry and one that has experienced an increase in popularity during the past decade. This work has reviewed a considerable number of existing research works in order to identify areas of interest that are lacking attention, to gain insight on the findings and recommendations of prior researchers, and to find guidance to potential directions of research. Two aspects of SV were found to have received a considerable lack of attention, an issue that late researchers have started to emphasise: the need to better adapt this new technology to the practical/real-life requirements of its potential users (hence solving the 'lack of adoption' issue); and, with equivalent importance, the need to explore and discover the seemingly vast potential applications of use and to then develop novel approaches to exploit these opportunities using SV tools.

This research thus identified that development processes had been largely neglected in the existing literature, which had been focused on materialising code structure only. Just a few works had tried to augment such visualisations with external information, such as data extracted from versioning repositories, to render the tools more useful.

The key contribution of this work lies in the novel incorporation of the Scrum development processes into existing visualisation techniques. Inspired by the calls of prior researchers, integration of the Scrum processes into such visualisation merges and unifies the *concept* with the *product*, which according to prior literature, is strongly desired but has so far not been achieved. Furthermore, the specific linking and synchronisation mechanism introduced here takes advantage of the visual decomposition of a system's structure to make the concept contextually available to its related product. The importance of utilising this decomposition structure to represent other aspects of information is further supported by the recent experiment of Kuhn et al. (2010) (discussed in Chapter 2) that revealed that developers of all levels subconsciously construct a mental model of a system based on its package structure to help guide their daily activities. This work has also revealed several software tasks and activities that are conceived to benefit from this novel visualisation technique.

Other contributions of this work have also been highlighted and discussed in previous chapters, and are summarised as follows:

- An enhanced version of the city metaphor 'named *Normalised City Metaphor*' that has been shown to result in a more uniform and realistic landscape of a visualised software city.

- The introduction of an XML Schema for representing a Scrum Data Model. As stated above, Scrum practice in the agile community has a universally agreed-on *de facto* model when it comes to representing data, but a formal standard for that model does not exist. It is thus hoped that the XML Schema model introduced in this work will contribute to a standardised format for representing and exchanging (at least at an organisation's systems-level) Scrum data in the future.

- The prototype tool, which features a novel approach making available three important aspects of software – its structure, its user requirements, and its documentation – all integrated and presented in one place using non-intrusive and non-distracting displays.

- Novel techniques implemented in the tool aimed at enhancing the user experience in terms of navigation and usability.

The next section summarises the key implications for practice that arise from the devised visualisation technique.

6.3 Implications for Practice

Software comprehension is the fundamental driver behind all software visualisation research. The problem space for software visualisation research is thus oriented to identifying the aspects of software tasks and activities that are likely to gain benefit from this technology, and then finding or developing suitable techniques and tools to support those tasks. Consequently, this work has some specific conceived applications and foreseen implications for the software engineering community. Chapter 5 (section 5.4.1) has presented detailed discussions on those anticipated potential applications and has also highlighted prior research in the field that called for or conceived similar applications. A brief summary of those potential applications of use is presented here for the reader's convenience.

Requirements Traceability: The introduced conceptual visualisation approach captures the original user requirements, intentions, and concepts, and explicitly links and synchronises them with related individual system components before making them contextually available to the user projected over the visualised software structure. This enables immediate and direct traceability from original user requirement to the produced system artefacts, in both directions.

Feature Locality: The captured user requirements (user stories) inherently represent the system's features, so the synchronisation of artefacts enables users to search for and/or readily locate the system artefacts involved in a particular feature in question.

Monitoring Development Progress: The novel approach of visualising developers' remaining vs. completed work projected over the software structure has the advantage of enabling stakeholders to monitor development progress at both the individual artefact level as well as the overall system level.

Projecting Scrum Data over Different System Versions: The introduced visualisation approach is considered to be potentially useful for system design review and inspection purposes. This can be realised by visualising a given Scrum Release over different later versions of software, or by visualising a given later version of software and then inspecting and studying the contribution and evolution effect that different Scrum Releases impose on the system.

Learning and Design Reasoning: The presentation of a system's user requirements, documentation, and software structure in one integrated environment is considered to be particularly suitable for learning about a new system (for newcomers) and for analysing and reasoning sessions (for developers and software engineers).

Stakeholder Communication: The visualisation of system structure, user requirements, and development progress unified in one place has a realised potential for information communication for various stakeholders including teams group meetings, managerial and, potentially, customer meetings.

6.4 Research Limitations and Difficulties Encountered

During the course of this research various difficulties were encountered, many of which were associated with the technologies involved. Some limitations have also confined the reaches of this work and have restrained some desired achievements. This section introduces and discusses some of those confining elements.

6.4.1 Real-world Scrum Data

This work, as will now be evident, relies extensively on *Scrum Data* as it plays a key role in the introduced conceptual visualisation approach. Even though the developed proof of concept tool was validated using real-world systems, the Scrum data projected on those systems was fictional. It is evident that real-world Scrum data belonging to the actual system being visualised would instead be strongly desired. It would be particularly interesting from a research point of view to actually study, examine, and analyse the resultant visual scene, and the effect and behaviour from applying this novel visualisation technique on real-world Scrum Data mapped and synchronised over the actual system that the Scrum activities produced. It would enable, probably for the first time, development stakeholders to visually see how the user requirements (user stories) map collectively to the evolution of the system. It is believed that various findings and results will emerge visually to the surface, but which have previously existed only in the minds of the developers (some such expected results were discussed in Chapter 5). With only a very slight modification to the currently used Scrum management tools, team members could see their exact contributions and their locality within the software structure, and how they relate to other developers' work. Engineers could explicitly trace all user requirements and their direct effect on the system. In order to realise this, and to study the real-world practicality of the work, effort is needed to engage a Scrum-practising organisation to collect their Scrum data over a period of time (several Sprints, for example) and then have that data visualised on their developed system. Unfortunately, this is beyond the scope and resources of this research work. Moreover, use of the simulated Scrum data enabled initial insights to be gained and was sufficient to demonstrate the feasibility of the approach.

6.4.2 Empirical Evaluation

Pertaining to the same area of concern just described is the desire for empirical validation. Wettel, in his PhD thesis, stressed a critical issue in regard to expectations for experimental validation in software visualisation research. He highlighted major distinctions between the software visualisation domain and other software engineering domains, being the lack of a definite problem space, the intrinsic explorative nature of software visualisation, and the fact that software visualisation effectiveness is measured by the extent of the scaffolding that it can provide to human cognitive skills and perceptions and therefore the assistance it provides for different software comprehension tasks. In other words, many other software engineering disciplines have benchmarks against which a research result can be compared or for which their problems can be clearly defined and objectively measured. Software visualisation, on the other hand, is highly and inherently subjective with its major problem being cognition and knowledge amplification. This implies that a sound and defendable experiment in this field requires extensive effort to design '*comprehension tasks and activities*' based on which the experiment is then carried out. The design of such tasks not only involves knowledge of software engineering, but also of cognition theories. For these reasons, preparing and designing experimental studies in software visualisation research is a challenging and time-consuming task. Taking into consideration the 'tool design' or 'Design Science' nature of this research, time constraints rendered such preparation of a well-designed scientific experiment beyond the scope of this work. Unsurprisingly, case studies are the most commonly found form of validation in software visualisation research, with solid empirical studies being noted by many researchers as significantly lacking compared to other SE fields. However, the fact that major parts of this research were aimed at addressing and fulfilling a set of 'desired features' and 'recommendations' identified by previous researchers (some of which were based on user studies and surveys) is thought to give this research rigor at least in this respect.

6.4.3 Difficulties Faced

Choosing the appropriate technologies to implement the proof of concept tool was a major challenge in this research. This was due in particular to the desire to address the

navigation issue in 3D software visualisation. As has been mentioned briefly above, the X3D[1] library (an extension of the old VRML97 3D modelling standard) appeared particularly promising. It was lightweight, based on XML, provided most of the desired features, and was highly interoperable in the sense that it can easily run on standard web browsers with the help of plug-ins. Moreover, a Java toolkit for it (Xj3D[2]) existed that supposedly enabled it to be integrated into the Eclipse development environment. In addition, a particular plug-in (BS Contact[3]) that supported the X3D standard also supported a 3D mouse device (SpaceNavigator[4]) which was seen as providing a useful opportunity to address the navigation issue in the 3D environment, especially given that such devices have not been previously employed in software visualisation studies. For these reasons, extensive effort was initially spent with the intent to base the ScrumCity tool design on X3D. Unfortunately, the Xj3D toolkit proved to suffer from a serious lack of documentation, and the documentation that was found to be available was obsolete and outdated compared to the available version of the toolkit. Integrating the Xj3D toolkit with the Eclipse development environment was therefore never accomplished. This was a major difficulty encountered during this work.

After evaluating other potential full-fledged libraries, the jME3 library was chosen (as was introduced in Chapter 4). jME3 proved to be very suitable, with comprehensive and up–to-date documentation resources, a very active community of developers, and, most importantly of all, an extensive set of features and functionalities. The integrated third party GUI library called Nifty[5] was particularly promising since it enabled the implementation of the overlay graphical user interfaces that were seen as a principal means to provide the desired user interactivity and *in situ* textual information in a non-distracting mechanism. However, use of the Nifty GUI library was not totally problem free. During the course of the prototype development, the available version (v1.3.1) suffered some bugs and technical problems (see Chapter 4, section 4.5.3) that to some extent affected the prototype tool. The major problems noted are:

[1] http://www.web3d.org/x3d/
[2] http://www.xj3d.org/
[3] http://bitmanagement.de/en/products/interactive-3d-clients/bs-contact
[4] http://www.3dconnexion.com/products/spacenavigator.html
[5] http://sourceforge.net/projects/nifty-gui/files/nifty-gui/

- A mouse event handling bug that appears intermittently where the jME3 scene graph competes with the Nifty GUI in consumption of mouse events. This renders interaction with the GUI to be improper in some random cases where the mouse for short periods appears to be locked by the scene or by the GUI.
- A TreeBox user control was available that was intended to be used for displaying the hierarchically-structured Scrum Data Model, but unfortunately, mouse events were not properly handled by the tree-structured list. For this reason, the simple ListBox control was used instead (as explained in Chapter 4, section 4.5.3).
- Lastly, even though Nifty supposedly provided one of the most customizable user interfaces for 3D libraries, some issues were encountered in changing default text colours and getting text to be wrapped in some cases.

While these technical issues certainly affected the usability of the developed prototype tool, they nonetheless did not hinder the demonstration of the introduced conceptual visualisation technique. Based on the highly active developer communities of both jME3 and Nifty, it is strongly believed that such issues will soon be addressed in future releases.

6.5 Future Research

As was stated at the beginning of this work, software visualisation is a relatively young discipline compared to other software engineering fields, and even though it has witnessed in the past decade an amazing proliferation of research work and new advancements, it still has a long journey in front of it before it begins to gain similar standing to other fields of software engineering. Leveraging awareness and comprehension are the fundamental drives behind SV research, but the implications for practice appear to be limitless. This is unsurprising since comprehension and awareness underpin almost all activities in software development. Furthermore, software visualisation has a close affinity with technology (particularly graphics technology) and therefore as technology advances, new approaches will inevitably appear that seek to utilise those technologies. With that being said, it is important to emphasise that while *representation techniques* have received much of the attention

of existing SV research, far less attention has been directed toward exploring the *potential applications of use* and how to provide support for these applications in existing visualisation techniques. Of similar importance is studying how visualisation tools can be brought closer to the industry and potential users. SV literature is affluent with potentially useful techniques but many of those techniques have never seen the light of practical use, due mainly to the lack of adequately supported tools that implement those techniques.

To conclude this thesis, some anticipated and (other envisioned) future improvements to the developed prototype tool are summarised here.

At present, ScrumCity has the capability to load only a single version of a system (as an Eclipse Project) and also a single Scrum Release that will be projected on that version of the system. It is strongly desirable that accommodation for multiple Scrum Releases and multiple versions of a system is provided in future releases of ScrumCity. This has the potential to render the tool even more useful, enabling users to gain insights on, monitor, and reason about how the various Scrum processes are collaboratively affecting the gradually evolving system, and allowing users to visually examine this evolution process in a way that was not previously possible.

New enabling technologies such as 3D mouse, multi-touch screens, and large display screens are believed to greatly support the usability of any 3D visualisation technique, and hence increasing its reach to the public. In this regard, there is strong potential to bring 3D mouse and multi-touch technologies to ScrumCity. The 3D graphics library used to develop ScrumCity (i.e., jME3) already provides support for multi-touch screens. Potential 3D mouse devices are also currently available in the market (although their support for jME3 could not be confirmed at this time). It would be strongly desired to test and study the impact and behaviour that these technologies could bring to ScrumCity (or any 3D SV tool, in this regard) in terms of navigational enhancement and improved user experience. Since ScrumCity is an Eclipse plug-in, it would be particularly interesting to SV research to test and evaluate it on a Windows 8 machine equipped with a multi-touch screen. Users would be able to manipulate and interact with their 3D software city in the virtual environment using both hand gestures as well as the traditional mouse device.

A last desired improvement to ScrumCity relates to the mechanism behind acquiring the system artefact documentation. Due to time constraints, the current mechanism for integrating system documentation into the visualisation relies on reading it from a user-provided XML file (which must conform to the specifically designed XML schema). A better approach that is certainly more practical would be to extract the in-code developer comments instead. This in fact aligns well with the fundamental motivations and drives behind the *conceptual visualisation,* one of the main objectives of which is to make available all developer's intents and rationale behind individual software artefacts. Those informal documentation sources (or *meta-data,* as some researchers refer to them) capture knowledge that is likely to be particularly valuable to other developers and maintainers and thus would be especially useful for users who would utilise the visualisation to learn about a new system. As stated in the foregoing discussion, this issue has been identified by previous researches who considered it an important aspect that future software visualisation research should address (Beck & Diehl, 2010; Burch, Diehl, & Weißgerber, 2005; Sensalire & Ogao, 2007b; Storey et al., 1999; Storey et al., 2005). While the current implementation in ScrumCity is thought to serve well to illustrate the concept, the design of the integration and presentation mechanism makes it also relatively straightforward to simply switch to another source of system documentation when it is available.

References

Alam, S., & Dugerdil, P. (2007a). *EvoSpaces: 3D Visualization of Software Architecture*. *Database*. Retrieved from http://www.myblogmap.de/diplom/literature/paper_09_-_EvoSpaces_3D_Visualization_of_Software_Architecture.pdf

Alam, S., & Dugerdil, P. (2007b). Evospaces visualization tool: Exploring software architecture in 3d. *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on* (pp. 269–270). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4400173

Andrews, K., Wolte, J., & Pichler, M. (1997). Information Pyramids (TM): A New Approach to Visualizing Large Hierarchies. *Proceedings of the IEEE Visualization'97*, (October), 49–52. Retrieved from http://courses.iicm.tu-graz.ac.at/liberation/iicm_papers/vis97.pdf

Anslow, C., Marshall, S., Noble, J., & Biddle, R. (2006). Evaluating X3D for use in software visualization. *Proceedings of the 2006 ACM symposium on Software visualization - SoftVis '06*, 161. doi:10.1145/1148493.1148524

Bacchelli, A., Rigotti, F., Hattori, L., & Lanza, M. (2011). *Manhattan — 3D City Visualizations in Eclipse*. Milano.

Bacim, F., Polys, N., Chen, J., Setareh, M., Ji, L., & Ma, L. (2010). Cognitive scaffolding in Web3D learning systems: a case study for form and structure. *Proceedings of the 15th International Conference on Web 3D Technology-Web3D '10*, 93–100. Retrieved from http://dl.acm.org/citation.cfm?id=1836063

Balzer, M., & Deussen, O. (2007). Level-of-detail visualization of clustered graph layouts. *2007 6th International Asia-Pacific Symposium on Visualization*, 133–140. doi:10.1109/APVIS.2007.329288

Balzer, M., Noack, A., Deussen, O., & Lewerentz, C. (2004). Software landscapes: Visualizing the structure of large software systems. In Eurographics Association (Ed.), *VisSym 2004, Symposium on Visualization*. Konstanz, Germany: Bibliothek der Universität Konstanz. Retrieved from http://graphics.uni-konstanz.de/publikationen/2004/software_landscapes/Balzer et al. -- Software Landscapes - Visualizing the Structure of Large Software Systems.pdf

Bassil, S., & Keller, R. K. (2001). Software visualization tools: Survey and analysis. *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on* (pp. 7–17). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=921708

Beck, F., & Diehl, S. (2010). Visual comparison of software architectures. *Proceedings of the 5th international symposium on Software visualization* (pp. 183–192). ACM. Retrieved from http://portal.acm.org/citation.cfm?id=1879238

Biaggi, A. (2008). *Citylyzer: A 3D Visualization Plug-in for Eclipse*. Università della Svizzera italiana.

Boccuzzo, S., & Gall, H. (2007a). Cocoviz: Towards cognitive software visualizations. *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on* (pp. 72–79). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4290703

Boccuzzo, S., & Gall, H. C. (2007b). Cocoviz: Supported cognitive software visualization. *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on* (pp. 273–274). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4400175

Boccuzzo, S., & Gall, H. C. (2008). Software visualization with audio supported cognitive glyphs. *2008 IEEE International Conference on Software Maintenance*, 366–375. doi:10.1109/ICSM.2008.4658085

Boccuzzo, S., & Gall, H. C. (2009a). Automated comprehension tasks in software exploration. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 570–574). IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=1747558

Boccuzzo, S., & Gall, H. C. (2009b). CocoViz with ambient audio software exploration. *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* (pp. 571–574). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5070558

Bonyuet, D., Ma, M., & Jaffrey, K. (2004). 3D visualization for software development. *Proceedings. IEEE International Conference on Web Services, 2004.*, 708–715. doi:10.1109/ICWS.2004.1314802

Booch, G. (2010). Why don't developers draw diagrams? *Proceedings of the 5th international symposium on Software visualization* (pp. 3–4). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1879214

Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., et al. (2010). Code bubbles: a working set-based interface for code understanding and maintenance. *Proceedings of the 28th international conference on Human factors in computing systems* (pp. 2503–2512). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1753706

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of ManMachine Studies*, *18*(6), 543–554. doi:10.1016/S0020-7373(83)80031-5

Burch, M., Diehl, S., & Weißgerber, P. (2005). Visual data mining in software archives. *Proceedings of the 2005 ACM symposium on Software visualization - SoftVis '05*, 37. doi:10.1145/1056018.1056024

Carneiro, G., Orrico, A., & Mendonça, M. (2007). Empirically Evaluating the Usefulness of Software Visualization Techniques in Program Comprehension Activities. *Universidade Salvador, Brasil*. Retrieved from http://www.nuperc.unifacs.br/grupos-de-pesquisa/gesa/projetos/source-miner-plugin-visualization/experimental-results-and-publications/Final Paper JIISIC 2007.pdf

Caserta, P., & Zendra, O. (2010). Visualization of the Static aspects of Software: a survey. *IEEE transactions on visualization and computer graphics*, *17*(7), 1–20. doi:10.1109/TVCG.2010.110

Charters, S. M., Knight, C., Thomas, N., & Munro, M. (2002). Visualisation for informed decision making; from code to components. *Proceedings of the 14th international conference on Software engineering and knowledge engineering* (pp. 765–772). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=568891

Churcher, N., & Irwin, W. (2005). Informing the design of pipeline-based software visualisations. *proceedings of the 2005 Asia-Pacific symposium on Information visualisation-Volume 45* (pp. 59–68). Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1082325

Churcher, N., Keown, L., & Irwin, W. (1999). Virtual worlds for software visualisation. Retrieved from http://www.cosc.canterbury.ac.nz/research/RG/svg/softvis99/softvis99-churcher-keown-irwin.pdf

Cockburn, A. (2004). Evaluating Spatial Memory in Two and Three Dimensions. *International Journal of Human-Computer*, 1–17. Retrieved from http://www.sciencedirect.com/science/article/pii/S1071581904000096

Cockburn, Andy. (2004). Revisiting 2D vs 3D Implications on Spatial Memory. *Proceeding AUIC '04 Proceedings of the fifth conference on Australasian user interface - Volume 28* (Vol. 28, pp. 25–31). Retrieved from http://dl.acm.org/citation.cfm?id=976314

Cornelissen, B., Zaidman, A., & Van Deursen, A. (2011). A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, *37*(3), 341–355. doi:10.1109/TSE.2010.47

De F. Carneiro, G., Magnavita, R., & Mendonça, M. (2008). Combining software visualization paradigms to support software comprehension activities. *Proceedings of the 4th ACM symposium on Software visuallization - SoftVis '08*, 201. doi:10.1145/1409720.1409755

Diehl, S. (2007a). *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software* (p. 187). Springer.

Diehl, S. (2007b). *Software visualization in the large*. *Computer* (Vol. 29, pp. 1–191). Springer-Verlag Berlin Heidelberg 2007. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=488299

Eichberg, M., Haupt, M., Mezini, M., & Schäfer, T. (2005). Comprehensive software understanding with SEXTANT. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on* (pp. 315–324). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1510127

Ellershaw, S., & Oudshoorn, M. (1994). *Program Visualization - The State of the Art*. *Program* (pp. 1–34). Citeseer. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.7410

Fasano, F., & Oliveto, R. (2009). Supporting Project Management with Fine-Grained Artefact Management in ADAMS. *International Journal of Computers and Applications - 2009*, *31*(3), 145–152. doi:10.2316/Journal.202.2009.3.202-2961

Feijs, L., & De Jong, R. (1998). 3D visualization of software architectures. *Communications of the ACM*, *41*(12), 73–78. doi:10.1145/290133.290151

Fronk, A., Bruckhoff, A., & Kern, M. (2006). 3D Visualisation of Code Structures in Java Software Systems. *Proceedings of the 2006 ACM symposium on Software visualization* (pp. 145–146). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1148515

Gallagher, K., Hatch, A., & Munro, M. (2008). Software Architecture Visualization: An Evaluation Framework and Its Application. *IEEE Transactions on Software Engineering*, *34*(2), 260–270. doi:10.1109/TSE.2007.70757

Ghanam, Y., & Carpendale, S. (2008). *A Survey Paper on Software Architecture Visualization*. *Software Engineering, IEEE Transactions on*. Retrieved from http://dspace.ucalgary.ca/bitstream/1880/46648/1/2008-906-19.pdf

Gračanin, D., Matković, K., & Eltoweissy, M. (2005). Software visualization. *Innovations in Systems and Software Engineering*, *1*(2), 221–230. doi:10.1007/s11334-005-0019-8

Graham, H., Yang, H., & Berrigan, R. (2004). A solar system metaphor for 3D visualisation of object oriented software metrics. *Proc. Australasian Symp. Information Visualization*, *35*, 53–59. Retrieved from http://dl.acm.org/citation.cfm?id=1082108

Greevy, O., Lanza, M., & Wysseier, C. (2006). Visualizing live software systems in 3D. *Proceedings of the 2006 ACM symposium on Software visualization - SoftVis '06*, 47. doi:10.1145/1148493.1148501

Gregor, S. (2006). The nature of theory in information systems. *MIS Quarterly*, *30*(3), 611–642. doi:10.1080/0268396022000017725

Hattori, L., & Lanza, M. (2010). Syde: A tool for collaborative software development. *Software Engineering, 2010 ACM/IEEE 32nd International Conference on* (Vol. 2, pp. 235–238). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6062168

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105. doi:10.2307/249422

Kienle, H. M., & Muller, H. a. (2007). Requirements of Software Visualization Tools: A Literature Survey. *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2–9. doi:10.1109/VISSOF.2007.4290693

Knight, C. (1999). Comprehension with virtual environment visualisations. *Program Comprehension, 1999*. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=777733

Knight, C., & Munro, M. (1999). Visualising software-a key research area. *Proceedings of the IEEE International Conference on Software Maintenance* (p. 437). Citeseer. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.3385&rep=rep1&type=pdf

Knight, C., & Munro, M. (2002). Program comprehension experiences with GXL; comprehension for comprehension. *Proceedings 10th International Workshop on Program Comprehension*, 147–156. doi:10.1109/WPC.2002.1021336

Knight, Claire, & Munro, M. (2000a). Mindless visualisations. *The 6th ERCIM "User Interfaces for All" Workshop*, (October). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.2183&rep=rep1&type=pdf

Knight, Claire, & Munro, M. (2000b). Virtual but visible software. *Information Visualization, 2000. Proceedings. IEEE International Conference on* (pp. 198–205). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=859756

Knight, Claire, & Munro, M. (2000c). Should users inhabit visualisations? *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.(WET ICE 2000). Proeedings. IEEE 9th International Workshops on* (pp. 43–50). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=883703

Knight, Claire, & Munro, M. (2001). Software visualisation conundrums. *History*, (July). Retrieved from http://www.dur.ac.uk/CompSci/research/technical-reports/2001/tech_report-05-01.pdf

Kot, B., Grundy, J., & Hosking, J. (2005). Information Visualisation Utilising 3D Computer Game Engines Case Study : A source code comprehension tool. *Source*, 53–60.

Krebs, R. (2012). *Vera: An extensible Eclipse Plug-In for Java Enterprise Application Analysis*. University of Bern. Retrieved from http://scg.unibe.ch/archive/masters/Kreb12a.pdf

Kuhn, A., Erni, D., & Nierstrasz, O. (2010). Embedding spatial software visualization in the IDE: an exploratory study. *Proceedings of the 5th international symposium on Software visualization* (pp. 113–122). ACM. Retrieved from http://portal.acm.org/citation.cfm?id=1879229

Lanza, M., Gall, H., & Dugerdil, P. (2009). EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution. *2009 13th European Conference on Software Maintenance and Reengineering*, 293–296. doi:10.1109/CSMR.2009.14

Lemieux, F., & Salois, M. (2006). Visualization techniques for program comprehension: A literature review. *The 2006 conference on new trends in software methodologies, tools and techniques: proceedings of the fifth SoMeT_06* (pp. 22–47). Amsterdam, The Netherlands: IOS Press.

Maletic, J. I., Marcus, A., & Collard, M. L. (2002). A task oriented view of software visualization. *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis* (pp. 32–40). IEEE Comput. Soc. doi:10.1109/VISSOF.2002.1019792

Malnati, J. (2007). *X-Ray: An Eclipse Plug-in for Software Visualization*. University of Lugano. Retrieved from http://www.inf.usi.ch/faculty/lanza/Downloads/Maln07a.pdf

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision Support Systems*, *15*(4), 251–266. doi:10.1016/0167-9236(94)00041-2

Marcus, A., Feng, L., & Maletic, J. I. (2003). 3D Representations for Software Visualization. *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*, 27. doi:10.1145/774834.774837

Markus, M., Majchrzak, A., & Gasser, L. (2002). A design theory for systems that support emergent knowledge processes. *Mis Quarterly*, *26*(3), 179–212. Retrieved from http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract

Nunamaker, J. F., Chen, M., & Purdin, T. D. M. (1991). Systems development in information systems research. *Journal of Management Information Systems*, *7*(3), 89–106. doi:10.1109/HICSS.1990.205401

Ogawa, M., & Ma, K.-L. (2008). StarGate: A unified, interactive visualization of software projects. *Visualization Symposium, 2008. PacificVIS'08. IEEE ...*, (VIDi), 191–198. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4475476

Ogawa, M., & Ma, K.-L. (2009). Code_Swarm: a Design Study in Organic Software Visualization. *IEEE transactions on visualization and computer graphics*, *15*(6), 1097–104. doi:10.1109/TVCG.2009.123

Pacione, M. (2004). Software visualization for object-oriented program comprehension. *Software Engineering, 2004. ICSE 2004.*, 5–7. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1317423

Panas, T., Berrigan, R., & Grundy, J. (2003). A 3D metaphor for software production visualization. *Proceedings on Seventh International Conference on Information Visualization, 2003. IV 2003.* (pp. 314–319). IEEE Comput. Soc. doi:10.1109/IV.2003.1217996

Panas, Thomas, Epperly, T., Quinlan, D., Sæbjørnsen, A., & Vuduc, R. (2007). Architectural Visualization of C / C ++ Source Code for Program Comprehension. *29th International Conference on Software Engineering Minneapolis, MN, United States*.

Panas, Thomas, Epperly, T., Quinlan, D., Saebjornsen, A., & Vuduc, R. W. (2007). Communicating Software Architecture using a Single-View Visualization. *Babel*, (Iceccs), 217–228. doi:10.1109/ICECCS.2007.20

Panas, Thomas, Lincke, R., & Löwe, W. (2005). Online-configuration of software visualizations with Vizz3D. *Proceedings of the 2005 ACM symposium on Software visualization - SoftVis '05*, *1*(212), 173. doi:10.1145/1056018.1056043

Parnin, C., & Görg, C. (2007). Design Guidelines for Ambient Software Visualization in the Workplace. *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on* (pp. 18–25). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4290695

Petre, M, Blackwell, A., & Green, T. (1998). Cognitive questions in software visualization. *Software visualization: Programming as a multimedia experience*. MIT Press. Retrieved from http://www.cl.cam.ac.uk/~afb21/publications/book-chapter.html

Petre, Marian. (2002). Mental imagery, visualisation tools and team work. *Proceedings of the Second Program Visualization Workshop* (pp. 2–13). HornstrupCentret, Denmark: University of Aarhus. Retrieved from http://www.daimi.au.dk/PB/567/PB-567.pdf

Petre, Marian, & de Quincey, E. (2006). A gentle overview of software visualisation. *Autumn 2006 PPIG Newsletter*, (September), 1–10. Retrieved from http://www.ppig.org/newsletters/2006-09/1-overview-swviz.pdf

Price, B. A., Baecker, R. M., & Small, I. S. (1994). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, *4*(3), 211–266.

Rigotti, F. (2011). *Visualizing Software Systems and Team Activity*. Università della Svizzera Italiana. Retrieved from http://www.inf.usi.ch/faculty/lanza/Downloads/Rigo2011a.pdf

Rilling, J., & Mudur, S. P. (2005). 3D visualization techniques to support slicing-based program comprehension. *Computers & Graphics*, *29*(3), 311–329. doi:10.1016/j.cag.2005.03.007

Sensalire, M., & Ogao, P. (2007a). Visualizing object oriented software: towards a point of reference for developing tools for industry. *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, 26–29. doi:10.1109/VISSOF.2007.4290696

Sensalire, M., & Ogao, P. (2007b). Tool users requirements classification: how software visualization tools measure up. *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa - AFRIGRAPH '07*, *1*(212), 119–124. Retrieved from http://dl.acm.org/citation.cfm?id=1294705

Sensalire, M., Ogao, P., & Telea, A. (2008). Classifying desirable features of software visualization tools for corrective maintenance. *Proceedings of the 4th ACM symposium on Software visuallization - SoftVis '08*, 87. doi:10.1145/1409720.1409734

Sensalire, M., Ogao, P., & Telea, A. (2009). Evaluation of software visualization tools: Lessons learned. *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 19–26. doi:10.1109/VISSOF.2009.5336431

Sharafi, Z. (2011). A Systematic Analysis of Software Architecture Visualization Techniques. *2011 IEEE 19th International Conference on Program Comprehension*, 254–257. doi:10.1109/ICPC.2011.40

Steinbrückner, F., & Lewerentz, C. (2010). Representing development history in software cities. *Proceedings of the 5th international symposium on Software visualization* (pp. 193–202). ACM. Retrieved from http://portal.acm.org/citation.cfm?id=1879239

Steinbrückner, Frank. (2010). Coherent Software Cities. *2010 IEEE International Conference on Software Maintenance (ICSM)* (pp. 1–2). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5610421

Storey, M. A. D., Fracchia, F. D., & Müller, H. A. (1999). Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, *44*(3), 171–185. doi:10.1016/S0164-1212(98)10055-9

Storey, M.-A. D., Čubranić, D., & German, D. M. (2005). On the use of visualization to support awareness of human activities in software development. *Proceedings of the 2005 ACM symposium on Software visualization - SoftVis '05* (Vol. 1, p. 193). New York, New York, USA: ACM Press. doi:10.1145/1056018.1056045

Storey, M.-A.-A. D., Wong, K., Fracchia, F. D., & Müller, H. A. (1997). On Integrating Visualization Techniques for Effective Software Exploration. *1997. Proceedings., IEEE Symposium on Information Visualization* (pp. 38–45). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=636784

Sulaiman, S., Idris, N. B., & Sahibuddin, S. (2005). Enhancing Cognitive Aspects of Software Visualization Using DocLike Modularized Graph. *The International Arab Journal of Information Technology*, *2*(1), 1–9. Retrieved from http://www.uop.edu.jo/download/Research/members/1-Shahida.pdf

Teyseyre, A. R., & Campo, M. R. (2009). An overview of 3D software visualization. *IEEE transactions on visualization and computer graphics*, *15*(1), 87–105. doi:10.1109/TVCG.2008.86

Theron, R., Gonzalez, A., & Garcia, F. J. (2008). Supporting the understanding of the evolution of software items. *Proceedings of the 4th ACM symposium on Software visuallization - SoftVis '08*, 189. doi:10.1145/1409720.1409750

Tichelaar, S., Ducasse, S., Demeyer, S., & Nierstrasz, O. (2000). A meta-model for language-independent refactoring. *2000. Proceedings. International Symposium on Principles of Software Evolution* (pp. 154–164). IEEE Comput. Soc. doi:10.1109/ISPSE.2000.913233

Tudoreanu, M. E. (2003). Designing effective program visualization tools for reducing user's cognitive effort. *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*, 105. doi:10.1145/774845.774848

Von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, *28*(8), 44–55. doi:10.1109/2.402076

Wettel, R. (2010). *Software Systems as Cities*. Faculty of Informatics of the Università della Svizzera Italiana. Retrieved from http://www.inf.usi.ch/phd/wettel/download.php?f=Wettel10b-PhDThesis.pdf

Wettel, R., & Lanza, M. (2007a). Program comprehension through software habitability. *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on* (pp. 231–240). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4268257

Wettel, R., & Lanza, M. (2007b). Visualizing Software Systems as Cities. *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 92–99. doi:10.1109/VISSOF.2007.4290706

Wettel, R., & Lanza, M. (2008). Codecity: 3d visualization of large-scale software. *Companion of the 30th international conference on Software engineering* (pp. 921–922). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=1370188

Wettel, R., & Lanza, M. (2011). Software systems as cities: A controlled experiment. *international conference on Software*. University of Lugano. Retrieved from http://dl.acm.org/citation.cfm?id=1985868

Wettel, R., Lanza, M., & Robbes, R. (2010). *Empirical validation of CodeCity: A controlled experiment*. Retrieved from http://doc.rero.ch/lm.php?url=1000,42,6,20110309110626-OX/ITR1005.pdf

Wieringa, R. (2010). Design science methodology. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10* (Vol. 2, p. 493). New York, New York, USA: ACM Press. doi:10.1145/1810295.1810446

Wieringa, R., Daneva, M., & Condori-Fernandez, N. (2011). The Structure of Design Theories, and an Analysis of their Use in Software Engineering Experiments. *2011 International Symposium on Empirical Software Engineering and Measurement*, 295–304. doi:10.1109/ESEM.2011.38

Xie, X., Poshyvanyk, D., & Marcus, A. (2006). 3D Visualization for Concept Location in Source Code. *Proceeding ICSE '06 Proceedings of the 28th international conference on Software engineering* (pp. 839–842).

Xu, S., Chen, X., & Liu, D. (2009). Classifying software visualization tools using the Bloom's taxonomy of cognitive domain. *Electrical and Computer Engineering, 2009. CCECE'09. Canadian Conference on* (pp. 13–18). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5090082

Yang, H., & Graham, H. (2003). Software Metrics and Visualisation. *Univ. of Auckland, Tech. Rep*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.135.7600&rep=rep1&type=pdf

Young, P., & Munro, M. (1998). Visualising software in virtual reality. *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on* (pp. 19–26). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=693276

Appendices

Appendix A: XML Schemas

1. Scrum XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="nz.ac.aut.scrumcity"
xmlns:sc="nz.ac.aut.scrumcity"
elementFormDefault="qualified">
    <element name="Release">

        <complexType>
            <sequence>
                <element name="ID" type="string"></element>
                <element name="Title" type="string"></element>
                <element name="Description" type="string"></element>
                <element name="StartDate" type="dateTime"></element>
                <element name="TotalWorkTime" type="double">
                    <annotation>
                        <documentation>
                            Any value supplied here will be ignored
                            during runtime/visualization as this value
                            will be calculated dynamically based on
                            values of constituting features.
                        </documentation>
                    </annotation>
                </element>
                <element name="ActualCompletionDate" type="dateTime">
                </element>
                <element name="ExpectedCompletionDate" type="dateTime">
                </element>
                <element name="Sprints" maxOccurs="1" minOccurs="1">
                    <complexType>
                        <sequence minOccurs="1" maxOccurs="unbounded">
                            <element name="Sprint" type="sc:SprintType">
                            </element>
                        </sequence>
                    </complexType>
                </element>
            </sequence>
        </complexType>
    </element>


    <complexType name="SprintType">
        <sequence>
            <element name="ID" type="string"></element>
            <element name="Title" type="string"></element>
            <element name="Description" type="string"></element>
```

```
47              <element name="StartDate" type="dateTime"></element>
48
49              <element name="ActualCompletionDate"
50                       type="dateTime">
51              </element>
52              <element name="ExpectedCompletionDate"
53                       type="dateTime">
54              </element>
55              <element name="Features" maxOccurs="1"
56                       minOccurs="1">
57                  <complexType>
58                      <sequence minOccurs="1" maxOccurs="unbounded">
59                          <element name="Feature" type="sc:FeatureType">
60                          </element>
61                      </sequence>
62                  </complexType>
63              </element>
64
65          </sequence>
66      </complexType>
67
68      <complexType name="FeatureType">
69          <sequence>
70              <element name="ID" type="string"></element>
71              <element name="Title" type="string"></element>
72              <element name="Description" type="string"></element>
73              <element name="Owner" type="string"></element>
74              <element name="Category" type="string"></element>
75              <element name="Type" type="sc:Feature_Type"
76                       minOccurs="1" maxOccurs="1">
77                  <annotation>
78                      <documentation>
79                          Type of Feature. Acceptable values are
80                          : Feature, BugFix, Enhancement.
81                      </documentation>
82                  </annotation>
83              </element>
84              <element name="Priority" type="sc:Priority_Type"
85                       minOccurs="1" maxOccurs="1">
86                 <annotation>
87                      <documentation>
88                          Priority of this Feature. Acceptable
89                          values are: High, Normal, Low.
90                      </documentation>
91                  </annotation>
92              </element>
93              <element name="MethodRefs" minOccurs="1" maxOccurs="1">
94                  <complexType>
95                      <sequence>
96                          <element name="MethodRef" type="string"
97                                   maxOccurs="unbounded" minOccurs="1">
98                              <annotation>
99                                  <documentation>
100                                 This should hold the Qualified Name of
101                                 any method that is a direct manifestation
```

```
                                any method that is a direct manifestation
                                of this feature, i.e., was created as a
                                result of this feature's implementation.
                                </documentation>
                            </annotation></element>
                    </sequence>
                </complexType>
            </element>
            <element name="ClassRefs"  minOccurs="1" maxOccurs="1">
            <complexType>
                    <sequence>
                        <element name="ClassRef" type="string"
                                 maxOccurs="unbounded" minOccurs="1">
                            <annotation>
                                <documentation>
                                This should hold the Qualified Name
                                of any Class that is a direct
                                manifestation of this feature, i.e.,
                                was created as a result of
                                this feature's implementation.
                                </documentation>
                            </annotation></element>
                    </sequence>
                </complexType>
            </element>
            <element name="Tasks"  minOccurs="0" maxOccurs="1">
            <complexType>
                    <sequence>
                        <element name="Task" type="string"
                                 maxOccurs="unbounded" minOccurs="1">
                            <annotation>
                                <documentation>
                                 A feature can have a set of multiple tasks
                                </documentation>
                            </annotation>
                        </element>
                    </sequence>
                </complexType>
            </element>
            <element name="WorkEntries"  minOccurs="0" maxOccurs="1">
            <complexType>
                    <sequence>
                        <annotation>
                            <documentation>
                            A set of work entries corresponding to the
                             daily updates that developers record for
                              each feature/user story that they have
                              worked on
                            </documentation>
                        </annotation>
                        <element name="WorkEntry" maxOccurs="unbounded"
                                 minOccurs="1">
                            <complexType>
                                <sequence>
                                    <element name="QName" type="string">
```

```
                                    </element>
                                    <element name="Date" type="dateTime">
                                    </element>
                                    <element name="Hours" type="double">
                                    </element>
                                    <element name="Type"
                                             type="sc:WorkEntry_Type">
                                    </element>
                                </sequence>
                            </complexType>
                        </element>
                    </sequence>
                </complexType>
            </element>
            <element name="CommenceDateTime" type="dateTime"></element>
            <element name="CompletionDateTime" type="dateTime"></element>
            <element name="OriginalWorkEstimate" type="double"></element>
        </sequence>
    </complexType>

    <simpleType name="Feature_Type">
        <restriction base="string">
            <enumeration value="Feature"></enumeration>
            <enumeration value="BugFix"></enumeration>
            <enumeration value="Enhancement"></enumeration>
        </restriction>
    </simpleType>

    <simpleType name="Priority_Type">
        <restriction base="string">
            <enumeration value="High"></enumeration>
            <enumeration value="Normal"></enumeration>
            <enumeration value="Low"></enumeration>
        </restriction>
    </simpleType>

    <simpleType name="WorkEntry_Type">
        <restriction base="string">
            <enumeration value="Remaining"></enumeration>
            <enumeration value="Completed"></enumeration>
        </restriction>
    </simpleType>
</schema>
```

2. System Artefact Documentation XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="nz.ac.aut.scrumcity"
xmlns:sc="nz.ac.aut.scrumcity"
elementFormDefault="qualified"
>

    <complexType name="ChangeRequestType">
        <sequence>
            <element name="Title" type="string" maxOccurs="1"
                     minOccurs="1">
            </element>
            <element name="RequestedBy" type="string" maxOccurs="1"
                     minOccurs="1"></element>
            <element name="Description" type="string" maxOccurs="1"
                     minOccurs="1"></element>
            <element name="Date" type="dateTime" maxOccurs="1"
                     minOccurs="1"></element>
        </sequence>
    </complexType>

<element name="SysArtifactDoc">
  <complexType>
   <sequence>
   <element name="ArtifactDoc" maxOccurs="unbounded" minOccurs="1">
        <complexType>
            <sequence>
                <element name="Author" type="string" maxOccurs="1"
                         minOccurs="0">
                    <annotation>
                        <documentation>
                         Name of the system artifact author
                        </documentation>
                    </annotation></element>
                <element name="Title" type="string" maxOccurs="1"
                    minOccurs="1">
                    <annotation>
                        <documentation>
                        Name of the system artifact, i.e.
                        package, class, or method name.
                        </documentation>
                    </annotation>
                </element>
                <element name="ArtifactQName" type="string"
                    maxOccurs="1" minOccurs="1">
                    <annotation>
                        <documentation>
                        A unique fully qualified name of this
                        artifact.
                        Example: 'com.package.class.method'
                        </documentation>
                    </annotation>
                </element>
                <element name="ShortDesc" type="string"
                         maxOccurs="1" minOccurs="1">
```

```
                    <annotation>
                        <documentation>
                         A short (preferably less than 50 words)
                         description of the purpose of this
                         artifact.
                        </documentation>
                    </annotation>
                </element>
                <element name="Documentation" type="string"
                         maxOccurs="1" minOccurs="0">
                    <annotation>
                        <documentation>
                        An elaborated description of this
                        artifact
                        </documentation>
                    </annotation></element>
                <element name="DeveloperNotes" type="string"
                          maxOccurs="1" minOccurs="0">
                    <annotation>
                        <documentation>
                         Notes to guide developers who might
                         need to modify/maintain this artifact
                         </documentation>
                    </annotation></element>
                <element name="UsageNotes" type="string"
                         maxOccurs="1" minOccurs="0">
                    <annotation>
                        <documentation>
                        Concise comment or example on how
                        this artifact can be used
                        </documentation>
                    </annotation></element>
                <element name="Date" type="dateTime">
                    <annotation>
                        <documentation>
                        The date when the system artifact
                        was first created.
                        </documentation>
                    </annotation></element>
                 <element name="ChangeRequests" maxOccurs="1"
                           minOccurs="0">
                    <annotation>
                        <documentation>
                        Any change requests for this artifact.
                        </documentation>
                    </annotation>
                    <complexType>
                        <sequence>
                        <element name ="ChangeRequest"
                        type="sc:ChangeRequestType" minOccurs="1"
                         maxOccurs="unbounded"></element>
                        </sequence>
                    </complexType>
                 </element>
            </sequence>
```

```
102                    <complexType>
103                        <sequence>
104                        <element name ="ChangeRequest"
105                        type="sc:ChangeRequestType" minOccurs="1"
106                         maxOccurs="unbounded"></element>
107                        </sequence>
108                    </complexType>
109                </element>
110            </sequence>
111        </complexType>
112    </element>
113    </sequence>
114    </complexType>
115  </element>
116 </schema>
```

3. Example XML Scrum Data

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Release xmlns="nz.ac.aut.scrumcity"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="nz.ac.aut.scrumcity ../schemas/Scrum.xsd ">
  <ID>10</ID>
  <Title>Version 1.1</Title>
  <Description>Release Description</Description>
  <StartDate>2012-06-20T14:00:00</StartDate>
  <TotalWorkTime>560.0</TotalWorkTime>
  <ActualCompletionDate>2012-10-20T00:00:00</ActualCompletionDate>
  <ExpectedCompletionDate>2012-09-10T00:00:00</ExpectedCompletionDate>
  <Sprints>
    <Sprint>
      <ID>12</ID>
      <Title>Modelling Source Code</Title>
      <Description>Sprint Description</Description>
      <StartDate>2012-06-20T00:00:00</StartDate>
      <ActualCompletionDate>2012-07-02T00:00:00</ActualCompletionDate>
      <ExpectedCompletionDate>2012-06-30T00:00:00</ExpectedCompletionDate>
      <Features>
        <Feature>
          <ID>12.1</ID>
          <Title>Integrate with Vera</Title>
          <Description>As a user, I need to be able to select a project
           from Eclipse Package Explorer Window and ask ScrumCity to
            Visualise it</Description>
          <Owner>Ted</Owner>
          <Category>Category</Category>
          <Type>Feature</Type>
          <Priority>High</Priority>
          <MethodRefs>
            <MethodRef>nz.ac.aut.scrumcity.util.VeraProjectModelRepoHelper#getInstance</MethodRef>
            <MethodRef>nz.ac.aut.scrumcity.vera.visualiser.JmeEmbedder#createControl</MethodRef>
          </MethodRefs>
          <ClassRefs>
            <ClassRef>nz.ac.aut.scrumcity.vera.visualiser.ScrumCityVisualiser</ClassRef>
            <ClassRef>nz.ac.aut.scrumcity.vera.visualiser.JmeEmbedder</ClassRef>
          </ClassRefs>
          <Tasks>
            <Task>Configure ScrumCity Plug-in to Use Vera</Task>
            <Task>Acquire the source code model from Vera</Task>
          </Tasks>
          <WorkEntries>
            <WorkEntry>
              <QName>nz.ac.aut.scrumcity.vera.visualiser.ScrumCityVisualiser</QName>
              <Date>2012-06-20T12:00:00</Date>
              <Hours>10</Hours>
              <Type>Remaining</Type>
            </WorkEntry>
             <WorkEntry>
              <QName>nz.ac.aut.scrumcity.vera.visualiser.ScrumCityVisualiser</QName>
              <Date>2012-06-23T12:00:00</Date>
              <Hours>7</Hours>
              <Type>Remaining</Type>
            </WorkEntry>
              <WorkEntry>
               <QName>nz.ac.aut.scrumcity.vera.visualiser.ScrumCityVisualiser</QName>
               <Date>2012-06-25T12:00:00</Date>
               <Hours>4</Hours>
               <Type>Remaining</Type>
             </WorkEntry>
          </WorkEntries>
          <CommenceDateTime>2012-06-22T12:00:00</CommenceDateTime>
          <CompletionDateTime>2012-06-27T12:00:00</CompletionDateTime>
          <OriginalWorkEstimate>10.0</OriginalWorkEstimate>
        </Feature>
      </Features>
    </Sprint>
  </Sprints>
</Release>
```

4. Example XML System Artefact Documentation

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SysArtifactDoc xmlns="nz.ac.aut.scrumcity"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="nz.ac.aut.scrumcity ../schemas/SystemArtifactDocumentation.xsd ">
    <ArtifactDoc>
        <Author>Mujtaba</Author>
        <Title>ScrumCityVisualiser</Title>
        <ArtifactQName>nz.ac.aut.scrumcity.vera.visualiser.ScrumCityVisualiser</ArtifactQName>
        <ShortDesc>This class is responsible for handling Vera's Object Model</ShortDesc>
        <Documentation>Some Detailed Documentation....</Documentation>
        <DeveloperNotes>This is where ScrumCity's jME3 canvas (JmeCanvasContext) gets created.</DeveloperNotes>
        <UsageNotes>This is ScrumCity's main class. ScrumCityCreator is invoked by this class</UsageNotes>
        <Date>2001-01-01T14:12:00</Date>
        <ChangeRequests>
            <ChangeRequest>
                <Title>Fix bug#1241</Title>
                <RequestedBy>Nick</RequestedBy>
                <Description>some description...</Description>
                <Date>2012-12-01T14:12:00</Date>
            </ChangeRequest>
            <ChangeRequest>
                <Title>Modify the scene constructor</Title>
                <RequestedBy>Sam</RequestedBy>
                <Description>None</Description>
                <Date>2012-12-01T14:12:00</Date>
            </ChangeRequest>
        </ChangeRequests>
    </ArtifactDoc>
</SysArtifactDoc>
```

Appendix B: Keyboard Function Map