ABSTRACTION LAYERED ARCHITECTURE: IMPROVEMENTS IN MAINTAINABILITY OF COMMERCIAL SOFTWARE CODE

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER AND INFORMATION SCIENCES

Supervisor
Associate Professor Roopak Sinha
Mr. John Spray (Datamars Ltd)

January 2020

By
Xingbin Cheng
School of Engineering, Computer and Mathematical Sciences

Abstract

Software maintainability significantly impacts the productivity of developing and maintaining a software code base in Software Development Life Cycle (SDLC). It is said that 90% of commercial software is under maintenance, so any improvements in maintainability can provide high rewards in terms of time and expense. Increased software maintainability can help improve a company's profitability by directly reducing ongoing software development costs.

Abstraction Layered Architecture (ALA) is an innovative reference architecture which aims to improve the maintainability of a software code base in the long run. However, its effectiveness in real projects has remained unexplored. This research explores the extent to which ALA improves the maintainability of commercial software through a joint industry/academic project. In this research, an existing Windows desktop application from Datamars Limited was re-developed by using ALA and compared with the original application based on ISO 25010 maintainability model and ISO 25023 maintainability measures. Specifically, the evaluation of ALA's effectiveness was carried out based on the five sub-characteristics of maintainability: modularity, reusability, analysability, modifiability and testability.

Our experiments show that ALA provides significant improvements in maintainability. During the evaluation, it was found that modularity, reusability, analysability and testability of the re-developed ALA application were overall higher than for the original application. However, the modifiability of the ALA-based application was

not as high as expected. We investigated the reason for the low modifiability of the ALA application, concluded that modifiability measures may improve during long-term maintenance of commercial projects, while the other metrics will remain unaffected.

Contents

List of Tables

List of Figures

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 nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

XINGBIN CHENG

Signature of student

Acknowledgements

This research brings me much challenge in each of the stage, but I still appreciate that I could have a chance to study here, accomplish all the goals in the whole journey, with the help of people around.
Professor. Roopak Sinha, thank you for being my first supervisor. You can always give the most appropriate guidance and advice, but leave the valuable space for me, which inspires and helps me to improve in the right direction.
Mr. John Spray, thank you for being my second supervisor. You are always patient and happy to share any techniques, skills, and discuss the problems with me. You are the most experienced technical specialist that I have ever seen.
My wife Eve, you are always there with me, no matter it is this study, or every moment in our life. I am so lucky to have you, your love, friendship and encouragement. Thank you for everything you've done for me.
My parents, thank you for nurturing me and support me selflessly for all these years.
Thanks for all the fellows in EMSOFT, you are always nice and helpful.
Thank you all for those who had helped me in this study. This accomplishment is accumulated by tiny steps, but cannot achieve without your aid. It is difficult, challenging, frustrated, but more rewarding, exciting and satisfactory.
Looking forward to the next journey of my life.
Xingbin Cheng, January, 2020

Chapter 1

Introduction

This chapter gives an overall view of this thesis. Section 1.1 introduces the background and context of this research. Section 1.3 lists the research questions, which demonstrate the domain, problems and objectives of this research. Section 1.4 briefly explains our approach of carrying out this research and the contributions. Section 1.5 gives the significance of this research. Section 1.6 lays out the rest of the whole thesis.

1.1 Background and Context

Software maintainability impacts the effectiveness and efficiency of developing and maintaining a code base, and it is directly correlated with the productivity of enterprises in commercial environments. It is said that 90% of commercial software is under maintenance, so any improvements here can provide high rewards. Maintainable software is easier to update and extend, which helps a company's profitability by reducing ongoing software development costs. Abstraction Layered Architecture (ALA) (Spray & Sinha, 2018) is an innovative architecture that aims to improve the maintainability of a software code base. This research explores how and to what extent does ALA impact the maintainability of commercial software through a joint industry/academic project.

Datamars (New Zealand) is a company which manufactures various hardware and software solutions for livestock management. Among the software solutions, many code bases have existed for more than 20 years. Datalink is one of the software products, it runs on Windows desktop, and connects to the multiple embedded devices of Datamars. It manages all the data, settings, and updates on the devices. In this research, we re-developed Datalink with ALA and C# language, then measured the maintainability of it by assessing the code base and tracing the performance of real maintenance tasks in Datamars. Comparisons between the re-developed Datalink and the legacy one were carried out to explore the maintainability improvement ALA brought under such circumstance.

Callaghan Innovation is a New Zealand government's innovation agency who provides services for innovative research and development projects. This research was funded by Callaghan Innovation due to the following potential values:

1. Spray and Sinha (2018) proposed ALA and they explained the conformity between ALA and ISO 25023 (2016) measures and proved the maintainability through a student-driven embedded project. However, there was a lack of evidence that ALA works on other kinds of applications, as well as how and to what extent ALA improves maintainability. This research provides evidence on how ALA's mechanisms conform to ISO maintainability measures, and the maintainability improvement ALA brings to a real desktop application.

2. ALA is an innovative architecture but lacks experience with applying the method to real applications. This research explores the approach of utilizing ALA in real projects, which opens a broader way for any potential future projects.

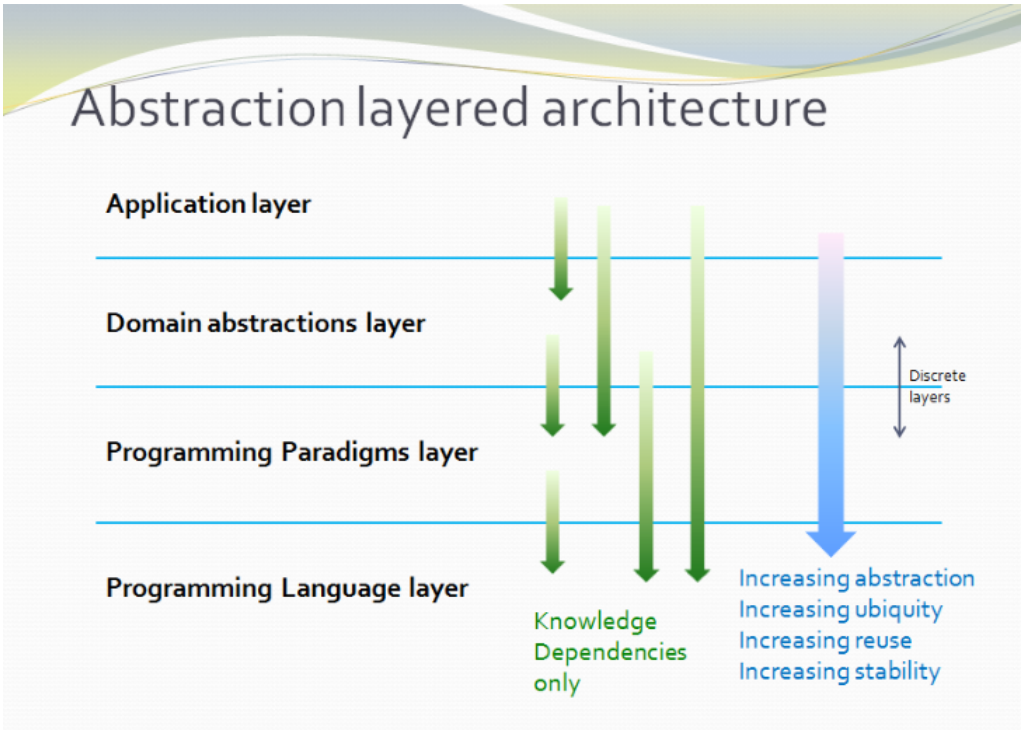


Figure 1.1: ALA Layers Structure

1.2 Abstraction Layered Architecture (ALA)

Abstraction layered architecture (ALA) is a reference software architecture, and it is independent of any specific domain, so it is a general reference architecture. Spray and Sinha (2018) elaborately explained the process that ALA was proposed in their article, and the main intention of this architecture was to improve maintainability of software code bases. ALA consists of four main layer from top to down i.e. application layer, domain abstractions layer, programming paradigms layer and language layer, as illustrated in Figure 1.1.

1. Application layer contains knowledge of a specific application, no more and no less. The application instantiates and wires together the objects of domain abstractions defined in the second layer.

2. Domain Abstractions layer contains all knowledge specific to the domain. They

are usually general functionalities that are non-specific to the application and are highly reusable. The Domain Abstractions should conform to the types defined in the third layer.

3. Programming paradigms layer contains all knowledge specific to the types of computing problem(s) a company's products solve, such as available programming paradigms and associated frameworks. This layer abstracts out how the Domain Abstraction layer and Application layer execute.

4. Language layer contributes the most generic knowledge, that of the programming language and associated libraries.

The dependency relations are also explained in the figure. ALA cares more about Knowledge Dependencies (Cataldo, Mockus, Roberts & Herbsleb, 2009), which are more comprehensive from the perspective of a maintainer in developing phase than Run-time Dependencies (Nicolau, 1989) that happening at the running phase of an application. This is also the reason that ALA aims to improve maintainability by organizing code.

Moreover, the abstraction, ubiquity, reusability and stability of the layers will increase as we going down the layers. With long-term maintenance being carried out, the units of the lower layers will become more and more abstract, ubiquitous, reusable and stable. At some point of the maintenance, the main changes of the tasks will become those at the application layer. Such feature is the strength of ALA, because the maintainability of a normal code base will decrease as time goes by, while ALA's maintainability will adversely increase.

1.3 Research Questions

As aforementioned, this research was funded by Callaghan Innovation due to its potential value. The research questions were formulated in the very early stage before the funding, which structured the research process and outlined the objective of each phase:

RQ1. What are the most relevant measures for assessing maintainability?

RQ2. Which mechanisms in ALA support these measures?

RQ3. How can the process of re-architecting an existing C# application using ALA be generalised for use in future projects?

RQ4. How do the existing implementations of the C# application and the new re-implementation as a result of answering RQ3 compare when assessed for maintainability using the measures identified in RQ1?

RQ5. How well does the assessment carried out in RQ4 relate to the expected enhancements in maintainability from ALA (as identified in RQ2)?

1.4 Solution and Contribution

Our solution was mainly based on the five formulated research questions. We carried out a joint industry & academic project to re-architect and re-develop an existing C# desktop application in Datamars with ALA, which is called Datalink. Then the re-developed application was used as a case study to answer the research questions. The contributions of this thesis mainly came from the following four aspects.

A Group of High-Quality C# Domain Abstractions and Programming Paradigms

The re-developed Datalink involves elaborately designed and implemented domain abstractions and programming paradigms. Those abstractions can be reused in a wide range of C# desktop applications, which considerably saves time and provides

reusability for any possible future projects.

We made part of the source code accessible on a GitHub repository (`https://github.com/cdxybf/ALA_Datalink`). This code repository demonstrates how ALA works in a real project from the implementation perspective.

A Strategy of Evaluating the Maintainability of Code Bases

Maintainability evaluation of a code base is far more than observing the effort of maintenance. The strategy we used in this research is based on the ISO 25010(2011)/25023(2016) quality model, which involves not only maintenance assessments on *Modifiability*, but also other aspects that impact maintainability i.e. *Modularity, Reusability, Analysability and Testability*. However, as ISO does not provide measures for code related characteristics e.g. "Coupling" and "Cohesion", we refined the measures to make them applicable on a pure code base with some commonly used metrics e.g. CK Metric Suite (Chidamber & Kemerer, 1994), Cyclomatic Complexity (McCabe, 1976).

A General Method to Develop ALA Applications

Based on the re-development of Datalink, we proposed a general method for ALA application development. This method is a reliable reference that was built on this real commercial project. It integrates development process management methods to guide and structure the activities in each phase, as well as approaches and principles to comprehend and design high-quality ALA applications.

Maintainability Improvement Evaluation of ALA in Commercial Software Code Base

This research provides empirical evidence on the evaluation of maintainability improvement of ALA. According to Spray and Sinha (2018), ALA intends to improve the

maintainability of a software code base in the long run. We evaluated the maintainability improvement by comparing the re-developed Datalink with the legacy one. Our conclusion is that the *Modularity, Reusability, Analysability and Testability* of ALA are higher than the legacy code base. Moreover, although the current *Modifiability* of ALA is lower, with more and more abstractions being implemented, it would increase gradually. Overall, if we consider long-term maintenance in an application, the maintainability of ALA is considerably higher.

1.5 Significance

Software maintainability significantly impacts software systems in the past, at present, and in the future (Coleman, Lowther & Oman, 1995). Chronologically, the importance of software maintainability has been growing since the early 1960s when only a small part of maintenance work was undertaken in *software development life cycle* (SDLC) (Chu et al., 2002). One decade later during the 1970s, the maintenance work on the legacy systems has become the primary activity in SDLC (Bennett, 1993). According to Grady and Booch (1998), the cost of maintenance is much higher in the 1990s compared with it in the initial development and 65% to 75% time is occupied by maintenance (Muthanna, Kontogiannis, Ponnambalam & Stacey, 2000). Up until now, 90% of commercial software is under maintenance, the cost of which keeps growing as the complexity of modern software has increased exponentially. Hence, putting more efforts for making the software maintainable during the SDLC can significantly reduce the total software cost (Kumar, 2012).

The improvements on software maintainability can provide high rewards. Spray and Sinha (2018) proved the effectiveness and efficiency of ALA in their experiments on an embedded software code base, which requires 12 man-years work to complete the conventional code which compares with one student-year work with ALA. In this

research, the utilization of ALA on the desktop application provided supportive evidence for other types of software rather than embedded one. The general development method of ALA helped to set the direction of future projects, which opens up a boarder approach for ALA. Furthermore, evidence shows how much ALA improves the maintainability of a commercial project in the long run, and the considerable reduction on time and expense it brings would have long term benefits on commercial companies.

1.6 Thesis Structure

The rest of this thesis is organized as follows. In chapter 2, the literature review is carried out to identify the most relevant maintainability measures and assess ALA's mechanisms with those measures, which aims to address the first two research questions. Chapter 3 discusses the research methodologies that we adopted to develop solutions for answering the last three research questions. Chapter 4 records the re-architecting and re-development process of ALA Datalink. In addition, a general method for developing ALA applications is proposed. Chapter 5 compares the maintainability of the re-developed Datalink and the legacy one by applying the measures identified in Chapter 2. Chapter 6 gives a summary to this research, as well as listing the contributions and possible future research directions.

Chapter 2

A Systematic Literature Review of Measures for Code Maintainability

A key first step in this research is to establish objective measures for maintainability, and such measures serve two purposes. Firstly, they can be used to compare ALA with existing solutions towards maintainability. Secondly, and more specifically to this project, these measures can be monitored during the architecting/development process to ensure that an ALA code base remains measurably maintainable. We conducted a systematic literature review (Kitchenham et al., 2009) to collect these measures, which helped to answer two important research questions:

RQ1. What are the most relevant measures for assessing maintainability?

RQ2. Which mechanisms in ALA support these measures?

The rest of this chapter is organized as follows. Section 2.1 describes the process we carried out to conduct the literature review, each step is explained in detail to make them reproducible. Section 2.2 presents the findings about maintainability measures. In Section 2.3, the most relevant maintainability measures were identified and refined to satisfy the aforementioned two research questions. Section 2.4 gives the conclusion and limitation of this literature review.

2.1 The Systematic Literature Review Process

Bettany and Saltikov (2012) explain that the systematic literature review employs a scientific methodology to research, appraise and summarize all relevant studies by starting with a particular review question. Besides that, the process should also be able to be replicated (Johnson, De Li, Larson & McCullough, 2000) and some techniques should be applied to minimize any bias (Petticrew & Roberts, 2008). The significance of a SLR is crucial; Xiao and Watson (2019) state that through reviewing the relevant literature, the depth and breadth of the existing work could be specified. The summarization and synthesis of current work reveal the contradictions and inconsistencies (Paré, Trudel, Jaana & Kitsiou, 2015), as well as identifying the gaps to conduct an exploration. Furthermore, it seeks to to elicit new research activities and suggest areas for investigation in future work (Keele et al., 2007).

A systematic review involves several discrete activities. In some specific domains such as medicine where guidelines have different view of the process steps needed in a systematic review and they usually includes detailed levels of a review model (Kitchenham, 2004). However, the general stages associated with conducting a systematic literature review usually consists of six main stages (Kitchenham et al., 2009). We discussed the stages respectively in the following subsections i.e. *Research Questions* in Section 2.1.1, *Search Process* in Section 2.1.2, *Inclusion and Exclusion Criteria* in Section 2.1.3, *Quality Assessment* in Section 2.1.4, *Data Extraction* in Section 2.1.5, and *Data Synthesis* in Section 2.1.6.

2.1.1 Research Questions

The research questions define the objectives of the whole systematic review process, present detailed and suitable findings of a given domain, and help identifying and scoping future research activities. Thus, the specification of which in any kind of review

is of crucial importance.

Spray and Sinha (2018) presented Abstraction Layered Architecture (ALA) and they proved its maintainability through a student-driven project. However, this evidence is insufficient and requires a deeper investigation, such as the re-architecting of a commercial code base carried out in this thesis. To assess the maintainability and feasibility of ALA in commercial software, we need to identify and collect proper maintainability measures, which helps to quantitatively assess the maintainability of ALA. Therefore, the two related research questions in the SLR are:

RQ1. What are the most relevant measures for assessing maintainability?

RQ2. Which mechanisms in ALA support these measures?

RQ1 aims to collect the most relevant maintainability measures. RQ2 allows us to carry out a preliminary assessment by comparing ALA's design notion with the collected measures, so that we can evaluate ALA's maintainability from its design in advance, which would finally enhance the evaluation of the maintainability of ALA.

2.1.2 Search Process

The key factors in the search process is identifying the databases and the terms for search activities. The library search engine of Auckland University of Technology was used as the main source because it integrates the epidemic databases such as Springer, Scopus, Science Direct, IEEEXplore, ACM Digital Library etc. Google scholar was used as a secondary source to assure the comprehensiveness of relevant sources. Kitchenham (2004) outlines the general steps of developing search terms, shown as follows.

Derive major search terms from the research questions by identifying Intervention, Outcome, and Context. First, since this research aims to investigate software maintainability, the fundamental scope is limited in software and its correlated systems.

Second, maintainability is decomposed in ISO 25010 (2011) quality model as modularity, reusability, analysability, modifiability and reusability, thus the terms involves maintainability facets as well.

Identify alternative spellings and synonyms for the search terms with the help of a thesaurus. Maintainability indicates the effort that was paid for maintenance, so maintenance was chosen as an alternative of maintainability as some studies might indirectly contributes to such topic. Apart from that, maintainability metrics might be explained as "assessing and measuring maintainability", so the terms "assess, assessment, measure and measurement" were included as alternatives. As for spelling, analysability is occasionally used as analyzability, and vice versa.

Use Boolean OR and AND to construct search strings. OR is used to connect terms which as similar meaning, such as measure and measurement described before, while AND concatenates terms to restrict the research. In this case, the term "software maintainability metric" is the most top level, the rest of the terms are alternatively concatenated to build the whole search string.

The search terms were finalized below and the process was performed manually. The search result of the study showed that over 50,000 items related to maintainability were presented. As for the sub-characteristics, studies about modularity occupied around 14,500, following was the reusability which resulted in 9,000 results. The number of published works about testability and modifiability were 3,800 and 1,500 respectively, and the least was analysability in which 600 records were found approximately.

(software OR software application OR software system OR application OR system) AND (maintainability OR modularity OR reusability OR modifiability OR analysability OR analyzability OR testability OR maintenance) AND (metric OR assess OR assessment OR measure OR measurement)

2.1.3 Inclusion and Exclusion Criteria

Ryan and Gwen (2010) explain that SLR provides a clear methodology to reduce bias risks as it follows strict and replicable procedures. The purpose of the inclusion and exclusion criteria is to set up the basis to find the most relevant studies in software maintainability and its correlated domains (Malhotra & Chug, 2016). Generally, the peer-reviewed papers which propose the standards, measurement, metrics and assessment of maintainability based on any model, investigation and experiments were supposed to be collected. Specifically, the detailed criteria to filter studies for this research are listed as follows:

- Studies perform activities or quality model which has direct, indirect influence, value, or empirical evaluation of the maintainability or the facets or subcharacteristics of maintainability, metrics, measurement, assessment and prediction were included.
- References to valuable studies in the previous step were included.
- Some articles which have only the literature review on maintainability and its facets as their main concentration were also included.
- Duplication of studies such as same study of different versions or from different databases were excluded.
- Some studies were excluded by the incompatible content between the title/abstract, key words and the topic of this research.

With the inclusion and exclusion criteria being applied, the relevant studies were refined to 102 results.

2.1.4 Quality Assessment

Quality assessment is a process to identify studies which directly correlated with the research questions. Keele (2007) describes that quality assessment provides more detailed inclusion and exclusion of the criteria, and an investigation of whether differences results of study are explained by quality differences. Furthermore, the individual studies can be weighted when the synthesis of result is performed and it offers the guidelines for future research and interpretation of findings. In addition to that, the quality of the study dominates the bias minimization and the maximization of internal and external validity (Khan et al., 2001). A well-formed quality assessment questionnaire is presented as follows (Kitchenham et al., 2009):

- Are the review's inclusion and exclusion criteria described and appropriate?
- Is the literature search likely to have covered all relevant studies?
- Did the reviewers assess the quality/validity of the included studies?
- Were the basic data/studies adequately described?

The four questions above are applied to check if the review addressed the two research questions which were elicited in Section 2.1.1. The question scores 1 if the criteria was fully addressed, and a goal of 0.5 is assigned if partially tackled or 0 for not tackled at all. The total points of each research question is 4 and any that scored more than 3 are considered to be adequately reviewed. The final points of RQ1 and RQ2 in this research is illustrated in Table 2.1.

2.1.5 Data Extraction

The objective of data extraction is to design forms to accurately record the information researchers obtain from the primary studies and the forms should be able to collect all the information needed to address the research questions and study quality criteria (Keele et al., 2007). Therefore, a standardized form should include name of the reviewer,

	RQ1 (Scored 3)	RQ2 (Scored 3)
Criterion 1	Scored 1. We included as many resources about maintainability as possible. Meanwhile, inappropriate and irrelevant studies were removed by conditions.	Scored 1. Based on the reliable data of RQ1, the measures used in RQ2 were properly filtered.
Criterion 2	Scored 1. We followed systematic steps to carry out the search process, it is likely to cover as many relevant studies as possible.	Scored 1. This is also based on the result of RQ1 so it has the same score as RQ1.
Criterion 3	Scored 0.5. The studies are too much so we paid more attention on the abstract and conclusion, which might not identify the quality and validity precisely.	Scored 0.5. It was also based on the result of RQ1, so it has the same result as RQ1.
Criterion 4	Scored 0.5. We tried to describe objective and non-ambiguous results. However, as English is not the first language of the author, there might exists inevitable language bias and ambiguities.	Scored 0.5. The result might not have been carried out properly due to language capability of the author.

Table 2.1: Quality of Review for Research Questions

date of collection, title, author, journal, publication details and space for additional notes. It is also suggested that a data extraction process should be performed by two or more researchers working collaboratively to assure the consistency. Alternatively, a second extraction of some random primary studies can be performed to do a test-retest process to achieve the same goal. The content of the extracted data will involve the following information (Kitchenham et al., 2009):

- The source (journal or conference) and full reference.
- Classification of the study Type (SLR, Meta-Analysis MA); Scope (Research trends or specific technology evaluation question).
- Main topic area.
- The author(s) and their institution and the country where it is situated.
- Summary of the study including the main research questions and the answers.
- Research question/issue.

- Quality evaluation.
- How many primary studies were used in the SLR.

2.1.6 Data Synthesis

Keele (2007) defines the data synthesis as collating and summarizing the results of the included primary studies. Either qualitative or quantitative synthesis is accepted but a quantitative summary with descriptive process will be a better approach. For the qualitative synthesis, the relevant information such as intervention, context and outcomes should be listed in a manner consistent with the research questions. Comparisons of similarities and differences between different study outcomes are supposed to be highlighted and structured in tables, and the recognition of consistencies of different studies is important in this process.

Kitchenham and Barbara (2004) explain that quantitative synthesis should be performed so that all study outcomes should be presented in a comparable way. A commonly used mechanism is forest plot which displays the means and variance for the difference of each study. The meta-analysis (Mulrow & Oxman, 1997) also offers a statistical way to obtain a quantitative synthesis with the help of techniques. In this review, both qualitative and quantitative comparisons were carried out on the maintainability metrics, investigations and the results.

2.2 Software Maintainability Metrics

ISO 25010 (2011)/25023 (2016) defines maintainability of software as the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers. IEEE (1993) similarly defines maintainability as the ease with which maintenance activities can be made. Various types of maintenance could occur after the delivery of a software product, such as correcting faults, improving

performance or other attributes and adapting to a changed environment. Maintainability demonstrates the difficulty of performing those maintenance activities.

2.2.1 Maintainability Metrics Evolution

Belady and Lahman (1976) were the first to define the concept of maintainability. In the period of 1960s to 1970s when procedure-oriented development was extensively used, researchers and practitioners paid more attention to make better programs rather than maintainable software (Swanson, 1976). Some frequently referred metrics which are used for measuring procedural programs were proposed during this period e.g. McCabe's Cyclomatic Complexity (McCabe, 1976) and Halstead Volume (Halstead et al., 1977). The effectiveness and efficiency of these metrics have been evaluated through empirical studies and Rombach (1987), Wake and Henry (1988) conclude that software maintainability can be predicted by metrics. From 1990 onward, object-oriented language started to become popular and Rombach (1990) argued that the metrics used to measure procedure-oriented programs are not appropriate to be applied on Object-Oriented ones blindly. For example, the Object-Oriented concepts of classes, and message passing cannot be characterized by any of the metrics mentioned above (Li & Henry, 1993).

In 1990s, thus, Object-Oriented design metrics based on characteristics i.e. encapsulation, inheritance and polymorphism were studied and presented such as the CK metrics (Chidamber & Kemerer, 1994), the suites proposed by Li and Henry (1993), Chen and Lum (1993), Lorenz and Kidd (1994), the MOOD metrics (Abreu & Carapuça, 1994) etc. Further, other investigations which aims to evaluate and verify the effectiveness, efficiency and accuracy of these metrics e.g. A critique of software defect prediction models (Fenton & Neil, 1999), maintenance and development estimation accuracy (Kitchenham, Pfleeger, McColl & Eagan, 2002) were carried out as well. Malhotra

and Chug (2016) state that the empirical investigations on software design metrics and maintainability successfully identified their correlations. From 2000 onward, researchers started to have the consensus that the maintainability can be measured by the time-consumption and changes made in the operations during maintenance activities. However, design metrics are more effective in the prediction of quality in the early stage of software development life cycle (SDLC), as it is more valuable to forecast and avoid the risks earlier than handling the maintenance later (Basili, Briand & Melo, 1996).

From 2000 onward, the research into Machine Learning (ML) brought software maintainability measures to a higher stage. By applying ML, the measures attempt to construct more complex, dynamic and intellectual strategies to predict and assess software maintainability. Aggarwal, Singh, Kaur and Malhotra (2006) utilized a model based on Artificial Neural Network (ANN) on maintainability prediction with Object-Oriented metrics. Zhou and Leung (2007) conducted a similar study to predict maintainability based on machine learning multivariate regression model. Furthermore, Malhotra and Chug (2012) compares the performance of applying several Machine Learning (ML) models to assess maintainability, and some new approaches such as code smell was utilized by ML to measure the maintainability of a system (Di Nucci, Palomba, Tamburri, Serebrenik & De Lucia, 2018). Compared to statistical measurements, these advanced techniques were empirically proved to have better performance (Malhotra & Chug, 2016).

In summary, common trends could be seen in the evolution of maintainability metrics. Oman and Hagemeister (1992) proposed Maintainability Index (MI) which comprises the procedural metrics i.e. Halstead's volume, McCabe's cyclomatic complexity, Number Of Comments (NOC) and Lines Of Code (LOC) to measure maintainability. In addition, the MI, DIT (Depth of Inheritance Tree) (Chidamber & Kemerer, 1994), Cyclomatic Complexity (McCabe, 1976) and LOC were integrated in to Microsoft's

official development environment Visual Studio since 2005 to help developers to improve their code quality. In recent years, researchers and practitioners combine metrics above as well as some complex ones such as cyclomatic dependency, god component (Lippert & Roock, 2006) into a code smell program, and run the program with the aforementioned ML technique to automatically assess and predict maintainability from a bigger and holistic picture (Fontana, Mäntylä, Zanoni & Marino, 2016). On the other hand, approaches for carrying out maintainability metrics keep changing. Before 2000, metrics were applied manually or with the help of tools. Later after that, hybrid techniques which mix tools and ML were utilized. Contemporarily, ML is the main technique of assessing maintainability, as the complexity of modern software has increased exponentially.

2.2.2 Maintainability Metrics and Classification

ISO 25010(2011) externally outlines the quality model of maintainability as five sub-characteristics which are modularity, reusability, modifiability, analyzability and testability, while ISO 25023(2016) defines the metrics for each sub-characteristic internally. Those metrics formulates approaches to quantify each sub-quality, so that maintainability can be reflected by compositions of the sub-characteristics. However, the ISO measures require the assessors to measure the code base itself, the later maintenance activities, as well as the test environment establishment, which is supposed to be a long-term process.

The Object-Oriented design metrics proposed in 1990s were also considered to have close-knit correlations with maintainability, such as the CK metrics suite (Chidamber & Kemerer, 1994), Li and Henry (Li & Henry, 1993). These metrics aim at predicting maintainability from the internal aspects, as they usually require the code base which has been implemented partially to measure the classes and relations between them. The

merits of these metrics are that assessors do not need to perform experiments in the later maintenance phase, but just need to apply them to the code bases in the early stage to get the predicted results. However, such prediction does not reflect the actual maintainability of a code base accurately.

Besides, some approaches provide methods to measure maintainability at the software architecture stage, which is even earlier than that in the design stage. For example, Bengtsson et.al (2004) state that the Architecture Level Maintainability Analysis (ALMA) provides a method to estimate maintainability of an architecture before implementation. The estimations are performed based on scenarios elicited by stakeholders, and architecture documentation such as the 4+1 views (Kruchten, 1995). Specifically, the potential maintenance efforts of the scenarios are estimated based on the architecture, so as to predict software maintainability. However, the uncertainty that arises with the ongoing progress of the project makes this method even harder to indicate the real maintainability of an application.

In summary, from the studies above, maintainability metrics vary in different stage of SDLC. Apart from that, the amount of metrics is also huge. A study of Saraiva, Soares and Castor (2013) demonstrates that there exist 568 maintainability metrics. Therefore, it is necessary to classify the metrics and identify the proper and applicable ones for this research. Generally, it can be carried out from multiple dimensions. For example, some metrics aim to predict while others measure the actual maintainability of the code base. Here the process of software development life cycle (SDLC) is followed, makes the classification as architecture, design, code and process. Each classification represents a different phase in SDLC, but following the successive order, shown as follows.

Architecture-level Metrics

In a general SDLC, the first step is usually to design the architecture, and the design usually starts with qualities elicited by the stakeholders. According to ISO 25010 (2011), maintainability is one of the eight qualities in the software product quality model. The assessment at the architecture level is to measure maintainability and its parallel qualities to indicate if it satisfied the expectation of the stakeholders.

As no implementation is performed in the architecture phase, the object that can be measured in this phase is more about the architectural documentation e.g. the 4+1 views (Kruchten, 1995) which includes the logical view, development view, process view and physical view. The scenario-based architecture assessment method (SAAM) (Kazman, Bass, Abowd & Webb, 1994) offers a means to measure the quality of an architecture by assessing the final version of the design of it. Bengtsson and Bosch (2003) explain that the scenario-based assessment is a practical approach and the empirical evaluation of this was presented in architecture-level modifiability analysis (ALMA) (2004), which provides systematic steps to estimate the effort in future maintenance tasks. The maintainability of an architecture can then be reflected by comparing the result with the history data.

Concentrating on modifiability, the ALMA usually involves more of the development team, marketing and product as stakeholders. It can be applied from a scenario classification to elicit the scenarios related or from a scenario first, then by carrying out the categorization of the scenarios. Besides that, techniques are provided to select relevant and eliminate redundant scenarios. Ionita, Hammer and Obbink (2002) state that the ALMA does not provide approaches to decide the accuracy of the analysis result and it cannot reason about the accuracy of maintenance prediction numbers. The scenario-based assessment might not provide accurate data for analysis.

Design-level Metrics

The design level metrics are mainly used to measure the classes and their relationships to a code base. The design phase usually happens after the design of architecture and this is where elements correlated with interface, classes and methods are carried out. With these concretes, multivariate metrics related at the class-level can be performed to produce reliable assessment data.

Chidamber and Kemerer (1994) presented the CK metrics suites which includes six metrics i.e. Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object classes (CBO), Response For a Class (RFC) and Lack of Cohesion in Methods (LCOM). The effectiveness and efficiency of which has been evaluated by researchers and practitioners in the last decades. The CBO and DIT metrics have significantly enhanced the probability of finding a fault in test (Briand, Wüst, Daly & Porter, 2000) which means the testability is strengthened. Dubey and Rana (2011) found that the maintainability of systems can invariably be improved by achieving lower CK values through the assessments on real projects whereas high values of CK metrics increases the system complexity and adversely impact the maintainability, testability and reusability (Kulkarni, Kalshetty & Arde, 2010). However, Li (1998) argued that there were some ambiguity and deficiencies in the CK metrics by applying the metric-evaluation framework (Kitchenham, Pfleeger & Fenton, 1995) and he made supplements on CK with Number of Ancestor Classes (NAC), Number of Local Methods (NLM), Class Method Complexity (CMC), Number of Descendent Classes (NDC), Coupling Through Abstract Data Type (CTA) and Coupling Through Message Passing (CTM).

The MOOD metric suite (e Abreu, 1995) is another attempt to measure maintainability which emphasizes more about the internal assessments of a class. Method

Hiding Factor (MHF) and Attribute Hiding Factor (AHF) conform to the encapsulation principle of Object-Oriented programming in that they measure the degree of information hiding. Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) assess how many methods and attributes have been inherited for a subclass. Two other metrics, Coupling Factor (CF) and Polymorphism Factor (PF) are used to test the coupling between classes and the number of overridden methods in inheritance respectively. Harrison, Counsell and Nithi (1998) state that the six MOOD metrics are valid measurements under the theoretical framework (Kitchenham et al., 1995). Comparing with the CK metrics, the MOOD metrics provide a different approach of assessment of a system, which is complementary to the CK metrics commutatively. Another empirical study demonstrates that the MOOD metrics might have a strong influence in resulting software maintainability and reliability and this is significant for a project at the planning stage (e Abreu & Melo, 1996).

Besides the suites above, a large number of Object-Oriented design metrics were proposed and evaluated for the predictions and assessments of maintainability in 1990s. However, limited topics were seen according to the study of Saraiva, Soares and Castor (2013), where they surveyed 568 metrics but only five topics were related e.g. coupling, cohesion, inheritance, size and architecture. They explain that the maintainability metrics frequently referred are rare, and they statistically synthesized the top extensively used metrics, from which 9 of 12 metrics is about "Coupling and Cohesion" while 3 ones are about inheritance and size, as demonstrated in Table 2.2.

Code-level Metrics

The measurements in this phase are based on the detail code base, and the metrics are usually carried out after or during the implementation of a system. The objects to be measured range from a variable definition and assignment, a statement, a piece of code, a method and a class, so the metric result of a code base is more detailed and the

Metric	Description	Topic	#Occurrences
CBO	Coupling Between Objects Classes	Coupling	43
DIT	Depth of Inheritance Tree	Inheritance	39
LCOM	Lack of Cohesion in Methods	Cohesion	39
NOC	Number of Children of a Class	Inheritance	39
RFC	Response For a Class	Coupling	36
WMC	Weighted Methods Per Class	Cohesion	33
LOC	Lines of Code	Size	25
MPC	Message Passing Coupling	Coupling	17
CC	Class Coupling	Coupling	16
DAC	Data Abstraction Coupling	Coupling	14
TCC	Tight Class Cohesion	Cohesion	13
LCC	Loose Class Cohesion	Cohesion	10

Table 2.2: Maintainability Top Used Metrics

amount of data increases drastically which makes the analysis more difficult to perform, especially on large systems.

Before 1990 when procedural programming was popular, maintainability metrics such as Cyclomatic Complexity (McCabe, 1976) and Weighted Halstead Effort or Volume (Halstead et al., 1977) were mainly code-oriented. Meanwhile, Adamov and Baumann (1987) reviewed some procedure-oriented metrics such as Function Points (FP) (Albrecht, 1979), Chapin's Q (Chapin, 1979), Information Flow (Henry & Kafura, 1981), Nesting Level (W. A. Harrison & Magel, 1981) etc. These metrics still play an non-substitutable role in measuring and indicating the quality of Object-Oriented program contemporarily, as they can be applied to any implementations of methods in classes.

Oman and Hagemeister (1992) describe that the factors influence software maintainability can be categorized and measured by different metrics from different aspects, and the result can be combined into a single index of maintainability. They presented maintainability index (MI) which comprises Halstead's volume, McCabe's cyclomatic complexity, number of comments (NOC) and lines of code (LOC) to measure a code base from holistic aspects.

In recent years, researchers and practitioners intend to use code smells to evaluate maintainability and detect the error-prone modules of a code base. According to Rasool and Arshad (2015), the most widely researched and used code smells are metrics-based and search-based. The former one is based on source code metrics which integrates some proper measures such as LOC and Cyclomatic Complexity to the source code, whereas the latter one utilize advanced techniques such as machine learning to detect how source code deviates from standard code design and implementations. The search-based code smell stands at a higher point to achieve a big picture of the code base from different aspects. With machine learning technique, it performs more accurately that the principle violation variables, interfaces, classes and components can be detected. In addition to that, more complex deficiencies and relationships such as Unstable Dependency (Martin, 2002), Dense Structure (Sharma, Fragkoulis & Spinellis, 2016) and Scattered Functionality (Garcia, Popescu, Edwards & Medvidovic, 2009) can be further identified.

Substantial empirical studies have demonstrated the effectiveness and efficiency of the metrics at design and code level, and they are appropriate attempts to measure and predict the maintainability of a software product. Moreover, consistencies can be seen in the progress of code metrics and some of them are still being used for the assessments of modern code bases, only that advanced techniques needs to carry out to automatically measure the quality due to the increasing complexity of modern software.

Process-level Metrics

The three types of metrics above are about predicting maintainability by measuring the software at different concrete levels. While the process metrics measure to what extent the code is maintainable directly. This phase happens after the delivery of the software product in SDLC, normal maintenance work is carried out and implementation of different types of tasks are required. The data recorded in this phase demonstrates

the maintainability of the code base more accurately.

According to Kaur and Singh (2015), three maintenance types are considered in SDLC: corrective maintenance is about fixing emergency program and debugging routine; adaptive maintenance responses to technology updates; and perfective maintenance aims to improve documented and requested enhancement. The maintenance effort depends on the number of changes and costs e.g. LOC changed, time-consumption, amount of human-effort requires. Banker, Kauffman and Kumar (1991) proposed the Annual Change Traffic (ACT) model, which was defined as the fraction of the software product's source instructions which undergo change during a typical year, either addition or modification. However, the LOC is too weak to represent the maintenance efforts and some factors such as understanding business rules, relevance of external systems could affect the assessment (Ahn, Suh, Kim & Kim, 2003).

Belady and Lehman (1972) describe that when measuring maintenance efforts, the considerations on productivity effort, software design and documentation complexity, and the extent of familiarity with the code base should be took. Jorgensen (1995) explains that cause of the task, code changing degree, code operation type, and maintainers' confidence would affect the maintenance effort. Technical constraints such as response time and platforms, maintenance tools and techniques such as development methodology and case tools, factors related to personnel such as number of programmers, experience directly or indirectly influence the estimation of efforts (Desharnais, Pare, Maya & St-Pierre, 1997). Generally, various factors might impact the maintenance efforts when considering process-level assessment, however, the studies above demonstrate that the outputs of the measurements are inevitably correlated with maintenance efforts consumption.

2.3 Metrics Refinement and Assessment of ALA

Spray and Sinha (2018) state that the maintainability improvements brought by ALA increases several-fold on the re-development of an embedded software carried out by a student comparing with the legacy code base. However, no evidence shows that ALA would reduce the maintenance efforts in other kinds of commercial projects, like desktop or mobile applications. Furthermore, they explained the conformity of ALA to the ISO 25023 (2016) quality metrics but there lack of evidence and evaluation on that. To assess the conformity and maintainability of ALA, it is necessary to refine the maintainability metrics to perform more accurate assessments and comparisons on the architecture and code base. Further on that, preliminary assessments of ALA is carried out by comparing with the refined metrics to reflect the mechanisms that ALA supports maintainability.

The refined metrics are based on the ISO 25010 (2011)/25023 (2016) quality model. The ISO 25010 decomposes maintainability as *modularity, reusability, analyzability, modifiability* and *testability*, while the ISO 25023 defines the specific metrics of each sub-characteristic. The decomposition of maintainability simplifies the complexity of performing assessments on a software code base with quantitative measurements. Wu (2018) states that the software architecture index framework they developed in a multinational company which based on ISO 25010 has been adopted for more than 100 software products, and it successfully improved their code quality and maintainability to a great extent. However, some of the metrics of the sub-characteristic defined by ISO 25023 is not applicable for the measurement of a code base. For instance, the modularity metrics does not give a clear definition of how to measure coupling. Hence, some additional metrics are considered and refined to enhance and clarify the assessments, which are explained respectively and specifically, shown as follows.

2.3.1 Modularity Metrics

The concept of modularity is used primarily to reduce complexity by breaking a system into varying degrees of interdependence and independence across and "hide the complexity of each part behind an abstraction and interface" (Baldwin & Clark, 2000). In software engineering, ISO 25010 (2011) defines modularity as "*the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components*", and modularity measures defined by ISO 25023 (BSI ISO, 2016) are used for measuring such degrees.

Table 2.3 illustrates ISO 25023 modularity metrics, the refined metrics, and the assessments of how ALA conforms to these metrics. The ISO coupling components metric requires the assessors to count the number of components which have no impact on others, or no coupling with others. The cyclomatic complexity adequacy is based on McCabe's complexity theory (McCabe, 1976). Considering that the ISO 25023 does not give the measurements to assess whether a component is independent, some coupling metrics aforementioned in Table 2.2 are studied and refined to measure the coupling of a code base. The one selected here is CBO, and the reason for that is explained in the table.

The mechanisms of ALA conforms to a great extent to component coupling metrics. Among the four layers of ALA, developers need to concentrate on the first three layers merely. The application layer depends on the rest layers, so it inevitably has coupling with all the components. However, as the primary elements of ALA, assets of domain abstraction layer depend on no assets at the same layer, neither does it in programming paradigm layer. Thus, ALA is supposed to have low coupling values so its modularity is reversely high. In terms of Cyclomatic Complexity, it is difficult to tell whether ALA has a proper CC value at a predictive stage, as such metrics is correlated with the actual design and implementation. However, Table 2.3 demonstrates that Cyclomatic

ISO 25023 (2016)	1. Coupling of components: X = A/B - A = Number of components which are implemented with no impact on others; - B = Number of specified components which are required to be independent; 2. Cyclomatic complexity adequacy: X = 1– A/B - A = Number of software modules which have a cyclomatic complexity score that exceeds the specified threshold; - B = Number of software modules implemented.
Refined Metrics	1. CBO (Coupling Between Objects Classes) (Chidamber & Kemerer, 1994). This metric includes two aspects of coupling i.e. afferent coupling which means the number of components depends on a component, and efferent coupling which means the number of components that a component depends on. CBO is chosen because both the objective of CBO and the main unit of ALA are classes, so it would be easy to apply CBO to the code base. Besides, CBO explicitly shows the relations between classes and interfaces which gives a intuitionistic reflection of whether a component has coupling with others. 2. CC (Cyclomatic Complexity). CC is used for the measurement of cyclomatic complexity adequacy. This metric is based on McCabe's complexity (McCabe, 1976), and it reflects the complexity of the control flow in a function or a class.
ALA (Spray & Sinha, 2018)	1. Coupling: - zero coupling between domain abstractions; - zero coupling between programming paradigms; 2. Cyclomatic Complexity: - cyclomatic complexity can be dealt by hierarchical layer-based decomposition; - cyclomatic complexity is reduced because modules based on abstractions naturally have a single responsibility.

Table 2.3: Modularity Metrics and Conformity of ALA

Complexity of ALA could be dealt with and reduced due to the design mechanism of its hierarchical structures and domain abstractions. Hence, we assume that cyclomatic complexity of ALA is considerably pertinent.

2.3.2 Reusability Metrics

Reuse is the use of previously acquired concepts and objects in a new situation. Reusability is a measure of the ease with which one can use those previous concepts and objects in the new situation (Prieto-Diaz & Freeman, 1987). In ALA, all abstractions and programming paradigms are designed for reuse. Similar with the situation of modularity, ISO 25023 does not provide a clear definition of what characteristics a reusable asset should have. To measure that, metric suites of CK (Chidamber & Kemerer, 1994) and MOOD (e Abreu, 1995) provides different approaches to investigate the classes from multiple aspects e.g. inheritance and encapsulation. However, some of these metrics are not appropriate for ALA. For example, there is no inheritance in ALA and the application of all inheritance metrics merely result meaningless values.

Considering the mechanisms of ALA, we refined metrics for measuring its reusability, as illustrated in Table 2.4. CBO, LCOM, WMC and IT are used to assess the potential that an asset could be reused in future development. Among which CBO, LCOM, WMC intends to predict the reusability, while IT reflects the actual times an asset has been reused (instantiated) in current code base. For coding rules conformity assessment, the result of Layer Violations (LV) provides evidence to demonstrate if dependencies happened in a right way, as well as if domain abstractions conformed to the defined interfaces of programming paradigms. Sarkar et al., (2006) state that Layer Violations can be categorized as back-call, which means a lower layer calls the layer up to it, skip-call which means a upper layer calls at least two layers down to it, and cyclic dependencies which means two layers call each other bidirectionally. In ALA, the skip-call is allowed between upper layers and lower layers, so we removed the skip-call measure but keep the rest two. In addition, the parallel-call should be considered to ensure no dependencies between classes or interfaces on the same layer.

Mechanisms of ALA is assessed through refined metrics. CBO and NOC could

ISO 25023 (2016)	1. Reusability of assets: X = A/B - A = Number of assets which are designed and implemented to be reusable; - B = Number of assets in a system; 2. Coding rules conformity: X = A/B - A = Number of software modules conforming to coding rules for a specific system; - B = Number of software modules implemented.
Refined Metrics	1. Reusability of assets: - CBO (Coupling Between Objects Classes). Excessive coupling weaken the encapsulation of a class and inhibits reuse (Laing & Coleman, 2001). - LCOM (Lack of Cohesion Methods). If a method referred more variables externally, it is more specific to the application and less reusable (Chidamber & Kemerer, 1994). - WMC (Weighted Methods per Class). Large method number of a class demonstrates more specific functionality to application, limiting the possibility to reuse (Goel & Bhatia, 2012). While a class with single and simple responsibility has a higher chance to be reused. - NOC (Number of Children). More children means more reuse (Chidamber & Kemerer, 1994). In ALA, this indicates interfaces reusability, as there is no class inheritance. - IT (Instantiated Times). The actual reused times of a class. 2. Coding rules conformity: - LV (Layer Violations) (Sarkar, Rama & Shubha, 2006). The layered architecture makes the dependencies only happening from up to down unidirectionally in ALA. Higher layer violations means lower coding rules conformity.
ALA (Spray & Sinha, 2018)	1. Reusability of assets: - Reusability increases typically by an order of magnitude as we go down each layer. - Two layers are dedicated to reuse, layer 2 for reuse at the domain level, and layer 3 for reuse at the programming paradigm level. - Interfaces and domain abstractions are reusable types. 2. Coding rules conformity: - Domain abstractions conform to coding rules via interfaces. - The interfaces that exist for connecting domain abstractions are at the reuse level (and abstraction level) of the framework layer.

Table 2.4: Reusability Metrics and Conformity of ALA

result better result because of the low coupling design and highly reusable programming paradigms. IT is supposed to be high with ALA's mechanism, as each asset at

abstraction layer and paradigms layer are abstract, and the application layer instantiates the abstractions for multiple times to construct and express various requirements. According to Spray and Sinha (2018), the code at the application layer usually occupies 1% of the total amount of code, and any changes of requirements would be directly expressed at the application layer instead of modifying the deeper layers. Such mechanism demonstrates that there might exist 99% of the code that is reusable. In terms of coding rules, it is indispensable for an ALA code base to conform completely to the layered architecture to achieve the expected reusability. However, as it requires the concrete code base to support applying the metrics, the conformity can only be evaluated at the implementation phase.

2.3.3 Analysability Metrics

ISO 25010 (2011) defines analysability as "*the degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified*". However, such concept is hard to be correlated with the measures defined by ISO 25023 (2016), where it concentrates more on system logs and diagnosis functions, as shown in Table 2.5. Besides, ALA does not define or outline any rules about system log and diagnosis functions and it is meaningless and difficult to apply those measures to ALA. Thus, the refinement of analysability metrics is mainly considered from its definition, which can be summarized as two goals: (1)The ease of ripple effects identification of an intended change; (2)The ease to locate the parts that need to change with specific maintenance intention; We came up with the refined metrics list for measuring analysability, as illustrated in Table 2.5.

Whether ripple effects of a modification on a class happens usually depend on if

ISO 25023 (2016)	1. System log completeness: X = A/B - A = Number of logs that are actually recorded in the system; - B = Number of logs for which audit trails are required during operation; 2. Diagnosis function effectiveness: X = A/B - A = Number of diagnostic functions useful for causal analysis; - B = Number of diagnostic functions implemented; 3. Diagnosis function sufficiency: X = A/B - A = Number of diagnostic functions implemented; - B = Number of diagnostic functions required.
Refined Metrics	- CBO (Coupling Between Object Classes). This metric is used for measuring the ripple effects of any changes on a class or an interface (Chidamber & Kemerer, 1994) (the first and second goal). - LCOM (Lack of Cohesion Methods). Similar with CBO, but stands at the methods level. A method which has less reference of other variables out of it has less side effects when making a modification, so it increases the analysability (Chidamber & Kemerer, 1994) (the first and second goal). - LOC (Lines of Code) (Albrecht & Gaffney, 1983). If an asset had more lines of code, the difficulty of analysis activities might increase (the second goal). - CP (Commenting Percentage) (Steidl, Hummel & Juergens, 2013). Proper comments helps maintainers locating where to change, enhance the analysability of a code base (the second goal).
ALA (Spray & Sinha, 2018)	- requirement changes are overlaid first on the top-level application, with changes decomposed and localized to interfaces and domain abstractions. - Use of abstractions (rather than just modules) together with the emphasis on knowledge dependencies rather than run-time dependencies.

Table 2.5: Analysability Metrics and Conformity of ALA

there existed other classes which has coupling with it. For example, if a large number of classes had dependencies on a single class, then changes on this class would probably impact those has coupling with it. If there were more classes depend on those impacted classes, then more ripple effects might be caused. Thus, the coupling metric CBO is considered as a measurement of assessing the possibility of arising ripple effects. A lower CBO value of a class means less coupling and it would be easier to identify ripple effects of changes. CBO also works on the second goal to identify the relationships between classes. Besides the metrics LOC, CP and LCOM can be used to measure the

internal side of a class i.e. properties, methods.

It is not feasible to carry out the assessment of ALA's analysability in current phase. Among the metrics refined, the only predictive one is CBO due to the design mechanism of ALA, and it has been explained in the previous sections. The other metrics are determined by the way the code base is implemented. Moreover, Spray and Sinha (2018) do not give clear explanations of the two goals for analysability we identified before, neither of the ISO 25023 measures. They focus more on requirements decomposition and dependencies, as shown in Table 2.5. Therefore, we assume that analysability of ALA might not be ideally tackled and the analysability of it is low.

2.3.4 Modifiability Metrics

Modifiability is conceptualized as "*the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality, and it is influenced by modularity and analysability*" (BSI ISO, 2011). Measures defined by ISO 25023 (2016) can be directly applied on a code base, as they are all about the ease of carrying out concrete maintenance activities with real intended modifications e.g. deficiencies correctness, new feature implementations. Thus, it is not necessary to refine metrics for modifiability, but just compare ISO measures with ALA's mechanisms, as shown in Table 2.6.

Modifiability cannot be directly assessed under such circumstances. As aforementioned, measures presented by ISO 25023 mainly work on a concrete code base, and real maintenance activities are required in the assessments. Apart from that, Spray and Sinha (2018) did not give clear clues of how ALA conforms to the measures, but more on explaining the independence of abstractions and interfaces, as shown in Table 2.6. Hence, prediction of ALA's modifiability primarily comes from the combination of modularity and analysability, as stated by ISO 25023 (2016). Previous assessments

ISO 25023 (2016)	1. Modification efficiency: $\sum(A_i/B_i)/n$ - A_i = Total work time spent for making a specific type of modification i; - B_i = Expected time for making the specific type of modification i; - n = Number of modifications measured, i ranges from 0 to n; 2. Modification correctness: X = 1 – (A/B) - A = Number of modifications that caused an incident or failure within a defined period after being implemented; - B = Number of modifications implemented; 3. Modification capability: X = A/B - A = Number of items actually modified within a specified duration; - B = Number of items required to be modified within a specified duration.
ALA (Spray & Sinha, 2018)	- Domain abstractions have zero coupling. - Domain abstractions and interfaces can be checked individually. - Module dependencies are always on interfaces that are at least one abstraction level more stable.

Table 2.6: Modifiability Metrics and Conformity of ALA

demonstrate that modularity of ALA is considerably high while analysability of ALA is low, so we assume that modifiability of ALA is medium.

2.3.5 Testability Metrics

ISO 25010 (2011) defines software testability as "*the degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met*". The measures of testability is not directly reflected and defined in ALA, so we refined the metrics from the definition of testability. In this case, we mainly consider the *ease of test criteria establishment and execution*, we refined the metrics as CBO, LOC, WMC and LCOM, as illustrated in Table 2.7.

For testability assessment in this phase, measures defined by ISO 25023 (2016) is not correspondingly explained in ALA, as shown in Table 2.7. However, we can specify the ease of tests of ALA from its mechanisms and testability definitions. First, test criteria/functions are easy to establish for both unit test and integration test. Domain

ISO 25023 (2016)	1. Test function completeness: X = A/B - A = Number of test functions implemented as specified; - B = Number of test functions required; 2. Autonomous testability: X = A/B - A = Number of tests that can be simulated by stub among the tests which depend on other systems; - B = Number of tests which depend on other systems; 3. Test restartability: X = A/B - A = Number of cases in which maintainer can pause and restart executing test run at desired points to check step by step; - B = Number of cases in which executing test run can be paused.
Refined Metrics	- CBO (Coupling Between Object Classes) (Chidamber & Kemerer, 1994). Less coupling with other components makes it easier to write an unit test case and run it. - LOC (Lines of Code) (Albrecht & Gaffney, 1983). It is about the size of the code base, a class with bigger size is supposed to be harder to test. - WMC (Weighted Methods per Class) (Chidamber & Kemerer, 1994). It measures the cyclomatic complexity of a class. A higher value of WMC means the class is more complex, increase the effort to test. - LCOM (Lack of Cohesion Methods) (Chidamber & Kemerer, 1994). Similar with CBO, but stands at the method level inside a class. A method which has less reference of other variables will be easier to test, as less conditions need to be considered.
ALA (Spray & Sinha, 2018)	- Domain abstractions can be tested individually. - Internal working of domain abstractions can be tested with straightforward integration tests by wiring each possible combination of abstraction. - Application can be tested easily using mocked or modified versions of I/O abstractions.

Table 2.7: Testability Metrics and Conformity of ALA

abstractions are independent units that can be tested individually, as well as being combined to verify a specific function or requirement. Second, single and simple responsibility of domain abstractions makes it effortless to achieve test goals, whether with autonomous test or manual test. Third, small sizes of unit tests and integration tests promotes to establish break points, and preliminary conditions for such break points. Hence, we conclude that testability of ALA is supposed to be significantly high.

2.3.6 Summary

Based on the quality model of ISO 25010 (2011) and ISO 25023 (2016), we carried out metrics refinements of each sub-characteristics of maintainability, making them directly applicable for ALA code base.

A preliminary assessment of ALA was carried out from its design mechanisms by comparing with ISO standards and the refined metrics, as illustrated in Table 2.8. We conclude that the modularity, reusability and testability of ALA is considerably high, as they conform to a great extent to the measures reviewed. Analysability is not well handled, because little common points could be identified between ALA mechanisms and the measures. So we assume that analysability of ALA might be low tentatively. Modifiability cannot be assessed in this phase because the metrics requires concrete maintenance activities. Thus we simply combined the result of modularity and analysability under the definition of ISO 25010, conclude that modifiability of ALA is medium.

For maintainability evaluations in the subsequent phases, metrics for modularity, reusability, analysability and testability can be applied directly on the code base. Thus this can be done once the code base is finished, and such measurements would produce data for predicting the potential efforts for future developments. However, to measure the actual performance of modifiability, maintenance activities needs to be carried out under real commercial environment. Such activities would generate data for analysing the actual modifiability of a code base.

2.4 Conclusion and Limitations

In this chapter, studies related to maintainability, metrics of it and its sub-characteristic were systematically reviewed. We specifically highlighted maintainability measures and

	Metrics	ALA's Mechanism	Result
Modularity	1. CBO 2. CC	1."Zero coupling" between abstractions; "zero coupling" between programming paradigms. 2.Cyclomatic Complexity can be dealt by single-responsibility domain abstractions.	High
Reusability	1. CBO 2. LCOM 3. WMC 4. NOC 5. IT 6. LV	1.Same as CBO in Modularity. 2.Depends on how many properties exist in a domain abstraction and how they are related, determined by implementation. 3.The CC of methods, would be lower than that in a class. 4.The NOC of programming paradigms would be significantly high. 5.All domain abstractions would be instantiated by more than one time. 6.ALA restricts the responsibility of layers, and theoretically LV would be 0.	High
Analysability	1. CBO 2. LOC 3. CP 4. LCOM	1.Same as CBO in Modularity. 2.A single-responsibility domain abstraction would have considerably reasonable LOC, determined by implementation. 3.The domain abstractions would be properly commented, determined by implementation. 4.Same as LCOM in Reusability.	LOW
Modifiability	Not Refined	As the measures require concrete maintenance activities, and ISO 25010 (2011) explains that Modifiability is influenced by Modularity and Analysability, we simply assume Modifiability of ALA is medium.	Medium
Testability	1. CBO 2. LOC 3. WMC 4. LCOM	1.Same as CBO in Modularity. 2.Same as LOC in Analysability. 3.Same as WMC in Reusability. 4.Same as LCOM in Reusability.	High

Table 2.8: Preliminary Assessment of ALA

the preliminary assessment of ALA by comparing its mechanisms with those measures. Finally we came up with the answers for the two research questions.

2.4.1 Answering Research Question 1 and 2

Through the literature review, the two research questions can be answered:

RQ1. What are the most relevant measures for assessing maintainability?

RQ2. Which mechanisms in ALA support these measures?

In terms of RQ1, ISO 25010 (2011) quality model decomposes maintainability as modularity, reusability, analysability, modifiability and testability, while ISO 25023 (2016) provides measurements for the sub-characteristics respectively. The ISO models systematically involves factors that have impact on maintainability in the SDLC and the metrics it defined are considerably relevant measures for assessing maintainability. Besides, it explicitly demonstrates the characteristics and mechanisms of a maintainable code base, or an architecture. However, as some measures of ISO 25023 cannot be directly applied on a code base, we reviewed and refined more general maintainability metrics e.g. McCabe's Cyclomatic Complexity (McCabe, 1976), CK Metric Suite (Chidamber & Kemerer, 1994), MOOD Metric Suite (Abreu & Carapuça, 1994) to make them applicable on a code base under the ISO quality model, and these refined metrics are correlated with maintainability as well.

With respect to RQ2, through the analysis of ALA's mechanisms and the identified measures of a maintainable code base, we conclude that ALA potentially and ideally supports modularity, reusability and testability to a great extent. Analysability might not be well handled in ALA, as no straightforward evidence shows ALA conforms to the refined metrics, neither for ISO standards. Modifiability cannot be measured without the code base and maintenance tasks, thus we speculated that it is at a medium level due to the performance of modularity and analysability. Overall, because most of the sub-characteristics of maintainability is well supported by ALA, we conclude that ALA code bases have considerable maintainability.

2.4.2 Limitations of This Literature Review

There exist multiple factors that might have influence on the interpretation of the findings in this literature review, and they are listed as follows:

1. The size of the studies is limited. Given the limited time to conduct the SLR, the lack of an even deeper analysis can be seen as a potential limitation of these findings.

2. Bias might not be removed due to the cultural background as English is the second language of the author that some studies might not be correctly understood and interpreted in the review.

3. A lack of prior studies on ALA makes the review having limited knowledge on the background and applications of this architecture.

Chapter 3

Methodology

This chapter discusses the methods used to answer each of the following five research questions:

RQ1. What are the most relevant measures for assessing maintainability?

RQ2. Which mechanisms in ALA support these measures?

RQ3. How can the process of re-architecting an existing C# application using ALA be generalised for use in future projects?

RQ4. How do the existing implementation of the C# application and the new re-implementation as a result of answering RQ3 compare when assessed for maintainability using the measures identified in RQ1?

RQ5. How well does the assessment carried out in RQ4 relate to the expected enhancements in maintainability from ALA (as identified in RQ2)?

For RQ1 and RQ2, a systematic literature review was conducted (see Chapter 2). The rest of this chapter is organized to explore the approaches for answering RQ3–5. Section 3.1 explains the process followed to select a research method. This process resulted in choosing a mixed-methods strategy, as well as specific ways to generate, gather and analyze data. Section 3.2 describes the method for re-architecting and implementing the ALA code base to assure that this research is reproducible. Finally,

Section 3.3 provides concluding remarks with additional information about the validity of data and experiments.

3.1 Selection of Research Method

General research methods involve qualitative, quantitative, or mixed-methods studies (Creswell, 2014). In this case, as we aim to achieve different research goals for RQ3, RQ4 and RQ5, a mixed-methods strategy which integrates both qualitative and quantitative methods was considered. However, according to Easterbrook et al.,(2008), there exist a lot of concrete methods which can be used for specific researches. Thus, we explored a number of options before choosing specific methods for this research:

Case Studies is particularly helpful in the situation where some specific problems and phenomena that need to be studied and explained in great-depth (Noor, 2008). A case study might not address the generalizability well, but the data being collected could be abundant.

Design Science works specifically when an artefact or a recommendation is considered to be the desired research goal, and such research could be carried out in either an academic or an organizational environment (Dresch, Lacerda & Antunes, 2015).

Controlled Experiments and Quasi-Experiments are widely applied in science research. They allow us to investigate how various variables are related, and whether there exists a cause-effect relationship between them based on some testable hypothesis (Easterbrook et al., 2008).

Survey Research refers to a set of methods which focuses on both qualitative and quantitative analysis, where data from a large number of sources would be collected through questionnaires, interviews, published and unpublished statistics, and the data were analysed with statistical techniques (Gable, 1994).

Ethnographies are a scientific approach to discovering and investigating social and

cultural patterns and meaning in communities, institutions and other social settings (Schensul, Schensul & LeCompte, 1999).

Action Research produces highly relevant research results, because it is grounded in practical actions, aimed at solving an immediate problem situation while carefully informing theory (Baskerville, 1999).

3.1.1 Our Approach

Considering the characteristics of the listed methodologies above, in this case, we mixed *Case Study* and *Quantitative Experiments* to achieve the research goals of RQ3, RQ4 and RQ5. The rationale of the selected methods to answer the corresponding research questions is explained in Table 3.1.

RQ	Methodology	Rationale
RQ3	Case Study	1. ALA is a reference architecture which was proposed and explained on its design mechanisms by Spray and Sinha (2018). A method to develop such kind of application needs to be established on a concrete project; 2. Since we don't have many cases for this new architecture, a case study is the first step to explore the feasibility of ALA, and it would generate abundant data which can help infer a general method of ALA application development.
RQ4, RQ5	Quantitative Experiments	1. It is not feasible to compare the re-developed code base with the legacy one directly. However, the maintainability metrics refined in Section 2.3 resulted in quantitative data which makes the comparison of the two code bases possible. 2. The actual modifiability can be measured and reflected by carrying out real maintenance tasks on the two code bases respectively.

Table 3.1: Selection of Research Methods

This mixed approach helps to improve the depth of data productions and collections, analysis and interpretations from multi-perspectives, and compensates for the limitations

of each other (Creswell & Creswell, 2017).

3.1.2 Experiment Design and Data Interpretation

Experiment design structures the process of carrying out the experiments to address the research questions. To evaluate the maintainability of ALA, we need to construct an ALA code base with the requirements of the existing application, and apply the metrics refined in Section 2.3 to it as well as the legacy code base to gather data for analysis. Mason et al.(2003) and Wohlin et al. (2012) explain that experiment design usually involves critical stages:

1. Problem definition. This was identified through the five formulated research questions RQ1–5.

2. Experimental planning. A qualitative case study was selected to achieve the research goal of RQ3, and a quantitative experiment approach was selected to address RQ4 and RQ5 (Table 3.1).

3. Operation or execution. We crucially followed the method of ALA architecture design and software development method (Section 3.2), and used a tool to generate and gather the metrics result (Section 3.1.3).

4. Interpretation or statistical analysis. We carried out both qualitatively and quantitatively comparative analysis.

To answer the research questions, we simply carried out comparative analysis to report the findings. Comparison is a significant test of focused questions in which specific intentions can be evaluated by comparing the obtained data (Rosenthal, Rosnow et al., 1985). In this case, the interpretation needs to be both qualitative and quantitative:

RQ3 was answered qualitatively, because we mainly recorded the development process and steps to infer a method for ALA development under a general framework.

RQ4 was addressed quantitatively, as metrics are quantitative in nature which allows us to compare the two code bases in a straightforward manner.

RQ5 requires both qualitatively and quantitatively reasoning, as the result of experiments might either support or oppose the preliminary assessment result. A mixed interpretation was used to help us identify the merits and downsides of ALA.

3.1.3 Data Generation and Gathering

For RQ4 and RQ5, we needed to collect data from the legacy and the re-developed code bases by applying the metrics refined in Section 2.3. However, the metrics could not be applied manually due to the size of the code bases and the complexity of the identified metrics. The legacy code base includes more than 30,000 lines of code and more than 400 classes according to the metric tool of Visual Studio 2017. Further on that, some metrics such as Cyclomatic Complexity required us to investigate all the branches of each line of code. As such activities is impossible to complete manually in a limited duration, an automatic tool was considered to generate and collect data.

We compared three tools: `Designite` (Sharma, Mishra & Tiwari, 2016), `NDepend` (Smacchia, 2007) and Visual Studio Metric Tool (Table 3.2). All of them were Visual Studio plugins and specifically designed for C# code bases. We finally concluded that NDepend is more suitable and effective for this research for the following reasons:

1. `NDepend` integrated all the metrics identified in Section 2.3.

2. `NDepend` provides an overall view of the dependency relations of all the classes,

	Metrics	Other
Designite (Sharma, Mishra & Tiwari, 2016)	It computes more than 30 design metrics, but only 6 out of 12 metrics identified in Section 2.3 are included.	1. It is an intelligent tool as it integrates code smells based on machine learning. 2. It offers a rich set of visualizations for software developers, designers, and architects to visualize various issues affecting the maintainability of their software.
NDepend (Smacchia, 2007)	It implements more than 20 metrics, and 12 out of 12 metrics identified in Section 2.3 are included.	1. It provides an overall view of the dependency relations of all the classes, interface even methods, which provides a higher-level view of the code base. 2. It supports customized query based on programmable conditions, so the user can combine the possible metric result by programming.
Visual Studio Metric Tool	It includes 4 out of 12 metrics identified in Section 2.3.	1. An official plugin of Microsoft and it is profoundly integrated in Visual Studio Environment, it is free and easy to use.

Table 3.2: Selection of Metric Tool

interface even methods, which helps us to specify maintainability from an overall view.

3. `NDepend` supports customized query based on programmable query language, which provides more flexibility when generating data.

3.2 Software Re-development

This section explains the methods correlated with the requirements, architecture design and SDLC management. Section 3.2.1 describes the way to elicit the requirements and manage the correctness and consistency of the requirements. Section 3.2.2 presents the way of re-architecting the legacy application with ALA. In Section 3.2.3, the process management method of carrying out the re-development is discussed and determined.

3.2.1 Requirements Elicitation and Management

For re-architecting an existing application, the first goal was to make sure it has the same requirements as the legacy application. In this case, we took three measures to assure the correctness, comprehensiveness and consistency of the requirements:

1. The requirement and development documentations of the legacy code base, as well as the legacy application itself, were referred to carry out the design and implementation.

2. The maintainer of the legacy application participated in the design process to provide accurate and consistent information on the requirements.

3. During the design and implementation, no requirement was allowed to change when comparing with that in the legacy application.

3.2.2 ALA Architecture Design

Software architecture design usually includes a series of activities such as requirements analysis and decomposition, and finally results architectural documentations such as the 4+1 views (Kruchten, 1995). The architecture design of ALA is similar, but more specifically focuses on the design of *Programming Paradigms and Domain Abstractions* which are the design activities, and *Expression of Requirements* which is the architectural documentation (Chapter 4).

Spray and Sinha (2018) state that the architecture design of ALA intends to put architectural elements and requirements together and express them in a straightforward manner. This specific method made the output of an ALA architecture design a diagram which includes abstractions (classes), programming paradigms (interfaces) and wiring instances of abstractions (requirements). To achieve that goal, we critically followed

the design notions and principles of ALA, make sure that we did not introduce any technical and personal bias in the design phase.

3.2.3 Software Development Life Cycle Management

Despite of the *requirements* and *architecture* aforementioned, we need to apply a method to manage the software development life cycle to simulate the commercial environment of Datamars and manage the process of ALA software development. Such method helped to formalize the re-development process and generate evidence for generalizing an ALA development method for RQ3.

The basic idea is to choose an agile method due to its extensive use for commercial projects like Datalink, but we specifically selected Scrum (Schwaber, 1997) to drive the re-development, because:

1. Scrum is a representative methodology in the agile family and it is used widely in companies nowadays, which helps us to investigate the feasibility and maintainability of ALA under such circumstance.

2. Since Datamars also adopts Scrum to drives its commercial projects, using Scrum helps us to simulate such commercial environment and create comparative foundations for the maintenance experiments in the later phases of this research.

3. The Scrum method conforms to the original research plan in the very first step, where an iteration zero was used to design the architecture of Datalink with ALA.

3.3 Conclusion and Approaches Validity

In this chapter, we explored methods for carrying out the research activities. Basically, the whole research was under a Design Science framework (Figure 3.1). A hybrid

strategy was used to address each research question. We mainly adopted a case study to answer RQ3, and quantitative experiments to answer RQ4 and RQ5. In addition, we followed the ALA architecture design and a general software development method to manage the operation and execution. Besides, a metric tool (`NDepend`) was used to automatically generate and collect metrics data.

Problem Definition					
Motivation	***Object***	***Purpose***	***Perspective***	***Domain***	***Scope***
To improve the maintainability of commercial software	Maintenance efforts, maintainabilit y metric result	To evaluate the effectiveness and efficiency under commercial environment	Researcher	Analysing maintenance efforts and maintainabilit y metrics results for the two code bases	Single project
Planning					
Design		***Criteria***		***Measurement***	
Case Study: ALA architecture design Waterall and Scrum model		The key steps of developing an ALA application is recorded		Evidence about the key steps is provided	
Experiment Design: Apply maintainability metrics on the two code bases		NDepend v2019.2.6 was used to generate and collect metrics data		The metrics results of ALA is much more effective than the legacy code base	
Experiment Design: Carry out maintenance tasks on the two code bases		The maintenance efforts (hours) would be recorded respectively		The maintenance efforts of ALA is less than the legacy code base	
Operation					
Preparation		***Execution***		***Analysis***	
Studies of the legacy application to minimize the personnel capability of the two maintainers		Metrics data would be recorded by the software, while maintenance efforts would be recorded manually		Both qualitative and quantitative analysis would be used	
Interpretation					
Context		***Extrapolation***		***Impact***	
Comparative analysis		Analysis of sample for representativeness		Based on a real application and published studies of this research to be replicated by peers during a review	

Figure 3.1: Design Framework of This Research

3.3.1 Validity of Data

It is up to the researchers and research participants who attempt to build validity into the different phases of the research from data collection through to data analysis and interpretation (Zohrabi, 2013). Data validity is fundamental and indispensable for what concerned with the reliability and evaluations of the research objectives. Researchers usually use different instruments to collect data, so that the quality of these instruments is very critical because the conclusions researchers draw are based on the information they obtain using these instruments (Fraenkel, Wallen & Hyun, 2011). In this case, we adopted measures to assure the data validity:

1. Metrics were applied objectively with `NDepend` and data were recorded under the same metrics refined.

2. For data collection in the experiments, `NDepend` was used to generate and gather a united form of data for maintainability metrics which alleviates the difficulty for data comparisons and analysis.

3.3.2 Validity of Experiments

Both "applied" and "theoretical" experiments almost always have the goal of generalizing the results to populations that are comprised partly (or wholly) of measures of future behaviors (Lynch Jr, 1982). However, it is impossible to randomly assign the experimental population here. To reduce the bias of the results, we utilised a commercial development environment, under which we created the similar foundations as the legacy Datalink to carry out the experiments. The measures for experiments validity are:

1. We utilised a commercial environment for producing reliable data. As foremen-tioned, a systematic software development life cycle method was followed to

manage the development procedure, and the whole process was carried out in a real company (Datamars).

2. The design notions and principles of ALA was stringently followed, making sure that we did not introduce technical and personal bias to the code base to provide more reliable and valid population for the experiment and analysis stages.

Chapter 4

Re-developing Datalink with ALA

This chapter details the process of re-architecting and implementing the legacy Datalink with ALA. We carried out the corresponding activities i.e. architecture design, implementation and maintenance of the general process of a SDLC. We adopted the Scrum (Schwaber, 1997) method to drive each activity, which aimed to simulate the real environment of Datamars. Based on the whole re-development, we proposed a general method to develop ALA applications.

The rest of this chapter is organized as follows. Section 4.1 describes iteration zero which was carried out to re-architect Datalink with ALA based on the requirements of the legacy application. The architectural elements of ALA i.e. programming paradigms, domain abstractions and requirements expression were iteratively designed and documented. Section 4.2 describes the way of using Scrum to drive the implementation. We also explained the technical background that helps the implementation and gave some examples of the code. At the end of the implementation, we inspected the completeness of the re-developed ALA Datalink by comparing with the legacy one. Section 4.3 discusses the maintenance activities in the re-developed ALA code base. Section 4.4 summarizes the re-development and proposes a general way to develop ALA applications.

4.1 Architecture Design of ALA Datalink

Bass et al., (2003) states that *"software architecture design is the process that decomposes the requirements into structures or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them"*. For ALA Datalink, Spray and Sinha (2018) emphasized the importance of up-front design, in which the functional requirements need to be reviewed thoroughly, and then be composed from domain abstractions.

The rest of this section is based on the architecture design of ALA Datalink. First, we explored the requirements of Datalink in Section 4.1.1, both functional and non-functional. Second, Section 4.1.2 details the process of designing Datalink architecture, and decomposing the identified requirements with ALA. Finally, Section 4.1.3 explains the way we documented the architecture design of ALA Datalink.

4.1.1 Requirements of Datalink

Software requirements can be classified as functional and non-functional (Loucopoulos & Karakostas, 1995). Functional requirements includes the features, functionalities that define the behavior of a system (Pohl, 2010), while non-functional requirements refer to the quality attributes of a system (L. Chen, Babar & Nuseibeh, 2012). For ALA Datalink, the requirements also come from the two aspects.

Functional Requirements

Datalink was designed for the customers of Datamars to manage the data of the portable embedded livestock devices through a computer. It is a Windows desktop application that was developed in C#, which allows the livestock devices to connect with it through the USB ports of a computer, and manage the data on the connected devices.

Figure 4.1 illustrates the home page of the legacy Datalink, which consists of

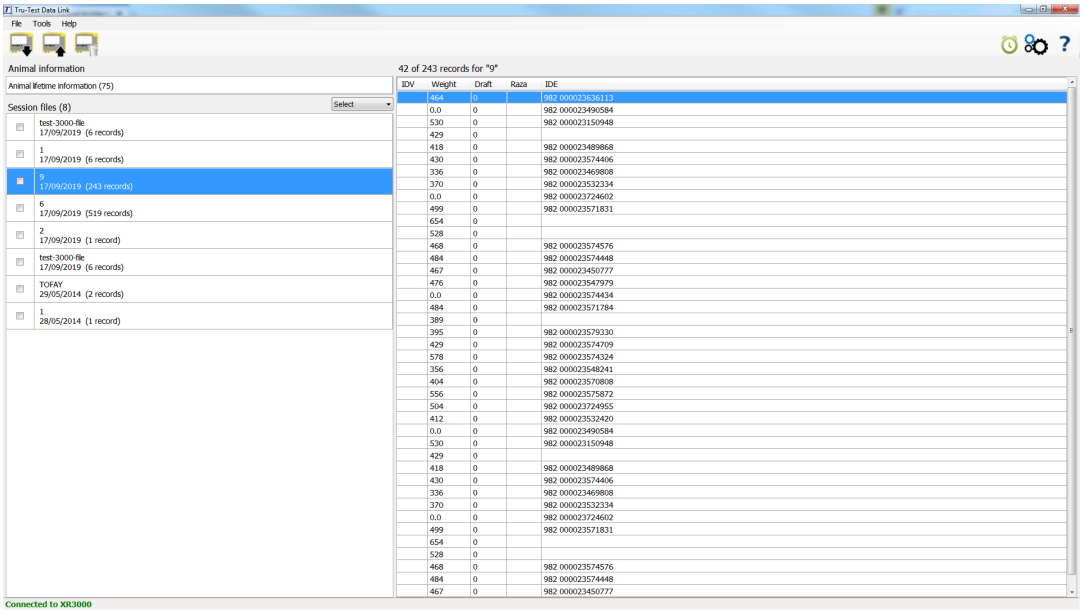


Figure 4.1: The Home Page of Legacy Datalink

several function areas e.g. the menu bar "`File`", "`Tools`" and "`Help`" at the top, the operational buttons at the down side of the menu bar, and the data displaying section that occupies most of the page in the middle and bottom. The data in the grids comes from a connected "`XR3000`" device which is marked as "Connected to `XR3000`" at the bottom left side, and the data is transported through USB cable from the device to the application.

Since we aim to re-develop the existing application, the requirements of Datalink were determined to be the same as the legacy application. We summarized the main requirements that we sought to accomplish in this research, shown as follows:

1. It is able to connect to all livestock management devices of Datamars e.g. "`XR3000`", "`XRS`", "`XRS2`" through USB ports to interact with the connected devices i.e. sending messages, receiving messages.

2. The data in any connected devices can be downloaded and displayed in the corresponding presentation areas. In Datalink, the data of a device is classified

as session files and session data. The former includes the basic information of session files e.g. name, date, which is presented in the grid at the left side in Figure 4.1, while the latter is the content of each session file which consists of the concrete animal information such as "`EID`", "`Weight`", and it is displayed in the grid at the right side of Figure 4.1.

3. The session data in any connected device can be downloaded and saved to `csv` files in the computer with a given format.

4. Any formatted session `csv` files can be imported to the application and uploaded to a connected device through the application.

5. The session data can be uploaded to the cloud services e.g. `NAIT` (National Animal Identification and Tracing is a system of agricultural animal tracing in New Zealand for bio-security and human health).

Non-functional Requirements

The implementation of non-functional requirements lies in the architecture, which significantly impacts the quality attributes of a system (L. Chen et al., 2012). For the re-developed Datalink, the architecture we adopted is determined with ALA, and this architecture was supposed to improve the *maintainability* of the code base. However, there exists other quality attributes need to be considered:

- *Interoperability*. According to ISO 25023 (2016), interoperability reflects the degree of information exchange and successful use of the exchange between more than two components, systems, and products. Datalink needs to fetch data from the devices or upload data to the devices, thus the interaction between the application and the devices needs to be considered through interoperability.

- *Usability*. ISO 25023 (2016) explains usability as the degree of the system or product can be used by particular users for achieving specific intentions under specific context, and the process is considerably effective, efficient and satisfied. In Datalink, the usability is an indispensable attribute because it is developed for the ordinary users, and it directly impacts the value of the software for customers.

4.1.2 The Architecture Design Process

Spray and Sinha (2018) state that the designer of an ALA application needs the skills of a software architect who should possess the knowledge of the design principles presented in their article. In this case, the design of ALA Datalink was under the help of the software architect of Datamars, and the maintainer of the legacy Datalink. The former provided suggestions and decisions on the design, while the later afforded accurate information on the details of the requirements.

The architecture design of Datalink with ALA is the process that decomposes the previously identified functional requirements into ALA elements i.e. *domain abstractions, programming paradigms and requirements expression*, which constitute the three programmable layers of ALA.

A General Process of ALA Architecture Design

The architecture design of ALA Datalink is not like building a mansion where each level (layer) is constructed from bottom to top after the lower level is accomplished. According to Spray and Sinha (2018), ALA design involves one requirement and design for it at a time, until all the requirements are included and designed.

Therefore, the whole architecture design is an iterative process, as illustrated in Figure 4.2. We started with picking up one requirement and decomposing it into programming paradigms and domain abstractions. We specified the requirements by

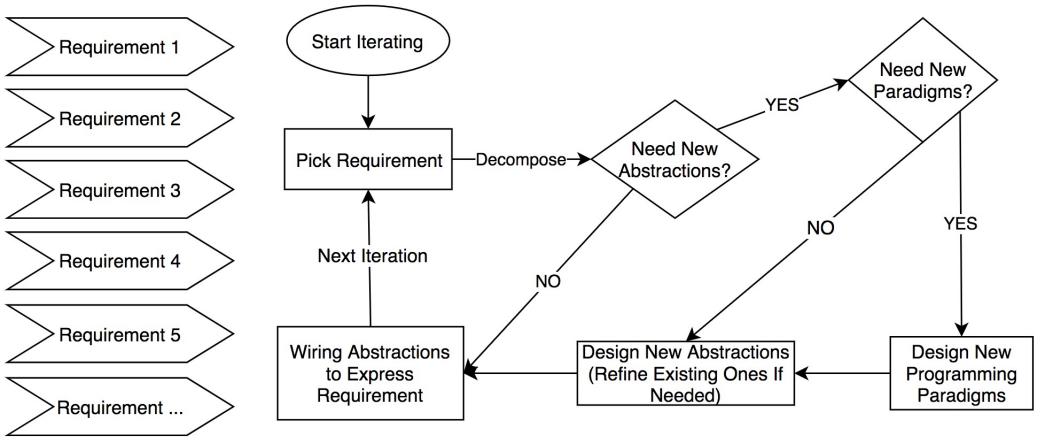


Figure 4.2: ALA Datalink Architecture Design Process

increasing the depth and breadth, the programming paradigms and domain abstractions were confirmed by the ongoing iterations.

Principles of Designing Domain Abstractions

Visser (2007) states that abstraction is the only way to achieve knowledge separation ultimately. Domain abstractions in ALA aims to get rid of knowledge from the specific requirements, and providing general and reusable functionalities to build specific applications under the way they are wired. Based on the experience of the software architect of Datamars, as well as the actual design activities we carried out, some suggestions are given to design high-quality domain abstractions:

1. Spray and Sinha (2018) state that any object-oriented system that is developed in any language involves certain degree of data flow, control, and user interactions considerably. Such commonalities bring us some frequently used abstractions, for example, a data grid which displays two-dimensional data with a giving data source, and a sort which orders the data in a data structure with an assigned sequence.

2. A good abstraction does not possess any knowledge of the application. Specifically, such abstractions usually have general functionalities, and it is hard to know the way it correlates with the application until it is instantiated, configured and wired.

3. A good abstraction usually has fewer input and output ports. Input ports of an abstraction represent the interfaces it implements, whereas output ports mean the interface fields it declares inside its class body. More input or output ports increases the particularity and complexity of wiring, correspondingly, decrease the extent of abstract.

The maintainability of ALA depends significantly on the quality of the domain abstractions. Good domain abstractions reduce the maintenance efforts to a great extent, where the potential maintenance intentions can usually be satisfied by wiring existing abstractions. Otherwise, the design and implementation of new abstractions would considerably increase the maintenance efforts. Thus, it is indispensable to follow the principles to design high-quality domain abstractions.

The Deliverable

The architecture design resulted nine programming paradigms and 47 domain abstractions, as illustrated in Table 4.1 and 4.2 (refer Appendix A to see the whole list and detailed explanations of the functionalities and design notions of those elements). The requirements expressions that located at the application layer is the way to wire the suitable abstractions to satisfy the requirements, which is illustrated in Section 4.1.3.

The domain abstractions in Table 4.2 are able to represent all the requirements of Datalink under different wiring compositions. Before these abstractions being instantiated and wired, it is hard to relate them with the requirements, because they are good abstractions that only provide general functionalities and are less specific to

Type	Paradigms
UI	IUI, IUIWizard
Event	IEvent, IEventB
Data Flow	IDataFlow, IDataFlowB, ITableDataFlow
Connector	EventConnector, DataFlowConnector

Table 4.1: Programming Paradigms of Datalink

Type	Abstractions
UI	Button, Grid, MainWindow, Menu, MenuItem, Menubar, Horizontal, OpenFileBrowser, OpenWindowsExplorer, OptionBox, OptionBoxItem, Panel, Picture, PopupWindow, ProgressBar, RightJustify, RowButton, SaveFileBrowser, StatusBar, Text, ToolBar, Tool, Vertical, Wizard, WizardItem
Data Processing	ConvertTableToDataFlow, ConvertToEvent, Count, Equals, Filter, Iterator, LiteralString, Map, Not, Select, Sort, StringFormat, Transact, Value
Gate	DataFlowGate, EventGate
SCP and I/O	LifeDataSCP, SCPDeviceSense, SCPProtocol, SessionDataSCP, SessionListSCP, CSVFileReaderWriter

Table 4.2: Domain Abstractions of Datalink

the application. Apart from the `SCP` and `I/O` abstractions which were specifically designed for the interaction between the devices and Datalink, the other abstractions can be commonly used in many different projects.

4.1.3 Architectural Documentation

According to Kruchten (1995), software architecture documentation involves the 4+1 views i.e. logical view, development view, process view and physical view. In ALA, Spray and Sinha (2018) state that the logical view plays the most important role as it impacts the maintainability directly. The architecture documentation of ALA Datalink was completely based on the logical view, but elaborates the design notions of the *domain abstractions and programming paradigms*, as well as the *requirements expressions*, which makes it possible to effectively guide the implementation.

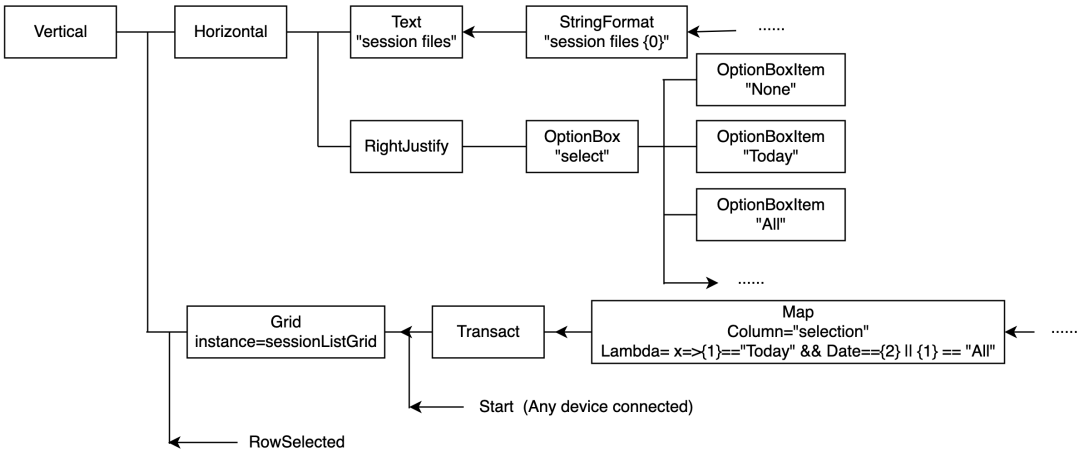


Figure 4.3: A Partial Requirements Expression of Datalink

Documenting the Design of ALA Datalink

Since we already have the domain abstractions designed in the previous section, we can wire them to satisfy the requirements of Datalink. Figure 4.3 partially illustrates the requirements expression of the data grid at the left side in Figure 4.1. Generally, the lines represent the wiring relations between the abstractions, while the arrow lines demonstrate that they are events or data flows go that direction, and whether it is event or data flow depends on the input port of the directed abstraction. For example, there exists a "Start" event for *Transact*, which means when the event triggers, the *Transact* starts to work and feeds data to the *Grid*.

Figure 4.4 partially illustrates the architecture documentation of ALA Datalink. The three rec-tangled sections represent the domain abstractions, programming paradigms and application layer respectively. The domain abstractions and programming paradigms were put in a list to explain the rationale of the design and functionalities (refer Appendix A to see the whole list of the detailed design of domain abstractions and programming paradigms), while the requirements expression describes the way the domain abstractions are wired to satisfy the requirements.

Besides, the software we used to document the architecture design is XMind. We

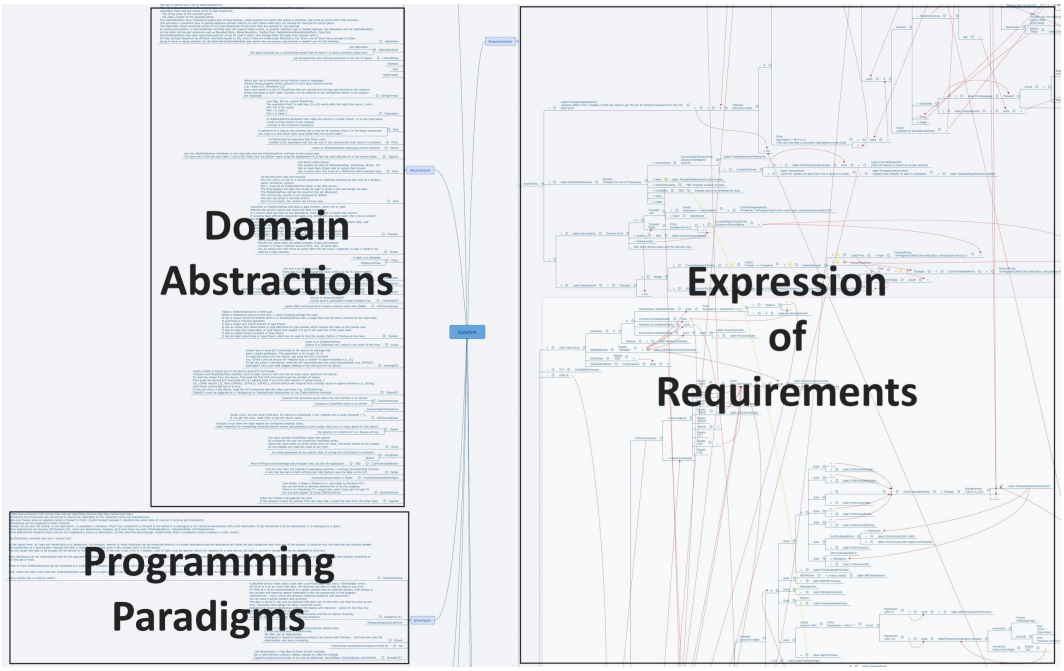


Figure 4.4: Architecture Documentation of Datalink

selected XMind for the following reasons:

1. XMind turns out to be a fine way to draw ALA design because it keeps itself laid out, leaving us free to represent requirements and invent abstractions as we go.

2. The red cross lines are also very quick to do, and can be tidied up relatively and easily to wire the instances of abstractions that are reused for multiple times in different requirements expressions.

A complete architecture design documentation of ALA Datalink can be download from here: `https://github.com/cdxybf/ALA_Datalink/blob/master/ALA_DataLink/Application/Datalink.xmind`.

Quality of ALA Datalink Documentation

A well-documented architecture plays an indispensable role in software development which helps to promote the effectiveness and efficiency of the implementation and

maintenance. According to Clements et al., (2002), a sound architectural documentation should cover seven fundamental principles, as shown in Table 4.3. We systematically compared the ALA architecture documentation with the principles, concluded that the architectural documentation of ALA Datalink is sound.

Principles	ALA Documentation
Writing from the perspective of the reader	The potential readers are developers, as it guides the development and maintenance. The documentation was also written from the perspective of implementation (Section 4.1.3).
Avoid unnecessary repetition	There exists only one documentation, and each section has clear and independent responsibility (Figure 4.4).
Avoid ambiguity	Programming paradigms and domain abstraction are easy to understand as they are small pieces, while requirements are straightforward and executable (Figure 4.4 and Appendix A).
Use a standard organization	It is easy to define and manage the organization style in one documentation. The documentation was developed by XMind, which unites the standard (Section 4.4 and downloading the whole design documentation from the link given above).
Record rationale	The technical elements are briefly but precisely explained, to the extent that it is able to be implemented (Appendix A or downloading the whole documentation).
Keep documentation current but not too current	Always keep updating before the development as well as maintenance (Section 4.3).
Review documentation for fitness of purpose	Bidirectional updates are required between it and the code base, so it is reviewed during the implementation and maintenance (Section 4.3).

Table 4.3: Architecture Documentation Quality of ALA Datalink

4.2 Implementation of ALA Datalink

Based on the architecture design carried out previously, in this section, we followed the Scrum (Schwaber, 1997) method to systematically implement Datalink with ALA. Section 4.2.1 explains the background of ALA's key mechanisms which are required for the implementation. Section 4.2.2 describes the process we followed Scrum method to carry out the implementation. Section 4.2.3 presents the code related affairs and some examples are provided. Section 4.2.4 discusses the completeness of the re-developed Datalink.

4.2.1 Background of ALA Mechanisms for Implementation

The background mechanisms for an ALA application development is necessary, because ALA is an reference architecture, and it might be difficult to start the implementation before possessing the required fundamental knowledge of the way it works. Therefore, this section discusses two mechanisms that need to be learnt before implementation, which mainly comes from the technical perspective.

The Way Wiring Method Works

Wiring is an important concept at the application layer of ALA. As aforementioned, the main effect of ALA's application layer is wiring the domain abstraction to satisfy the requirements. However, it is not possible to wire any two selected abstractions. According to Spray and Sinha (2018), only those abstractions that conforms to a same programming paradigm can be wired.

Figure 4.5 illustrates the way wiring works in the application layer of ALA. Basically, the wiring intends to connect any two compatible abstractions to express a specific requirements. The compatible abstractions in the graph is abstraction B and C, where B has a field that declares with the type programming paradigm (interface) A, while C

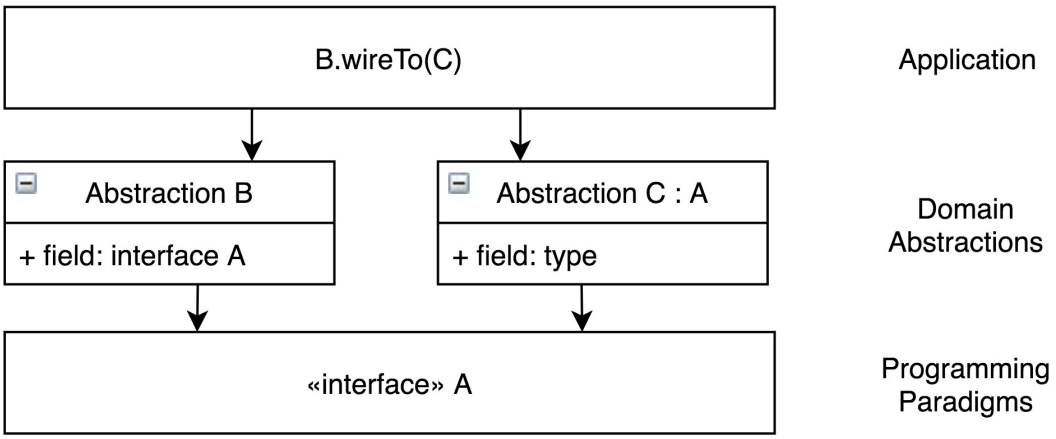


Figure 4.5: Wiring Mechanism of ALA

implements A. Such compatibility makes B able to wire to C at run-time to assign an instance of C to the field of B's instance.

Besides, in the actual development, some additional explanation needs to be clarified to help comprehending wiring:

1. The implementation of wiring method utilizes advanced characteristic of a language i.e.reflection to implicitly assign the instance of the wired abstraction to the declared field of an wiring abstraction.

2. Wiring allows multiple abstractions wire to a single one, but a declared field in the wiring abstraction can only be assigned once with a compatible instance value.

3. ALA allows an abstraction to conform to multiple programming paradigms. Thus, an abstraction can wire to the same abstraction on its different declared fields, based on how many fields overlaps with the programming paradigm types that the wired abstraction implemented.

Two Important Design Patterns

Design pattern in software engineering aims to solve common problems that could happen in different contexts (Pree & Gamma, 1995). In ALA, the two design patterns that need to be learnt are the *Composite* and *Decorator*. Both of the two design patterns are categorised as structural pattern, and they are concerned with how classes and objects are composed to form larger structures (Gamma, 1995). The way the two abstraction works in ALA is briefly explained in Table 4.4.

Pattern	Definition	Rationale in ALA
Composite	Composite pattern lets clients treat individual objects and compositions uniformly (Gamma, 1995).	ALA uses wiring to put all related elements in a tree structure to represent the application hierarchy, so the system can be driven by events and data flows once it starts.
Decorator	It allows behavior to be added to an individual object dynamically without affecting the behavior of other objects from the same class (Gamma, 1995).	Reusable abstraction in ALA needs to have single-responsibility. Such design pattern allows us to keep the simplicity of the abstractions while achieve the complexity of the application.

Table 4.4: The Design Patterns Used in ALA

With respect to *Composite* pattern, it is the fundamental mechanism of driving an ALA application. In the implementation of an abstraction, the fields are supposed to wire instances of other abstractions by declaring programming paradigm (interface) types, which means the abstraction does not know which abstractions would be wired at run-time. However, it treats any type of instances uniformly through programming paradigm (interface) types to eliminate all the coupling between abstractions, merely adding dependencies on the programming paradigm layer.

In terms of the *Decorator* pattern, the decorated object is carried out through programming paradigms (interfaces). The decorating activity happens at run-time when an instance is wired to a declared field. The class possesses that field would

dynamically add responsibility to that instance without changing the simplicity of the wired abstraction. Decorator is mainly used in data processing abstractions such as *Sort*, *Select* and *Filter* (refer Appendix A to see the details of the abstractions).

4.2.2 Driving the Implementation with Scrum

Since we followed the Scrum (Schwaber, 1997) method to carry out the implementation of ALA Datalink, the standard steps of Scrum were considered in the process. However, as only one developer participated in the re-development, we did not carry out the activities correlated with team building and collaborations. Therefore, we merely took some necessary steps in the sprints i.e. *Sprint Plan*, *Daily Stand-up Meeting* and *Sprint Review*.

Sprint Plan

The sprint plan is usually a meeting that determines the tasks of the next sprint with all the people of a Scrum team participating in (Schwaber, 1997). In this case, although only one developer was considered, we still planned the sprints to simulate what a development team does in a real company.

Sprint planning was mainly carried out based on the architecture design documentation, which were supposed to be the backlog of the sprints. Objectives of the implementation is not only exploring the method of ALA development, but also creating a code base for maintainability evaluation. Apart from that, we have to avoid repeated and time consuming tasks at the beginning due to time limitations. Thus, three measures were adopted in the planning to generate such code base:

1. Constructing the main UI abstractions first at the early stage as they provide straightforward evidence on the progress and are also container for displaying data;

2. We implemented the requirements horizontally from left to right based on the design documentation, as both the depth and breadth of the requirements grow in that direction.

3. Finishing the home page of Datalink first because it creates a place for all the functionalities. Moreover, the home page presents the primary data flows of Datalink, which is one of the main functions of the application.

Paradigms	Abstractions	Requirements
IUIWizard, IUI	MainWindow, Menubar, Menu, MenuItem, Wizard, WizardItem, Panel, StatusBar, Vertical, Text	Building the main window, menus, wizard and organizational UI abstractions
IDataFlow, IEvent	OptionBox, OptionBoxItem, RightJustify, RowButton, Horizontal, SCPProtocol	UI abstractions, seeks to find the way of downloading data from devices
IRequestResponseDataflow	SCPProtocol, Transact, SessionListSCP, SCPDeviceSense, ConvertTableToDataFlow	Connect a plugin device, fetching and handling session list data
ITableDataFlow	SessionListSCP, Grid, SessionDataSCP, Map, Select, Sort, StringFormat, ConvertToEvent	Handle session list data; Downloading session data from device
N/A	SessionDataSCP, LifeDataSCP, Count, Equals, LiteralString, Not	Session data processing and transacting for displaying in grid
IDataFlowB, IEventB	ProgressWindow, ProgressBar, Button, Filter, Tool, Toolbar, Value, DataFlowGate	Downloading progress of session list data; Gates for blocking and triggering operations
ITableDataFlow	Transact, SessionDataSCP, Iterator, OpenFileBrowser, OpenWindowsExplorer, EventGate, CSVFileReaderWriter	Downloading session/life data iteratively and save to csv file
N/A	CSVFileReaderWriter, SaveFileBrowser, SessionListSCP, SessionDataSCP	Exporting session data from multiple csv files to device

Table 4.5: The Sprints of Datalink Implementation

Table 4.5 illustrates the eight sprints that we carried out in the implementation process, based on the measures mentioned above. One thing that should be notified

is that the eight sprints were not planned completely at the beginning. Instead, every sprint was planned after the previous one was finished. Some of the paradigms and abstractions in later sprints might overlap those in the previous ones, which means we optimized those elements in later sprints as they might not conform to the original design and are not reusable as expected.

Daily Stand-up Meeting and Sprint Review

According to Schwaber (1997), both the stand-up and sprint review require the whole development team participating in the meeting. Although there was only one developer for the re-developed ALA Datalink, the two meetings were still convened with the architect of Datamars, but for more specific purposes.

With respect to daily stand-up meeting, it aims to notify the progress of each member in the team, as well as expose the potential issues (Schwaber, 1997). However, in this case, the daily stand-up meeting mainly serves two purposes:

1. It was usually convened at the end or the beginning of a day to explore the issues that encountered in the implementation, or the architect provided suggestions on the development of domain abstractions.

2. The finished abstraction of a day was reviewed by the architect to assure the quality, because the quality of abstractions is of crucial importance at the maintenance stage.

The daily stand-up meeting was convened everyday in the first four sprints, because studying and understanding the domain abstractions is difficult for a beginner of ALA, and deviations between implementation of abstractions and the design arises frequently. With the developer getting better understanding of ALA and the requirements, we did not have stand-up meetings at the last four sprints.

In terms of the sprint review meeting, it was convened at the end of every sprint. Schwaber (1997) states that such meeting usually involves the product owner who can review the functional quality of the application. In this case, the product quality was reviewed by comparing the re-developed application with the legacy one, as well as the requirements identified in Section 4.1.1. Apart from that, we summed up the problems in the implementation and tried to resolve them in a general way. Such as the principles of designing abstraction (Section 4.1) and ALA mechanisms required before implementation (Section 4.2.1).

4.2.3 Implementation and Examples

This section details the implementation of ALA Datalink from the technical perspective. Moreover, we specifically explained the composite pattern utilization in ALA. To give an example, we present the implementation of programming paradigms, domain abstractions and requirements in the first sprint.

The first sprint aims to create the structure of the home presentation page of Datalink, thus most of the programming paradigms and domain abstractions are correlated with UI. As the dependencies in ALA happens from top to down based on its layers, we firstly created the *IUI* interface (programming paradigm), which allows us to establish the domain abstractions depending on it, and then constructed the UI structure through the method in the interface uniformly.

Building Programming Paradigms

There exist two programming paradigms in the first sprint i.e. *IUI* and *IUIWizard.* However, they were not implemented simultaneously. The *IUIWizard* was created when we realized that the IUI cannot satisfy the way *Wizard* and *WizardItem* wire to each other, as they need more interactions rather than UI, then we used *IUIWizard* specifically

```
1  using System.Windows;
2
3  namespace ProgrammingParadigms
4  {
5      /// <summary>
6      /// Hierarchical structure of the UI
7      /// </summary>
8      public interface IUI
9      {
10         UIElement GetWPFElement();
11     }
12 }
```

```
2  namespace ProgrammingParadigms
3  {
4      /// <summary>
5      /// An extended interface for Wizard and WizardItem.
6      /// </summary>
7      public interface IUIWizard : IUI
8      {
9          bool Checked { get; }
10
11         void GenerateOutputEvent();
12     }
13 }
```

Figure 4.6: The IUI and IUIWizard Programming Paradigm

for the two abstractions.

Nevertheless, no matter if it is *IUI* or *IUIWizard*, the principles and methods of implementation are similar. The objective of the paradigms is creating an united interface so that the abstractions implement them would be wirable. Figure 4.6 illustrates the implementation of the two interfaces. *IUIWizard* extends *IUI* and it is more specific for the *Wizard* and *WizardItem* abstractions.

Building Domain Abstractions

Most of the designed domain abstractions were accomplished when we created the IUI interface, because those abstractions only depend on *IUI*. The abstractions implement the *IUI* interface and they return the `WPF` element when the method is called. This approach unites the interactions of the UI abstractions through *IUI*, which provides consolidated ports when the abstractions are wired to express requirements.

Figure 4.7 illustrates the way we implemented the *Menubar* abstraction. The Line 11 indicates that this abstraction implements *IUI* interface, so it is an *IUI* abstraction that can be wired to other abstractions which declares *IUI* as a field. The 14 line shows that it can wire to multiple *IUI* abstractions, as a Menubar usually contains different menus. In the implementation method at line 25 to 29, we can specify that all the abstractions wired to it would be accessed through the *IUI* interface method and add

```
using ProgrammingParadigms;
using System.Collections.Generic;
using System.Windows;

namespace DomainAbstractions
{
    /// <summary>
    /// Menubar as found on most applications. Displays Menu's horizontally.
    /// Implements IUI. Has a list of Menus which are IUI.
    /// </summary>
    public class Menubar : IUI
    {
        // outputs
        private List<IUI> children = new List<IUI>();

        // private fields
        private System.Windows.Controls.Menu menuBar = new System.Windows.Controls.Menu();

        /// <summary>
        /// An IUI element which is a windows-desktop application style MenuBar.
        /// </summary>
        public Menubar() { }

        // IUI implementation ------------------------------------------------
        UIElement IUI.GetWPFElement()
        {
            foreach (var c in children) menuBar.Items.Add(c.GetWPFElement());
            return menuBar;
        }
    }
}
```

Figure 4.7: The Implementation of Menubar Domain Abstraction

the WPF element to it, so it would display all the UI elements wired to it.

This graph also demonstrates that the domain abstractions are not specific to the application since they do not know where they would be wired to and what abstractions they would wire until they are instantiated and wired in the application layer.

Requirements Expression

The requirements expression happens at the application layer, where the domain abstractions are wired in a way intended to satisfy the requirements. Figure 4.8 depicts the way we wired the abstractions implemented in sprint 1 to build the home presentation

```
mainWindow
// UI
.WireTo(new Vertical() { Layouts = new int[] { 0, 0, 2, 0 } }
    .WireTo(new Menubar()
        .WireTo(new Menu("File")
            .WireTo(new MenuItem("Get information off device", false) { IConName = "3000Import.png" }
                .WireFrom(XR3000ConnectedConnector)
                .WireTo((getInfoWizard = new Wizard("Get information off device") { SecondTitle = "What information do you want to get off the de
                    .WireTo(new WizardItem("Get selected session files") { ImageName = "Icon_Session.png", Checked = true }
                        .WireTo(new Wizard("Get information off device") { SecondTitle = "What do you want to do with the session files?", ShowBa
                            .WireTo(new WizardItem("Save selected session files as files on the PC") { ImageName = "Icon_Session.png", Checked =
                                .WireTo(saveSessionsDataToFileBrowser)
                            )
                            .WireTo(new WizardItem("Send records to NAIT") { ImageName = "NAIT.png" })
                            .WireTo(new WizardItem("Send sessions to MiHub Livestock") { ImageName = "MiHub40x40_ltblue_cloud.png" })
                            .WireTo(new WizardItem("Send to \"remote system\" using Animal Data Interface") { ImageName = "ttlogo76x32.png" })
                            .WireTo(new WizardItem("Send to LIC") { ImageName = "LicLogo.png" })
                            .WireTo(getInfoWizard)
                        )
                    )
                    .WireTo(new WizardItem("Get all animal lifetime information") { ImageName = "Icon_Animal.png" }
                        .WireTo(new Wizard("Get information off device") { SecondTitle = "What do you want to do with the animal lifetime informa
                            .WireTo(new WizardItem("Save all animal lifetime information as a file on the PC") { ImageName = "Icon_Animal.png", C
                                .WireTo(saveLifeDataToFileBrowser)
                            )
                            .WireTo(new WizardItem("MiHub Life Data") { ImageName = "MiHub40x40_ltblue_cloud.png" })
                            .WireTo(new WizardItem("Send to \"remote system\" using Animal Data Interface") { ImageName = "ttlogo76x32.png" })
                            .WireTo(getInfoWizard)
                        )
                    )
```

Figure 4.8: A Piece Code of Requirements Expression

page of ALA Datalink. The UI structure includes *Menubar* and *Menus*, as well as the popup *Wizard* and its sub-items when the menu items are selected.

Figure 4.9 partially illustrates the way the *Composite* pattern is used in ALA application. The graph is specifically explained for the composition of *IUI* correlated domain abstractions. From the perspective of an overall view, the rationale of any other kind of paradigms and abstractions is similar. With more and more abstractions implemented and wired, the scale of the composition would increase until all the requirements are expressed.

The *Decorator* pattern is not discussed here, because it is more about data processing which occurs in the later sprints. The difference between *Composite* and *Decorator* pattern is that the former usually handles multiple objects to form a large structure, whereas the later intends to decorate one object to add responsibility to that object, which is simpler comparing with the former one.

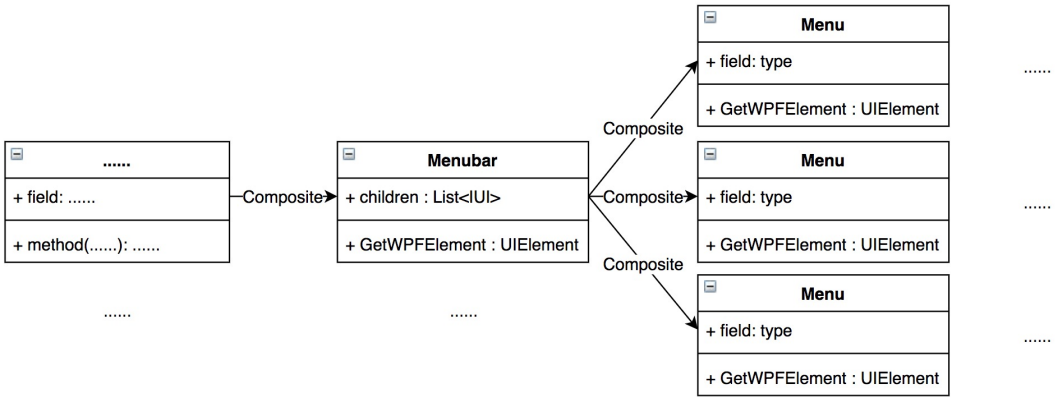


Figure 4.9: A Part of Composite Pattern Utilization

Verification

Verification in ALA Datalink mainly exists in domain abstractions and requirements expression. The former aims to assess the quality of the abstractions while the later intends to check the correctness of the functionalities.

In this case, we carried out unit tests to verify the quality of domain abstractions. According to Spray and Sinha (2018), the domain abstractions are independent assets and it is very easy to build and run unit tests for them. The unit test cases in ALA Datalink were built based on the design notions and functionalities of the abstractions.

Figure 4.10 illustrates the unit test case we built for *Button* abstraction. This abstraction can layout to an assigned shape and title, and generate event when clicked. We verified UI, title and event which are the functionalities of the abstraction to assure its quality.

In terms of functional verification, we simply compare the re-developed Datalink with the legacy one. Moreover, we also verified the functionalities based on the functional requirements identified in Section 4.1.1.

```
9       [TestClass]
10      public class ButtonTestCase
11      {
12          [TestMethod]
13          public void ButtonTest()
14          {
15              var button = new Button("test button")
16              {
17                  InstanceName = "test button",
18                  FontSize = 15,
19                  Height = 30,
20                  Width = 80,
21                  Margin = new Thickness(0, 0, 0, 0)
22              };
23
24              ButtonEvent buttonEvent = new ButtonEvent() { Flag = false };
25              button.WireTo(buttonEvent);
26
27              System.Windows.Controls.Button btn = (System.Windows.Controls.Button)((IUI)button).GetWPFElement();
28
29              // test ui
30              Assert.AreEqual(15, btn.FontSize);
31              Assert.AreEqual(30, btn.Height);
32              Assert.AreEqual(80, btn.Width);
33              Assert.AreEqual(new Thickness(0), btn.Margin);
34
35              // test title and instance name
36              Assert.AreEqual("test button", btn.Content.ToString());
37              Assert.AreEqual("test button", button.InstanceName);
38
39              // test click event
40              btn.RaiseEvent(new RoutedEventArgs(System.Windows.Controls.Primitives.ButtonBase.ClickEvent));
41              Assert.AreEqual(true, buttonEvent.Flag);
42          }
```

Figure 4.10: An Unit Test Case of Button

4.2.4 Deliverable of Implementation

As previously mentioned, we carried out eight sprints to re-develop Datalink with ALA. At the end of the implementation, we accomplished most of the requirements summarized in Section 4.1.1, and the designed 47 domain abstractions were completely implemented. Figure 4.11 depicts the home page of Datalink developed with ALA. Comparing with the graph illustrated in Figure 4.1, we could hardly find difference between them. However, the code bases that generate such application is totally different. ALA Datalink was developed under the rules of ALA, which could be referred from `https://github.com/cdxybf/ALA_Datalink`, and it was constructed on the domain abstractions. In terms of the legacy one, we cannot provide the source code

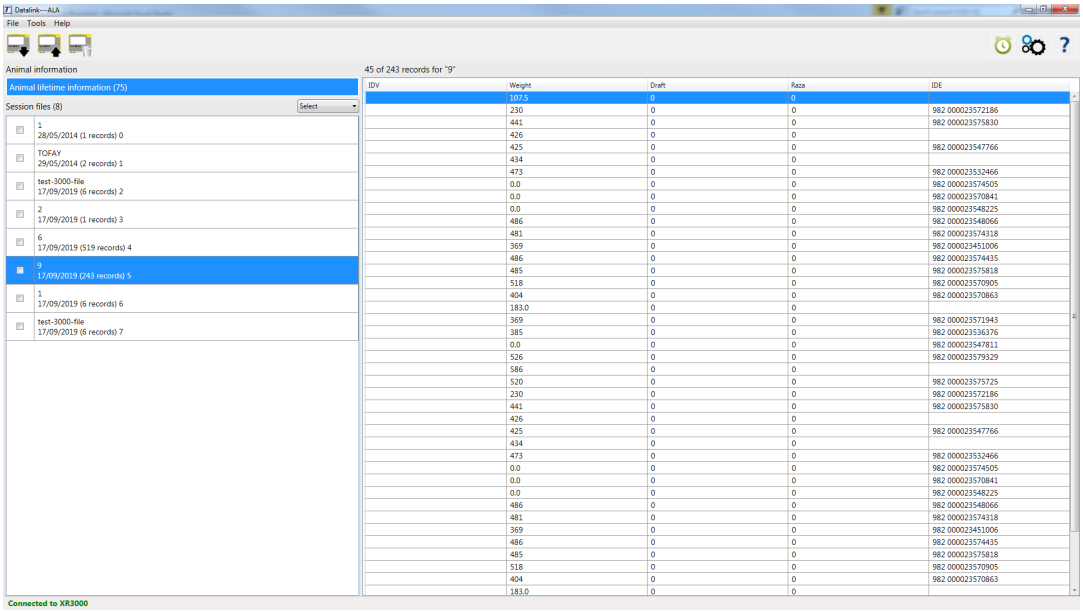


Figure 4.11: ALA Datalink Home Page

due to confidential agreements, but the source lines of code is about ten times of ALA Datalink, and each functionality is specifically designed and implemented, which is different from ALA's notion of abstractions.

Requirements	Finished
It is able to connect to any livestock management devices of Datamars e.g. "XR3000", "XRS", "XRS2" through USB ports to interact with the connected devices i.e. sending messages, receiving messages.	Connect to XR3000 merely
The data in any connected devices can be downloaded and displayed in the corresponding presentation areas.	YES
The session data in any connected device can be downloaded and saved to csv files in the computer with a given format.	YES
Any formatted session csv files can be imported to the application and uploaded to a connected device through the application.	YES
The session data can be uploaded to the cloud services e.g NAIT.	NO

Table 4.6: Completeness of ALA Datalink

Table 4.6 depicts the completeness of ALA Datalink based on the functional requirements we identified in Section 4.1.1. Except the fifth one, the other requirements were

fully completed in ALA Datalink. One thing needs to notify is that the application can only connect to `XR3000` at the end of the re-development, as it is the most representative device of Datamars. Once it is able to connect to `XR3000`, the connection of other devices would be much easier, and the functionalities of other devices in the application are the same as `XR3000`.

4.3 Maintenance Activities of ALA Datalink

Maintenance activities usually happen after the delivery of the software, and the objective is to achieve the availability when environment changes or unexpected error happens. Kaur and Singh (2015) explain that three forms of maintenance are considered:

1. Corrective maintenance is about fixing emergency program and debugging routine;
2. Adaptive maintenance are performed when it is needed to response to technology updates;
3. Perfective maintenance aims to improve documented and requested enhancement.

In the maintenance of ALA Datalink, the first thing is to assure the conformity between the code base and the design documentation. The design documentation is the direct representation of the code base and requirements, and it is straightforwardly comprehensive and intelligible rather than a pure code base for readers which improves the effectiveness of maintenance to a great extent.

4.3.1 Resolve Nonconformity Between Design and Implementation

It is important to resolve the inconformity between design and implementation before any maintenance in ALA. The impact of design documentation demonstrates that any inconformity between it and implementation might negatively mislead the activities

correlated with it in SDLC. To make matters worse, more time might be spent on the analysis of code base to acquire the knowledge of the way it works, which directly reduce the maintainability.

From the experience and practice that we encountered in the process, the nonconformity between design diagram and code base mainly comes from three aspects. Before the actual maintenance, these nonconformity should be corrected:

1. The implementation of programming paradigms and domain abstractions does not completely achieve the functionalities designed, or deviates from the original design. This happens when the programming paradigms or domain abstraction were not elaborately implemented. For example, a "Grid" abstraction which is usually used as the destination of a "Transact" for displaying data, but sometimes it is also used as the source of a "Transact", so the "Grid" should be implemented both source and destination functions.

2. The architectural design has not been considered in a proper way, so that it generates some inappropriate abstractions, or produces ambiguity. It is not always effective of the design, because some abstraction might not be abstract enough, or over-designed. For example, the abstraction "Value" which was proved to be useless in our later maintenance, we can use "DataFlow" instead.

3. The requirements expression (wiring) does not completely conform to that in the design diagram. This is especially important, because the wiring in documentation is the only guidance to follow for composing the requirements in ALA's application layer. If it was not completely and correctly followed, more efforts would be spent in later maintenance activities to achieve the correctness and effectiveness.

4.3.2 Perfective Maintenance

In ALA Datalink, regular perfective maintenance is carried out to continuously satisfy the evolutionary marketing and ongoing customer requirements. For such kind of maintenance, the process is similar to the way we re-developed Datalink with ALA, which has been discussed previously in the architecture design (Section 4.1.2) and implementation (Section 4.2).

Apart from that, we also need to pay attention on the design documentation and the potential modifications on paradigms and abstractions:

1. The design diagram describes the overall UI and data flow explicitly that the location of the potential changes is discoverable with less efforts for specific requirements. Thus, maintenance is easier to carry out by following the design diagram rather than the code base.

2. Requirements implementation by making modifications on existing domain abstractions is not advocated in ALA, neither on programming paradigms. Because modifications on existing abstractions requires re-testing all the correlated instances and wired requirements, which brings more uncertainty and effort for the maintenance activity.

4.3.3 Corrective and Adaptive Maintenance

Corrective maintenance aims to fix the errors and emergencies in a software product. Such kind of maintenance mainly operates the application layer and domain abstraction layer:

1. Some mistakes occur when unexpected wiring exists. The way to fix this is to check the corresponding area of the design documentation, re-wiring the abstractions and change the wiring in the code.

2. If there existed errors at the internal side of domain abstractions, the maintenance needs to inspect the code of those abstraction. However, implementation of abstractions is not described in the design documentation as ALA does not put concerns on implementation, thus such maintenance would not result any modification in design diagram.

In terms of adaptive maintenance, the string-head of it exists at the language layer of ALA's architecture, which directly or indirectly influences the other three layers that depending on it. However, those effects are usually not described in the design diagram as we emphasize little about layers in design. Therefore, the way to carry out adaptive maintenance is to inspect the code without changing the design documentation.

For example, if .NET framework updates from a lower version to a higher version, there might exist some updates to do at the domain abstraction layer. However, the functionalities and the fundamental design of the application and abstractions will not change. Thus, what we need to do is adjust the updates at the .NET framework (language) layer to make the upper three layers work as usual, while the design documentation will not change.

4.4 A General Way to Develop ALA Applications

We reported the process that we carried out to develop Datalink with ALA previously, which conforms highly to a Waterfall model (Balaji & Murugaiyan, 2012). However, when we specify each step in the process, the measures we took is more close to an agile model (Martin, 2002).

In this section, we discuss the generalization of ALA from two aspects. First, the Waterfall model is used to outline the stages need to take in the development. Second, the agile model is referred to iteratively carry out the concrete activities in each stage.

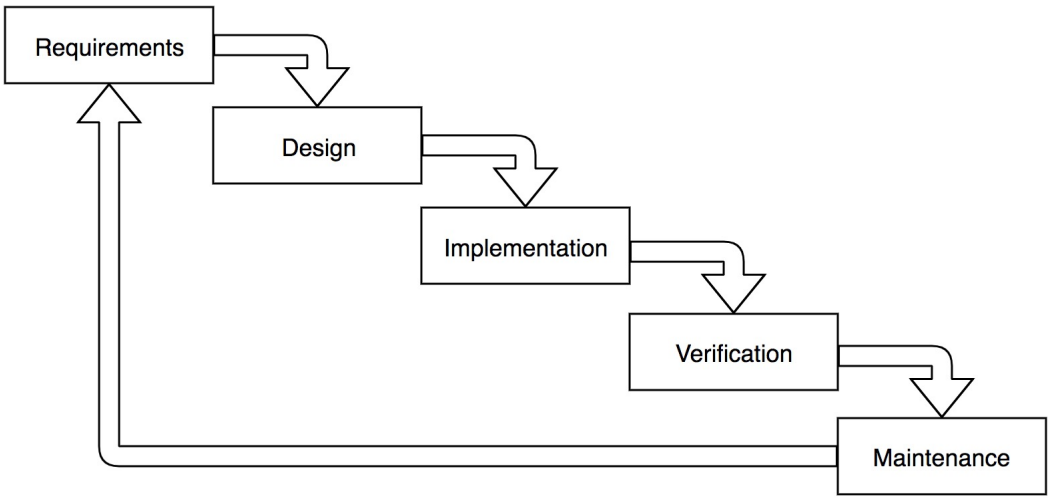


Figure 4.12: A General Process of ALA Application Development

4.4.1 A Waterfall Model to Outline the Process

Generally, a Waterfall model includes requisite steps i.e. requirements analysis, architecture design, implementation, verification and maintenance (Balaji & Murugaiyan, 2012), as illustrated in Figure 4.12.

Requirements Analysis

Requirements analysis is not required in ALA, because modern commercial software usually needs fast changes to respond to the market. Thus, the requirements in ALA are merely the input for the next stage.

Architecture Design

In an ALA application, the architecture does not need to be designed, because ALA has its own four layers architecture and such architecture should keep consistent at any time in ALA. The design here refers more to the design of *programming paradigms and domain abstractions* (Section 4.1), which directly impacts the maintainability in later phases.

Implementation and Verification

The implementation and verification in ALA is similar to general software development. The implementation aims to transfer the design and requirements into an executable code base, while the verification intends to verify the quality of the software from functional and non-functional aspects (Section 4.2.3).

Maintenance

Maintenance in ALA should be particularly highlighted, as ALA was designed to improve maintainability for code bases. However, the ALA maintenance is different from that in a Waterfall model, because ALA always require the maintainer to design programming paradigms and domain abstractions for any perfective maintenance (Section 4.3), and this is why there exists an additional connection between maintenance and requirements in Figure 4.12, which do not usually exist in a general Waterfall model.

4.4.2 An Agile Model to Carry Out the Process

The Agile software development method aims to create a prototype, and add frequent increments on that prototype to continuously satisfy the customer requirements (Martin, 2002). In ALA, the artefacts being iteratively added are not only the requirements, but also the design, implementation and verification, which refers to each step in the Waterfall model.

Iterative Design of Programming Paradigms and Domain Abstractions

The design activity in ALA application development is the process that transfer the requirements to reusable domain abstractions and programming paradigms. However, this activity is different from the general software requirements decomposition, because

the extent of abstract and reusability of the abstractions need to be considered when carrying out the design.

Generally, the programming paradigms and domain abstractions are designed from both non-technical and technical aspects:

1. The non-technical aspect mainly comes from the process of the design (Section 4.1.2), where we described the iterative steps.

2. The technical aspect aims to help designing more reliable and high-quality domain abstractions (Section 4.1.2), which is supposed to have significant influence on the maintainability of ALA code bases.

Incremental Releases of Domain Abstractions and Requirements

Before the actual implementation, we suggest it is better to study the correlated knowledge of ALA (Section 4.2.1). The knowledge helps designers and developers to understand the mechanisms of ALA and design abstractions.

The domain abstractions and requirements are not completed separately in the actual implementation. Instead, it is also an iterative process that goes in the depth-first direction, which means after we finished an abstraction, we wired it in the application layer to make it satisfy some requirements, verify it and implement the next abstraction.

Figure 4.13 illustrates the iterative process of implementing the abstractions and requirements. The implementation is accompanied by verification and each requirement is complete through the order from domain abstractions, requirements to verification. Such iterations assure the quality of both the abstractions and the implementation of requirements.

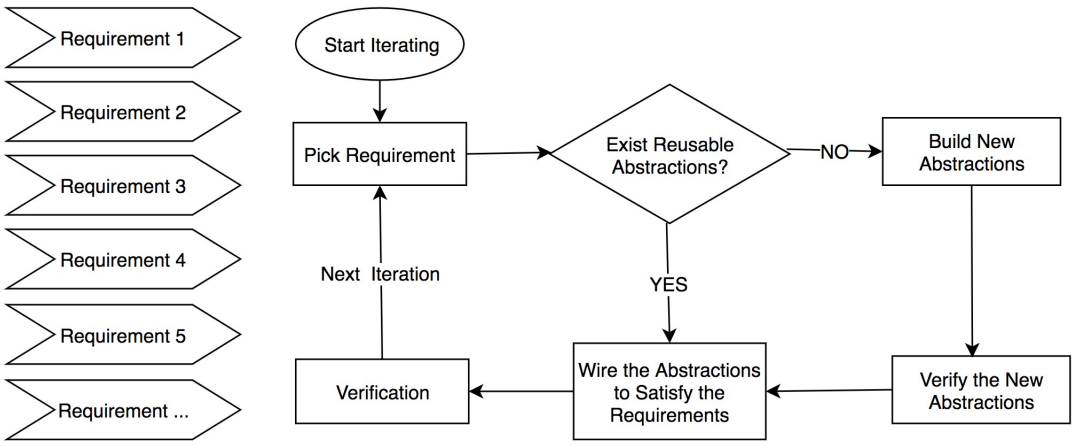


Figure 4.13: A General Process of ALA Implementation

Chapter 5

Maintainability Evaluation Based on ISO Sub-Characteristics

ISO 25010 quality model (2011) decomposes *Maintainability* as *Modularity, Reusability, Analysability, Modifiability and Testability*. In this chapter, we evaluated the sub-characteristics of ALA and legacy Datalink code bases with metrics refined in Section 2.3.

Generally, the evaluation was based on the comparisons of the metrics results of the two code bases, which leads to the organization of the rest of this chapter. First, with the help of the `NDepend`, we give a structural overview of the two code bases to reflect how they are different from each other at a design-level sight (Section 5.1). Second, based on the ISO maintainability model, we carried out the comparisons of metrics results of the five sub-characteristics. We also connected the result of the comparisons to the preliminary assessments in Section 2.3. Finally, we summarized the comparisons and correlations and give a conclusion to this chapter. The result is *Modularity, Reusability, Analysability and Testability* of ALA Datalink are higher whereas *Modifiability* is lower than the legacy application. Our investigation demonstrates that ALA could bring long-term *Maintainability* on a code base (Section 5.7).

5.1 NDepend Dependency Graph - Overall Views of the Two Code Bases

NDepend (Smacchia, 2007) allows us to have overall views of the two code bases about their components/classes, interfaces and the relations between them, which is defined as dependency graph in NDepend. In this section, we compared the dependency graphs of the two code bases, which explicitly demonstrates the differences of the structures. Particularly, we highlighted the classes in the domain abstraction layer of ALA Datalink, which clearly reflects the "zero coupling" feature of ALA.

5.1.1 The NDepend Dependency Graphs

The NDepend dependency graph is similar with a class diagram (Rumbaugh et al., 1991), but emphasizes more on the external connections with other elements rather than the internal methods and properties. It puts all the elements in a hierarchical structure and connects them with directed lines. The connections represent all the possible dependencies of the class diagram i.e. abstraction, realization, association and aggregation but is not specified to a specific type in the graph.

As the dependency graph is very large, we can merely provide clipped ones here. The complete graph can be downloaded from https://github.com/cdxybf/ALA_Datalink/tree/master/Dependency%20graphs.

ALA Datalink Dependency Graph

Figure 5.1 depicts a part of the dependency graph of the re-developed ALA Datalink. From the direction left to right, the elements can be layered correspondingly as the three programmable layers of ALA i.e. application, domain abstractions and programming paradigms. The application layer merely involves one class, which expresses all the

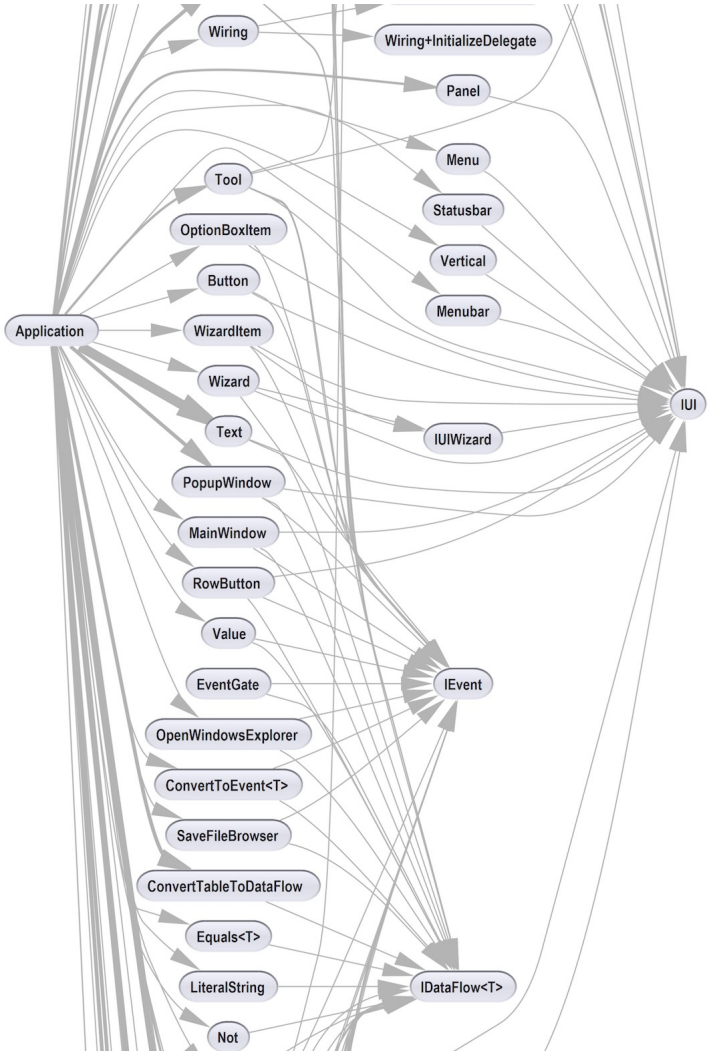


Figure 5.1: A Partial Dependency Graph of ALA Datalink

requirements by wiring the instances of the classes at the domain abstraction layer in the middle of the graph. While programming paradigms which locate to the right side e.g. IUI, IEvent defines the standard followed by the domain abstractions, makes them able to describe their own functionalities and become interactive.

Overall, the ALA dependency graph can be regarded as well-organized, properly ordered and straightforwardly expressed relations that build on classes and interfaces.

Figure 5.2: A Partial Dependency Graph of Legacy Datalink

The Legacy Datalink Dependency Graph

Figure 5.2 depicts the dependency graph of the legacy Datalink. It is strenuous to follow such a complex and messy graph from the first sight. The intuitionistic view of those chaotic connections makes it difficult to find a start point. To make matters worse, the red lines which represents bidirectional dependencies is inundated in the graph, and such relations creates a large amount of circular dependencies in the code base, which increases the efforts of potential maintenance to a great extent.

As this code base has been maintained for over twenty years, it is hard to tell which class or pieces of code is not used anymore and has no actual influence on the maintenance. We can merely tell from the appearance that the complexity increases much more than that in ALA, and the unceremonious, disordered and crossing connections

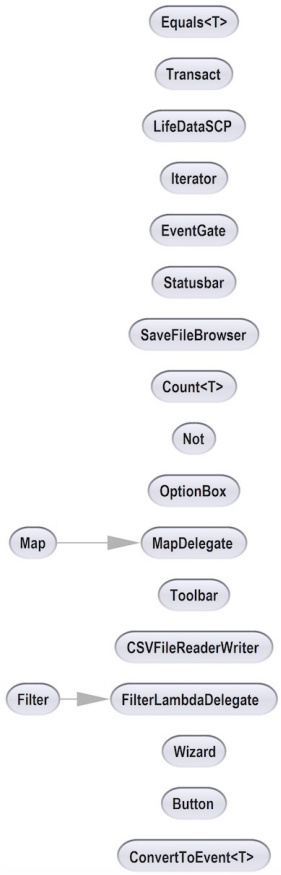


Figure 5.3: Partial Domain Abstractions of ALA Datalink

between elements is extremely hard to follow by intended maintainers.

5.1.2 Zero Coupling of ALA's Domain Abstractions

Zero Coupling is a specialized and prominent feature of the domain abstraction layer of ALA. According to Spray and Sinha (2018), domain abstractions should have no direct or indirect dependency with any other domain abstractions at compiling time. While in run-time, this happens inevitably because the domain abstractions are wired and interact with each other to express a specific requirement.

Figure 5.3 illustrates a part of the domain abstractions dependency graph generated by `NDepend`. We can specify that all the abstractions are isolated and there are no

relations with others in this layer. However, *Map* and *Filter* were two exceptions that depend on the delegates. The delegates provide lambda expression for customizing the functionalities when the abstraction is instantiated, and the lambda expression allows a program to use anonymous methods as parameters (Schildt, 2010), which provides more flexibility and extensibility for a class. In this case, it helps us to move the application knowledge out of the two abstractions, and leave them as the decisions for the application. As the delegates are only used for the two abstractions, this dependency does not break the "zero coupling" feature of ALA.

The *Zero Coupling* mechanism of ALA makes domain abstractions extraordinarily modular and testable, thus creating stable and reusable foundations for future development and maintenance.

5.2 Modularity

According to the *Modularity* metrics refined in Section 2.3.1, the ones used here are Coupling Between Objects classes (CBO) and Cyclomatic Complexity (CC). The CBO can be classified as afferent coupling and efferent coupling. Afferent coupling measures the number of components that depends on a component, whereas efferent coupling assesses the number of components a component depends on. In this case, the unit of the component is classes and interfaces, as they are the fundamental elements of both ALA and the legacy code bases.

5.2.1 Components Coupling

Figure 5.4 and 5.5 illustrates the afferent coupling and efferent coupling of the two code bases respectively. Generally, the coupling of ALA code base is much lower than the legacy one, which means the ALA components are more independent and they have lower a coupling value. Moreover, the diagram also provides evidence on the "zero

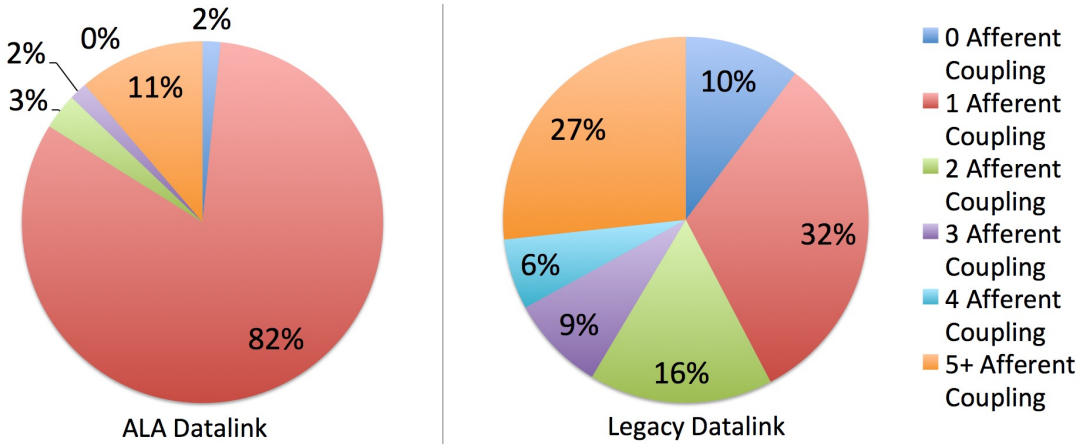


Figure 5.4: Comparison of Afferent Coupling

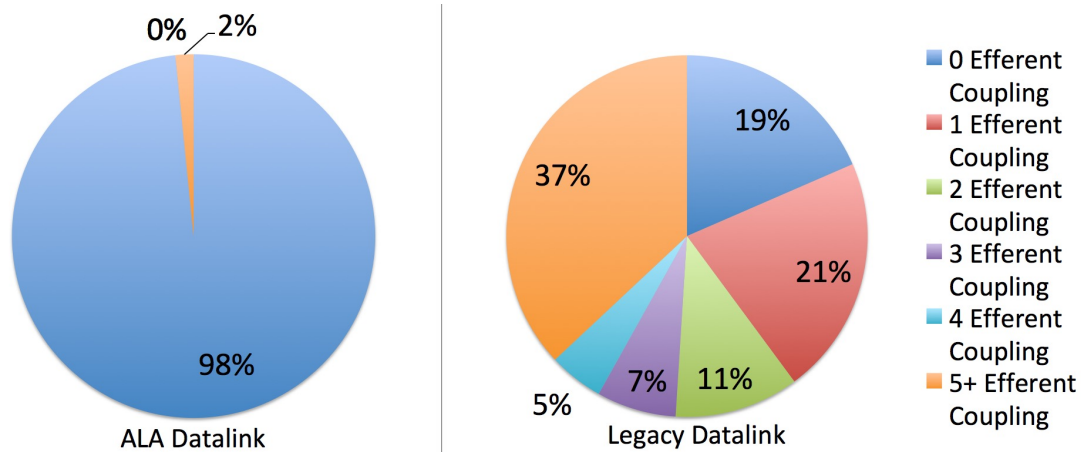


Figure 5.5: Comparison of Efferent Coupling

coupling" feature of ALA. However, the "zero coupling" feature refers more to the efferent coupling rather than the afferent coupling, because 98% of ALA's components demonstrates zero efferent coupling whereas only 2% depicts zero afferent coupling.

According to ISO 25023 (2016), *Component Coupling* is the ratio between the components that implemented to be independent and designed to be independent. In this case, the result of ALA Datalink is 100%, because the implementations conform completely to the design that all the domain abstractions were independent from their parallel elements. In terms of the legacy Datalink, we cannot infer the result because it

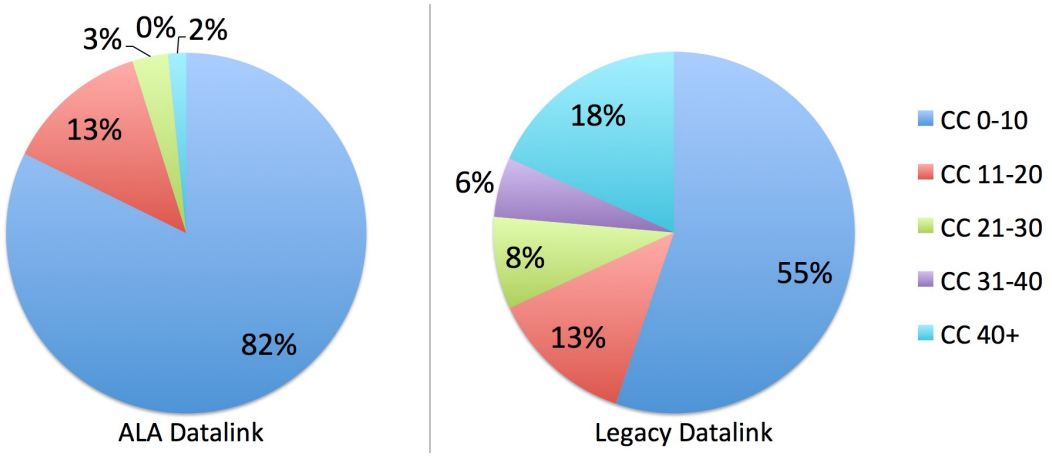


Figure 5.6: Comparison of Cyclomatic Complexity

has been maintained for over 20 years, so it is impossible to explore the original design to find the components that need to be isolated. Nevertheless, from the overall view, components of ALA Datalink are intended to be much more modular than the legacy ones.

5.2.2 Cyclomatic Complexity Adequacy

Figure 5.6 presents the Cyclomatic Complexity (CC) statistics of the two code bases. From the pie charts, we cannot calculate the *Cyclomatic Complexity Adequacy*. Because ISO 25023 (2016) defines it as the percentage of components that does not exceeds a specific CC value (Section 2.3.1). Besides, it is hard to determine the specific CC value here, as it varies in different projects and organizations. Thus, in this case, we simply used the mean CC value of the two code bases as the threshold, and calculated the *Cyclomatic Complexity Adequacy* of the two code bases respectively.

Code Base	Mean CC	CC Adequacy (ALA)	CC Adequacy (Legacy)
ALA Datalink	8.32	67.74%	96.77%
Legacy Datalink	27.83	50.56%	74.61%

Table 5.1: Comparison of Cyclomatic Complexity Adequacy

Table 5.1 presents the results of the calculations. The mean cyclomatic complexity value of ALA and legacy Datalink is 8.32 and 27.83 respectively. If the former is used as threshold, ALA can achieve 67.74% for its *Cyclomatic Complexity Adequacy*, while it is 50.56% for the legacy one. If the later is used as threshold, the *Cyclomatic Complexity Adequacy* of ALA increases to 96.77% whereas the legacy Datalink grows up to 74.61%.

5.2.3 Summary and Correlation with Preliminary Assessment

In summary, no matter if we consider *Components Coupling* or *Cyclomatic Complexity Adequacy*, ALA code base performs much better than the legacy one. However, as the legacy Datalink has been maintained for over 20 years, it is hard to specify the components design for calculating the rate of *Components Coupling*. Under such circumstances, it is impossible to further quantify the improvements in *modularity* ALA brings by comparing with the legacy application.

Nevertheless, the modularity of ALA has increased by at least 24% even if we merely count the growth in *Cyclomatic Complexity Adequacy*. Furthermore, "zero coupling" of ALA is obviously a significant feature that contributes to modularity, and it eliminates the dependencies between domain abstractions. Such result provides evidence to support the high modularity of ALA and it conforms to the result of the preliminary assessment in Section 2.3.1, where we concluded that the modularity of ALA is considerably high by comparing its mechanisms with the ISO 25023 (2016) measures and the refined metrics.

5.3 Reusability

ISO 25023 (2016) describes that reusability measures consists of *Reusability of Assets* and *Coding Rules Conformity*. The former assesses the ratio between the assets that

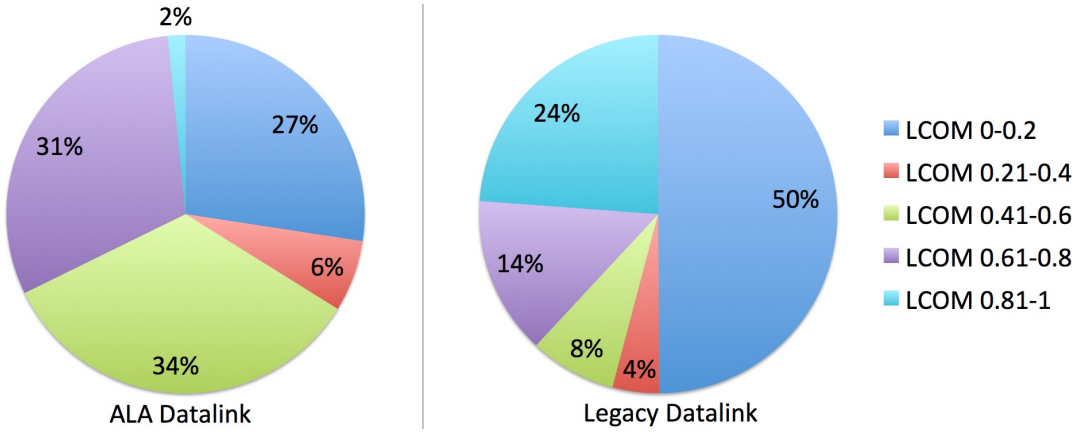


Figure 5.7: Comparison of Lack of Cohesion Methods (LCOM)

implemented to be reusable and the total assets of the code base. While the later measures the percentage of modules that conform to coding rules for a specific system. In this case, the unit of assets and modules is united as the class or interface, as both of them are the fundamental elements of the two code bases.

5.3.1 Reusability of Assets

The metrics refined in section 2.3.2 for measuring *reusability* are Coupling Between Objects classes (CBO), Lack of Cohesion Methods (LCOM), Weighted Methods per Class (WMC), Number of Children (NOC) and Instantiated Times (IT). The first three metrics aim to measure the possibility that the assets can be potentially reused, while the last two metrics intend to measure the actual times the assets has been reused.

CBO has been discussed previously in Section 5.2, where we concluded that the coupling of ALA code base is much lower than the legacy one. In this case, we calculated the mean CBO of the two code bases, the result of ALA is 3.61 while it is 10.75 for the legacy Datalink. Thus, no matter from the average coupling values, or from the overall trend presented before, the components of ALA code base are much more possible to be reused.

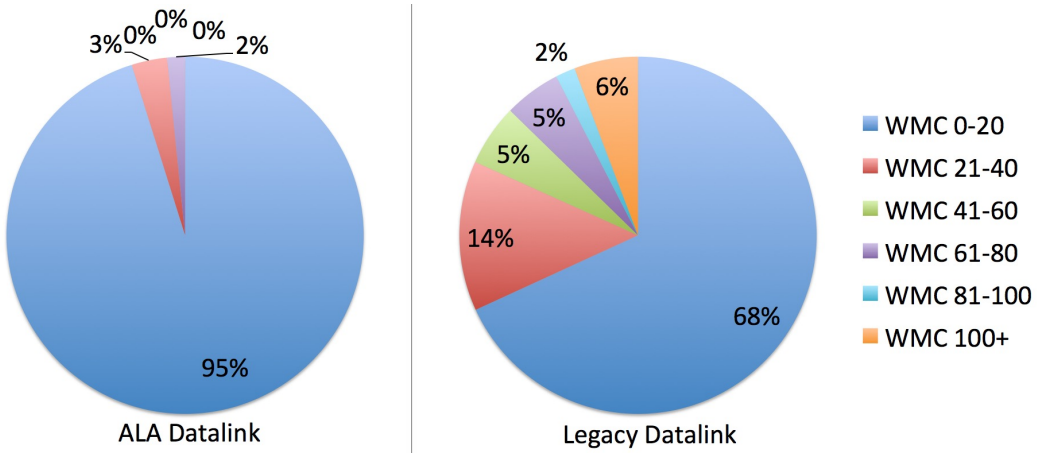


Figure 5.8: Comparison of Weighted Methods per Classes (WMC)

Figure 5.7 and 5.8 illustrates the LCOM and WMC result of the two code bases. According to Chidamber and Kemerer (1994), both high LCOM and WMC value increases the complexity of a class, which limit the reuse possibility. The assets with LCOM value 0-0.2 occupy 50% in the legacy Datalink, which is much more than the 27% of the ALA one. Thus we conclude that the legacy code base is more reusable. However, in terms of WMC, it is apparent that the ALA code base is more reusable because 95% of the assets have lower WMC value. Such adverse result of LCOM and WMC does not mean there exist conflicts, because LCOM cares about the code level reusability, whereas WMC aims to measure the method level reusability.

Figure 5.9 and 5.10 depicts the NOC and IT metric result of the two code bases. As inheritance is not used in ALA, the NOC simply represents the number of interface implementations for ALA. The greater NOC values represents greater reuse (Chidamber & Kemerer, 1994). In this case, the percentages of reused interfaces and ancestor classes are approximately identical for the two code bases. With respect to IT, it gives the actual number of a classes has been instantiated. The graph demonstrates that ALA classes have been reused for more times, and the reusability of ALA code base is higher.

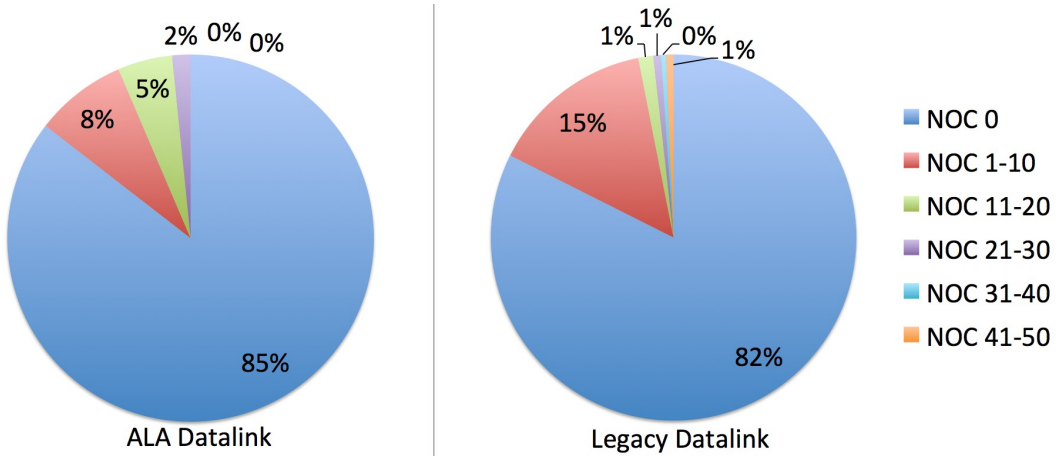


Figure 5.9: Comparison of Number of Children (NOC)

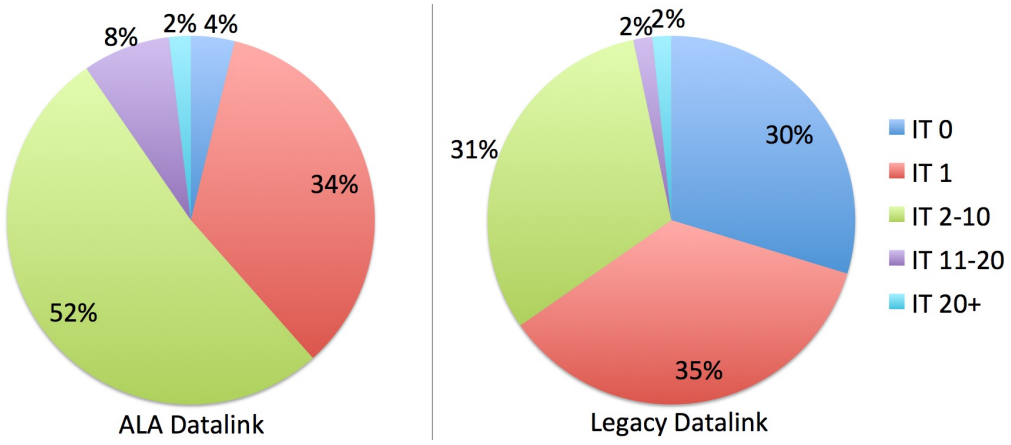


Figure 5.10: Comparison of Instantiated Times (IT)

5.3.2 Coding Rules Conformity

ALA and the legacy Datalink follow different coding rules. According to the metrics refined in Section 2.3.2, the metrics for ALA and the legacy code base are *Layer Violation* and *Circular Dependency Violation*.

Code Base	Metric	Result	Coding Rule Conformity
ALA	Layer Violation	0%	100%
Legacy	Circular Dependency Violations	15.14%	84.86%

Table 5.2: Coding Rules Conformity Results

Table 5.2 illustrates the results of the two metrics. *Layer Violation* of ALA is zero because during the re-development of Datalink, we crucially followed ALA rules to make sure that no incorrect dependency would be introduced. On the other hand, the legacy Datalink did not follow its rules as expected by the maintainers. Among the total 449 classes and interfaces, there totally exists 141 circular dependencies, and the number of correlated classes is 68. Therefore, we conclude that the ALA code base has higher *Coding Rules Conformity*.

5.3.3 Summary and Correlation with Preliminary Assessment

We applied five metrics to measure the reuse ratio of the two code base. Table 5.3 illustrates the statistic of assets reuse ratio of the two code bases. We simply picked the percentage most reusable assets from each pie chart to constitute the table. We gave a mean value of reuse ratio for the two code bases, the figure of ALA is 49.8%, which is higher than 37.1% of the legacy Datalink.

Metrics	ALA	Legacy	Description
CBO	50%	14.5%	mean value of 0 afferent and efferent coupling data
LCOM	27%	50%	pick the value between LCOM 0 to 0.2
WMC	95%	68%	pick the value between WMC 0 to 20
NOC	15%	18%	pick the value more than (include) NOC 1
IT	62%	35%	pick the value more than (include) IT 2
Mean	49.8%	37.1%	mean reusable ratio of the two code bases

Table 5.3: Summary and Comparison of Assets Reusability

However, such result does not conform to the expected reuse ratio we carried out in the preliminary assessments, where we concluded that *Reusability of Assets* of ALA could achieve 99% theoretically, whereas the measured result of ALA here is merely 49.8%. Nevertheless, both of *Reusability of Assets* and *Coding Rules Conformity* of ALA is higher than the legacy Datalink. The former exceeds 12.7% while the later over-tops 15.14%. Thus we conclude that the average reusability enhancement of ALA

is 13.92% comparing with the legacy Datalink.

5.4 Analysability

According to the metrics refined in section 2.3.3, the metrics used for measuring analysability is CBO, LCOM, LOC and CP. However, these metrics does not conform to the original measures of ISO 25023. In this case, we mainly adopted the metrics to measure analysability from two aspects, which come from analysability definition of ISO 25023 (2016):

1. The ease of ripple effects identification of an intended change;

2. The ease to locate the parts with deficiencies or causes of failure, and intended maintenance activities.

5.4.1 Ripple Effects Identification

Ripple effects of changes are directly associated with the extent of coupling. Coupling of analysability assessments here can be classified as internal coupling and external coupling. The internal coupling is measured by LCOM, where it emphasizes the methods and properties of a class. While the external coupling is measured by CBO, where it cares more about the dependencies between classes.

Both CBO and LCOM has been discussed before in section 5.2 and 5.3. However, the two metrics gave us adverse results when applying them on the two code bases. We analysed the reason for that and conclude ALA is effortless in identifying ripple effects between classes, while the legacy code base is easier to identify ripple effects inside classes, as illustrated in Table 5.4.

	ALA	Legacy	Description
Mean CBO	3.61	10.75	For this metric, the ALA code base performs better on the external analysability between classes.
Mean LCOM	0.43	0.37	For this metric, the legacy code base performs better on the internal analysability of classes, mainly on the properties and methods.

Table 5.4: Ease of Ripple Effects Identification

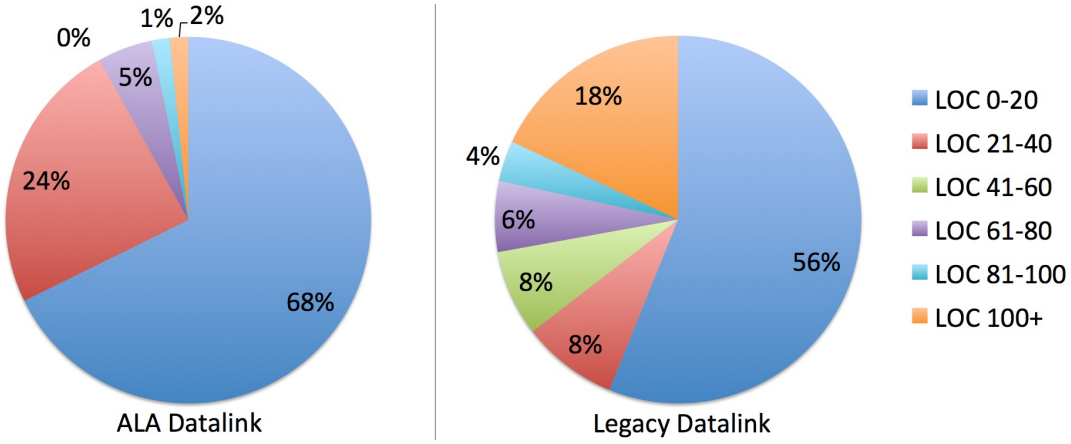


Figure 5.11: Comparison of Lines of Code (LOC)

5.4.2 The Ease of Locating Failures or Change Parts

In this section, we applied the metrics LCOM, LOC and CP to measure the ease of locating failures or changes to parts. The reason we used LCOM again is that failures of the assets usually lie on the internal side of them. The result of LCOM has been discussed before in Section 5.3, we concluded that the legacy code base is considered to be easy to locate failures and changes to parts.

Figure 5.11 and 5.12 depicts the result of LOC and CP for the two code bases. For LOC, we conclude ALA code base is more analysable, because classes of ALA intend to have less lines of code. In terms of CP, excessive low and high CP value would decrease the analysability. Arafat and Riehle (2009) studied different open source projects, concluded that the average commenting percentage is 19%. Such value is less than the mean value 33% of ALA, and more than the mean value 14% of the legacy

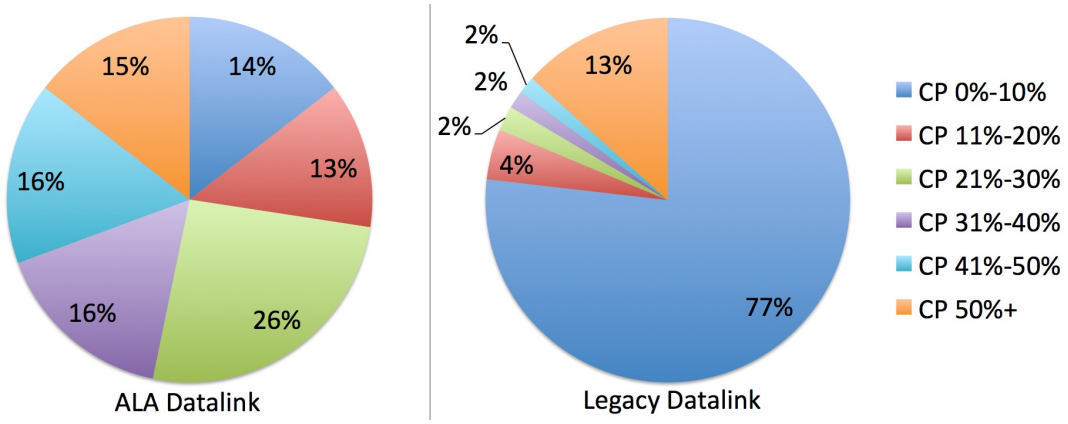


Figure 5.12: Comparison of Commenting Percentage (CP)

code base. However, if we consider the sections that close to 19%, the two sections 11% to 20% and 21% 30% were counted, and the result of ALA and the legacy code base changed to 39% and 6% respectively. Hence, from this point, the ALA code base is considerably more analysable.

5.4.3 Summary and Correlation with Preliminary Assessment

In this section, four metrics were applied to measure the analysability of the two code bases. In summary, CBO, LOC and CP results better analysability for ALA code base, whereas LCOM results better analysability for the legacy one. We simply picked the most analysable sections presented in the pie chart for the statistic of analysability, as shown in Table 5.5. Particularly, as both of CBO and LCOM contributes to *Ripple Effects Identification*, we calculated LCOM twice to come up with the mean analysability value.

When correlating this result with the preliminary assessment, the actual analysability of ALA is higher than the expectation. From the preliminary assessment, we concluded that the analysability of ALA might be low due to the inconformity between ALA's mechanism and ISO 25023 (2016) measures, as well as the unrealistic utilization of

Metrics	ALA	Legacy	Description
CBO	50%	14.5%	mean value of 0 afferent and efferent coupling data
LCOM	27%	50%	pick the value between LCOM 0 to 0.2
LOC	68%	56%	pick the value between LOC 0 to 20
CP	39%	6%	pick the value of sections close to 19%
Mean	42.2%	35.3%	mean analysable ratio of the two code bases

Table 5.5: Summary and Comparison of Analysability

the refined metrics. However, according to the actual result presented in Table 5.5, the mean analysable percentage of ALA is 42.2%, while the ratio of the legacy code base is 35.3%. Thus, the analysability of ALA is higher than the legacy Datalink.

5.5 Modifiability

According to ISO 25023 (2016), modifiability measures involve *Modification Efficiency, Modification Correctness and Modification Capability*. As these measures were proper ones for modifiability, we did not refine them and merely apply them to the two code bases directly. However, we were not able to measure *Modification Correctness* due to time limitation, as it requires the assessor to track the correctness of those modifications for a specific period.

To measure *Modification Efficiency* and *Modification Capability*, we involved six real user stories that proposed by the product manager of Datamars. However, we only implemented two of them due to the time limitation of the legacy Datalink maintainer. The time-consumption of the rest four user stories were merely estimated and analysed to assess the efficiency and capability.

5.5.1 Modification Efficiency

Figure 5.13 and 5.14 illustrates the time-consumption of the modification tasks for the two user stories (refer Appendix B to see the detailed user stories) on the ALA code

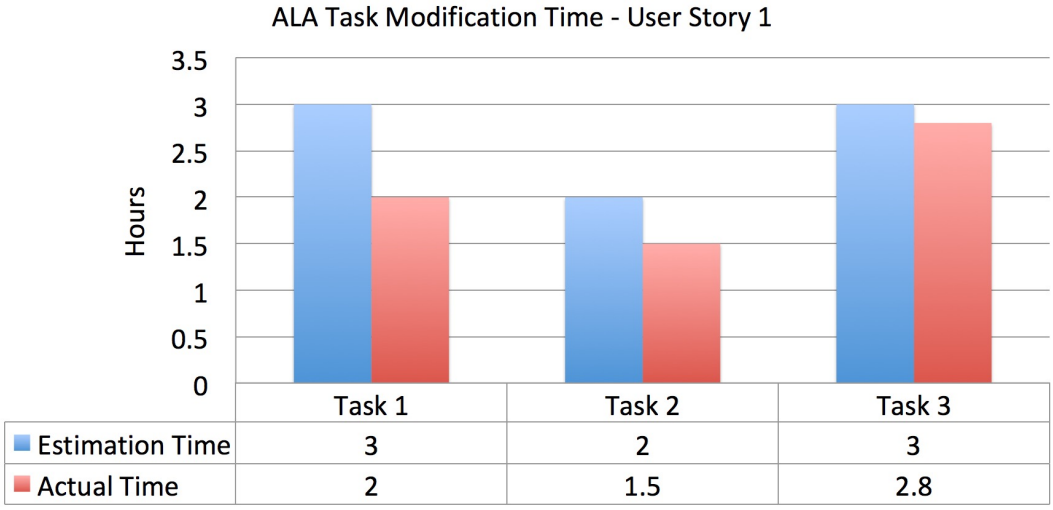


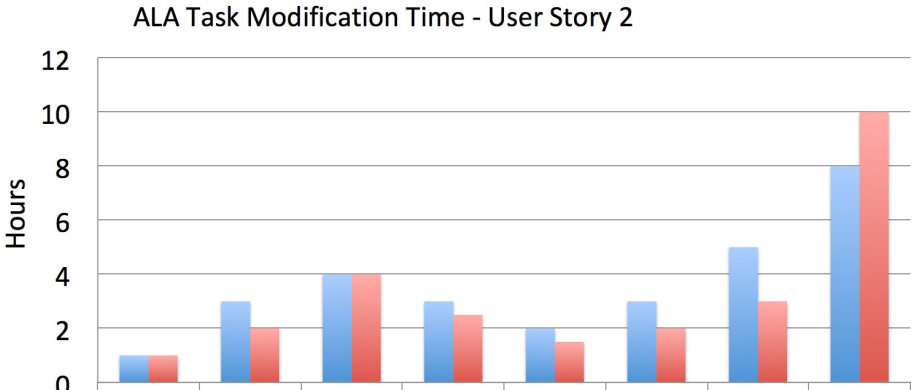
Figure 5.13: Modification Efficiency of ALA - User Story 1

base. For the legacy one, as the maintainer intended to implement the user story as a whole, we were not able to record the time-consumption for the sub-tasks. However, we recorded the total time-consumption for the two user stories on the legacy code base, which allows us to calculate the modification efficiency of it.

Code Base	User Story 1	User Story 2	Mean Modification Efficiency
ALA Datalink	78.33%	84.58%	81.46%
Legacy Datalink	80%	83.33%	81.67%

Table 5.6: Modification Efficiency Comparison

The corresponding implementation time and estimation time for the two user stories were 4/5 and 20/24 respectively for the legacy code base, and we calculated the *Modification Efficiency* and compare it with the ALA one, as shown in table 5.6. The result demonstrates that the *Modification Efficiency* of ALA and the legacy Datalink are approximately identical.



	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8
Estimation Time	1	3	4	3	2	3	5	8
Actual Time	1	2	4	2.5	1.5	2	3	10

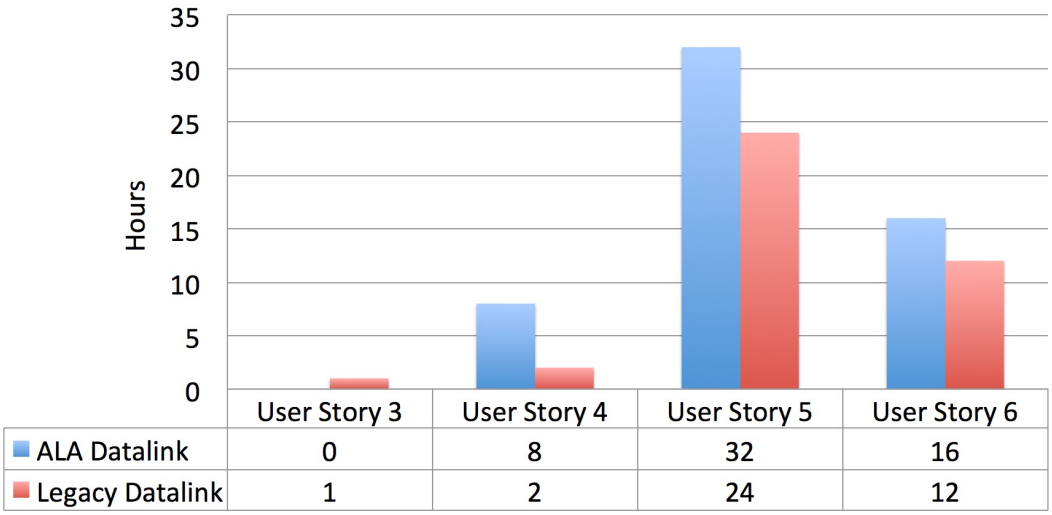
Figure 5.14: Modification Efficiency of ALA - User Story 2

5.5.2 Modification Capability

Figure 5.15 illustrates the estimation of time consumption for the four additional user stories (refer Appendix B.1 to see the details). Despite of user story 3, the estimated time consumption of the other user stories of ALA is higher than the legacy code base. Generally, ALA requires 56 hours in total to finish all the four user stories, whereas the legacy Datalink merely needs 39 hours.

Therefore, during a unit time period (we assume it is one hour), the *Modification Capability* of the legacy code base is 1/39, whereas the ALA code base is 1/56. Correspondingly, the percentage of the *Modification Capability* of the two code bases are 2.6% and 1.8% respectively. The ALA code base decreases 30.78% when comparing with the legacy one.

However, if we specify the *Modification Capability* of the implemented 2 user stories, the total time for the legacy and ALA code base is 29 and 32.3 respectively. Under such circumstance, the *Modification Capability* is 1/29 and 1/32.3, which are 3.45% and 3.1% in percentage, and the ALA code base decreases 10.21% comparing with the legacy one.



	User Story 3	User Story 4	User Story 5	User Story 6
ALA Datalink	0	8	32	16
Legacy Datalink	1	2	24	12

Figure 5.15: Estimation of Modifications Comparison

5.5.3 Summary and Correlation with Preliminary Assessment

Overall, the *Modification Efficiency* of the two code bases is approximately identical. In this case, the factor that determines the modifiability is mainly *Modification Capability*. From the assessments we carried out previously, we can conclude that the modifiability of the legacy code base is higher than the ALA one. Such result does not conform to the preliminary assessment, where we concluded that modifiability of ALA is medium. The result here demonstrates that the modifiability of ALA might be low.

5.6 Testability

According to the metrics refined in Section 2.3.5, the ones used to evaluate testability are CBO, LCOM, WMC and LOC. As these metrics do not conform to the original measures of ISO 25023, in this case, we merely correlated them with testability definition proposed by ISO 25023 (2016), which is the ease of test criteria establishment and execution.

5.6.1 Ease of Test Criteria Establishment and Execution

The metrics CBO and LCOM have been discussed before in the previous sections. According to Chidamber and Kemerer (1994), high-coupling components require more test efforts to make them rigorous, whereas high LCOM value makes the components more difficult to test. We conclude that the former results better testability in ALA code base, while the later results better testability in the legacy code base.

In terms of WMC and LOC, both of them are correlated with the size and complexity of the class. The complexity determines the time that might be consumed to develop and test the class (Chidamber & Kemerer, 1994). In this case, the two metrics have been discussed before, where the results of ALA code base performed better than the legacy one. Thus, for both of the two metrics, we conclude that the ALA code base has higher testability.

5.6.2 Summary and Correlation with Preliminary Assessment

In this section, we carried out four metrics to measure the testability of the two code bases. Three of them indicates that ALA has higher testability, while one of them demonstrates the legacy code base possess higher testability. We picked the most representative sections in the pie charts to seek the improvements of ALA on testability, as shown in Table 5.7.

Metrics	ALA	Legacy	Description
CBO	50%	14.5%	mean value of 0 afferent and efferent coupling data
LOC	68%	56%	pick the value between LOC 0 to 20
LCOM	27%	50%	pick the value between LCOM 0 to 0.2
WMC	95%	68%	pick the value between WMC 0 to 20
Mean	60%	47.13%	mean testable ratio of the two code bases

Table 5.7: Summary and Comparison of Testability

The figure demonstrates that the highly testable assets of ALA is 60%, whereas it is

47.13% of that for the legacy code base. However, as both of the two code bases did not provide measurable concretes for the original measures defined by ISO 25023 (2016), we merely correlated them with the refined metrics. Consequently, such metric results conform to the preliminary assessment, where we concluded that the testability of ALA is particularly high.

5.7 Analysis, Discussion and Summary

In this chapter, we assessed and compared the performance of the two code bases based on ISO sub-characteristics i.e. *Modularity, Reusability, Analysability, Modifiability and Testability*. Except *Modifiability*, the other sub-characteristics performed as expected or exceeded the expectations. This section analyses the reasons for the low *Modifiability* of ALA code base, and then gives a summary to this chapter.

5.7.1 Analysis of ALA's Low Modifiability

According to ISO 25023 (2016), *Modifiability* is impacted by *Modularity* and *Analysability*. However, in this case, although *Modularity* and *Analysability* of ALA Datalink were much higher than the legacy one, the actual *Modifiability* of ALA was adversely low. We analysed the tasks carried out for the implemented two user stories, conclude that such low *Modifiability* was impacted by the construction of new abstractions and the complexity of the application layer.

Construction of New Abstractions

As we discussed in Section 4.3, the implementation of new user stories was categorized as perfective maintenance, and such kind of maintenance in ALA usually needs to wire existing abstractions to satisfy the requirements, which is considered to be effortless.

However, if the new requirements need to design and implement new abstractions, Boehm (1996) states that the intention of developing such reusable assets would increase the effort. Table 5.8 illustrates the new programming paradigms and domain abstractions that built for the two user stories (refer the detailed decomposition of user stories in Appendix B.2), both of the two user stories required new abstractions, which increased the time-consumption.

User Story	New Paradigms and Abstractions Implemented
User Story 1	DeviceIdSCP
User Story 2	IIterator, ListOfFiles, SelectExternal, FileSessions, ConvertIteratorToTable

Table 5.8: New Paradigms and Abstractions for New User Stories

Complexity of The Application Layer

According to Spray and Sinha (2018), the application layer of ALA is designed to to manage all the knowledge that is specific to the application. On the one hand, such mechanisms make the abstractions free of application-specific knowledge and make them more reusable and stable, improving maintainability. On the other hand, if we implement more functionalities, the growing size and complexity of the application file increases the difficulties of analysing and modifying it, which decreases the maintainability of the code base.

US 1/Tasks	T1	T2	T3
Estimated(hrs)	3	2	3
Actual(hrs)	2	1.5	2.8
Efficiency	150%	133.33%	107.14%

Table 5.9: Task Efficiency of User Story 1

Table 5.9 and 5.10 illustrate the task efficiency of the two implemented user stories, and we calculated the results as the division of actual and estimated implementation time. For both user stories, the last task in the tables is the one that correlated with application

US 2/Tasks	T1	T2	T3	T4	T5	T6	T7	T8
Estimated(hrs)	1	3	4	3	2	3	5	8
Actual(hrs)	1	2	4	2.5	1.5	2	3	10
Efficiency	100%	150%	100%	120%	133%	150%	167%	80%

Table 5.10: Task Efficiency of User Story 2

(wiring instances of abstractions to satisfy the requirements), we can conclude from the figure that:

1. There exists a trend that the more complex the user story is, the more time was spent on the application layer. Correspondingly, the efficiency of the application task decreases.

2. For both user stories, the efficiency of the application tasks is the lowest. Considering the efficiency of all the other tasks exceed 100%, the application task becomes the main factor that impact *Modification Efficiency* we evaluated in Section 5.5.1.

3. If we have all the domain abstractions implemented, the only activity carried out in the maintenance would only be the application task, which is the desirable maintenance in ALA. With more and more abstractions being implemented, there exists considerable possibility for such circumstance, and the *Modifiability* of ALA code base merely need to consider the application layer, which would increase *Modifiability* to a great extent.

5.7.2 Discussion of the Overall and Long-term Maintainability

According to the evaluation carried out in the previous sections, *Modularity, Reusability, Analysability and Testability* of ALA are considerably higher than the legacy code base, whereas *Modifiability* of ALA is lower. We were not able to integrate all the results of each measure and each sub-characteristic into *Maintainability*, as we do not know to

what extent each result contributes to *Maintainability*. Therefore, we merely discuss each sub-characteristic respectively.

Spray and Sinha (2018) state that ALA aims to improve the long-term maintainability of a software code base. If we consider long-term maintenance for a ALA software code base, we can conclude that although we have not implemented enough reusable abstractions currently, the ongoing maintenance would help to improve the maintainability of ALA:

1. *Modularity* would not change too much if more abstractions and wiring code were added, because the "Zero Coupling" feature would always keep ALA highly modular.

2. *Reusability* would increase with maintenance being carried out, because more abstractions would be implemented and reused.

3. *Analysability* might decrease as more new features means more complexity on application layer, which increases the difficulty for analysis. But this cannot be avoided in any software code bases.

4. *Modifiability* would increase to a great extent, because the point that no or less new abstractions required would be met, and the maintenance merely need to work on the application layer.

5. *Testability* would not be influenced, because we mainly discuss the unit test, which is correlated with single abstractions.

5.7.3 Summary

In this chapter, we carried out the assessments of the legacy Datalink and the re-developed ALA Datalink. Based on the ISO 25010 (2011) quality model, and ISO

25023 (2016) maintainability measures, the results of the evaluations were compared between the two code base on *Modularity, Reusability, Analysability, Modifiability and Testability*. Besides, the dependency graphs of the two code bases were presented, which allows us to have an overall view of the two code bases straightforwardly on their structures.

According to the assessment results, *Modularity, Reusability, Analysability and Testability* of ALA Datalink were higher than the legacy one. However, the *Modifiability* of ALA Datalink was not as expected. We analysed the reason for that based on the two user stories and the sub-tasks we carried out, concluded that the construction of new abstractions and the increasing complexity of the application layer leaded to the low *Modifiability*. Nevertheless, with more and more reusable abstraction being implemented, the overall *Maintainability* of ALA Datalink would keep increasing considerably.

Chapter 6

Conclusion

This chapter gives a summary of the whole research, which was based on the re-architecting and re-developing of an existing desktop application with ALA, and evaluating the maintainability of that code base based on ISO 25010 (2011) quality model and ISO 25023 (2016) measures within a commercial environment.

The rest of this chapter is organized as follows. Section 6.1 provides a summary to this research as a whole and discusses the activities carried out and observations obtained in the previous chapters. Section 6.2 summarizes the answers of the last three research questions (the first two questions were answered in Chapter 2). Section 6.3 discusses the contribution of this research, mainly on the feasibility of utilization in commercial projects, as well as maintainability in long-term maintenance for ALA. Finally, three potential directions of future works are involved for future researches.

6.1 Summary

The scope of this research is to explore the maintainability of the Abstraction Layered Architecture (ALA) (Spray & Sinha, 2018) through a real C# desktop application.

Maintainability measures were extensively reviewed in the systematic literature

review, from architecture-level, design-level, code-level to process-level (Section 2.2.2). The accuracy of the categories increases when measuring *maintainability* because they require different objects for assessments. However, applying only one type of measures does not provide enough evidence to prove ALA's *maintainability*. Therefore, the ISO 25010 (2011) quality model was reviewed and considered. It decomposes *maintainability* as *modularity, reusability, analysability, modifiability and testability*, which simplifies the complexity of *maintainability*. Moreover, ISO 25023 (2016) provides measures for each sub-characteristic, which integrates measures in design, code and process level. ISO 25010/25023 provide us a holistic model and comprehensive measures for *maintainability*.

ISO 25023 measures cannot be directly used for ALA, whether it is comparing with the mechanisms of ALA, or applying the measures on a concrete ALA code base. The main reason is although ISO 25023 defines the measures for the sub-characteristics, for example, the "Coupling of Components" measure for *modularity*, it does not give measures to assess "Coupling". Therefore, we refined the ISO 25023 measures, making them measurable for not only the mechanisms of ALA, but also for a potential code base (Section 2.3).

In order to investigate the way that ALA supports *maintainability*, a preliminary assessment of ALA was carried out by comparing the refined ISO 25023 measures with ALA's mechanisms. The assessment was performed through all the sub-characteristics of *maintainability*, and the result of each sub-characteristics was presented respectively (section 2.3). However, as *modifiability measures* were correlated with concrete maintenance, we did not assess it and merely predicted the modifiability based on the results of *modularity* and *analysability*.

We followed Scrum (Schwaber, 1997) to re-develop Datalink with ALA. Such method was intended to simulate the real environment of Datamars, so we could

investigate the feasibility of ALA under such circumstance (Section 4.2.2). The re-developed code base involves the same functional and non-functional requirements of the legacy one, which created foundations for the later assessments and comparative experiments on the sub-characteristics of the two code bases (Section 4.1.1. Furthermore, a general software development method with ALA was formulated based on the *software development life cycle* we followed, as well as the actual activities we carried out in the re-development of Datalink (Section 4.4).

We opted to use `NDepend` (Smacchia, 2007) to apply all the refined measures on the two code bases to compare the result of *modularity, reusability, analysability and testability*. As *modifiability* is correlated with real maintenance, we carried out two user stories which were presented by the product manager of Datamars on the two code bases, recorded the effort required and compared the result. Apart from that, we analysed the reason that ALA possesses high *modularity, reusability, analysability and testability*, but low *modifiability*. We concluded that the *maintainability* of ALA would be considerably high if the desired abstractions were implemented before the maintenance.

6.2 Answering Research Questions

As aforementioned, we have five research questions in total and RQ1 and RQ2 were clearly answered in Section 2.4.1. In this section, we discuss the answers of the rest three questions:

> **RQ3.** How can the process of re-architecting an existing C# application using ALA be generalised for use in future projects?
>
> **RQ4.** How do the existing implementation of the C# application and the new re-implementation as a result of answering RQ3 compare when assessed for

maintainability using the measures identified in RQ1?

RQ5. How well does the assessment carried out in RQ4 relate to the expected enhancements in maintainability from ALA (as identified in RQ2)?

6.2.1 Research Question 3

According to the process we generalized in Section 4.4, the ALA development method can be explained from two aspects.

The first aspect mainly refers to *Software Development Life Cycle (SDLC)*. Each step we took in the development conforms more to a Waterfall (Balaji & Murugaiyan, 2012) model, because we have carried out the corresponding activities i.e. *architecture design (Section 4.1), implementation (Section 4.2) and maintenance (Section 4.3)*, and the output of each step would be the input of the next. However, if we investigate the process of each activity, the essence of those activities is close to an Agile (Martin, 2002) model:

1. In the design and implementation, the programming paradigms and domain abstractions were consummated through iterations by specifying requirements with increasing depth and breadth (Section 4.2.2).

2. In terms of maintenance (mainly refers perfective maintenance here), the new requirements are composed in the same way of design and implementation described above (Section 4.3).

The second aspect is more specific to ALA, because ALA works in an innovative way of building layers. Spray and Sinha (2018) state that ALA allows the assets at a higher layer have knowledge dependency (Cataldo et al., 2009) on all the layers at the downside. It does not care about run-time dependency (Nicolau, 1989) because the

instances of the abstractions would interact with each other to make the system running. In this case, feasible measures of building an ALA application are:

1. The *Composite and Decorator* design patterns and the *wiring method* make abstractions of ALA forming a structure, which explicitly expresses the requirements, and is executable (Section 4.2.1).

2. Designing programming paradigms and domain abstractions can be guided by the principles from the experience of Datamars' architect and the correlated activities we carried out (Section 4.1.2).

3. The design documentation is an indispensable part of ALA. The implementation and maintenance should completely follow the design, which in turn improves the maintainability of the code base (Section 4.3).

6.2.2 Research Question 4

The comparison of maintainability assessment of the two code bases was carried out based on the five sub-characteristics defined by ISO 25010 (2011) quality model. Our experiments show that *Modularity, Reusability, Analysability and Testability* of ALA are higher, while *Modifiability* of ALA is lower at a point in time.

Modularity

Modularity assessment consists of *Components Coupling* and *Cyclomatic Complexity Adequacy*. For *Components Coupling*, it is 100% for ALA Datalink because all components (abstractions) were designed and implemented to be independent. While we were not able to measure that of the legacy code base due to the uncertainty of the number of components that were designed to be independent. In terms of *Cyclomatic*

Complexity Adequacy, the figure of ALA Datalink over-tops at least 24.05% of the legacy one (Section 5.2).

Reusability

Reusability assessment involves *Reusability of Assets* and *Coding Rules Conformity*. We calculated the mean value of percentage for potential reusable code, concluded that ALA Datalink is 49.8% while the legacy Datalink is 37.1%. With respect to *Coding Rules Conformity*, the figure of ALA code base is 100% whereas it is 84.86% for the legacy one. Thus, the overall mean increasement of *Reusability* ALA brings is 13.92% (Section 5.3).

Analysability

Analysability assessment includes *Ripple Effects Identification* and *Ease of Locating Failures or Change Parts*. For the former assessment, ALA code base performs better on the external analysis of classes, while the legacy one performs better on the internal analysis of classes. In terms of the later assessment, the ease of ALA Datalink is 42.2%, which is better than 35.3% of the legacy Datalink (Section 5.4).

Modifiability

Modifiability assessment contains *Modification Efficiency* and *Modification Capability*. The *Modification Efficiency* of the two code bases are approximately identical, which are 81.46% for ALA and 81.67% for the legacy code base. However, the *Modification Capability* of the legacy Datalink is higher, which exceeds 10.21% for the implemented user stories and 30.78% for the estimated user stories (Section 5.5).

Testability

Testability is directly related to *Case of Test Criteria Establishment and Execution*. We merely calculated the mean value based on the metric result, concluded that the percentages of testable code for ALA and the legacy code base are 60% and 47.13% respectively (Section 5.6).

6.2.3 Research Question 5

For the five evaluated sub-characteristics, the results of *Modularity, Reusability and Testability* conform to the preliminary assessment we carried out in Section 2.3, whereas that of *Analysability and Modifiability* do not conform to the preliminary assessment.

Analysability of ALA Datalink is higher than the expectation, because the "Zero Coupling" feature and clear structural code layers of ALA makes it highly analysable (Section 5.4).

However, *Modifiability* of ALA is much lower than the expectation. We analysed the reasons, concluded that the construction of new abstractions and the increasing complexity of application file lowered *Modifiability* of ALA. Nevertheless, with more and more abstraction being finished, there exists some point that the maintenance of new requirements merely needs to work on wiring existing abstraction, which is supposed to improve *Modifiability* of ALA to a great extent if long-term maintenance is required (Section 5.5).

6.3 Contributions

In this research, we re-developed an existing C# application of Datamars with ALA, and evaluated the maintainability of the application within a commercial environment, which leads to the contributions.

6.3.1 A Group of High-Quality C# Domain Abstractions and Programming Paradigms

The re-developed Datalink is considered to be a successful ALA application because it not only achieved the requirements completeness of the legacy one, but also demonstrated the maintainability of that code base. The domain abstractions and programming paradigms in this application were elaborately designed and implemented. They can be reused in a wide range of C# desktop application development and are valuable references for any future ALA projects.

We make part of the source code publicly accessible on a GitHub repository (`https://github.com/cdxybf/ALA_Datalink`). Due to the confidential agreement of Datamars, some abstractions that are used to interact with the devices were removed. The rest of the code base involves all the UI and some of the data processing abstractions, as well as all the programming paradigms. Besides, we also kept part of the application file, which demonstrates how the application works with the mechanism of ALA.

6.3.2 A Strategy of Evaluating the Maintainability of Code Bases

The strategy of evaluating the maintainability of the two code bases in this research was constructed on ISO 25010 (2011) quality model and ISO 25023 (2016) measures. Such strategy was built to measure maintainability for a pure code base because:

1. Saraiva et al., (2013) surveyed 568 maintainability metrics which makes the selection of the most relevant ones difficult. Building a strategy on ISO model helps us to select the most related measures.

2. The original ISO measures involve holistic aspects of software development from a higher view that correlated with components, assets and third-party systems, but

it does not provide measures to assess characteristics e.g "Coupling", "Cohesion", which makes it inapplicable directly on a pure code base.

Therefore, comparing with other maintainability measures such as CK suites (Chidamber & Kemerer, 1994) and Maintainability Index (Oman & Hagemeister, 1992), our strategy provides not only the comprehensiveness of the ISO quality model, but also the feasibility and flexibility of applying it on a pure code base.

6.3.3 A General Method to Develop ALA Applications

The re-development of Datalink allows us to propose a general method to develop ALA applications. This method involves non-technical and technical aspects. The non-technical aspect refers *Software Development Life Cycle (SDLC)*, which was completely based on Waterfall (Balaji & Murugaiyan, 2012) and Agile (Martin, 2002) model. The technical aspect includes the rationale of ALA mechanisms which drives the application to run, and the principles of designing programming paradigms and domain abstractions that help to improve the quality of the design and implementation.

This method provides a general way for any potential ALA application development in the future. It is a reliable reference that built based on a real project in commercial environment. Considering that ALA was proposed in 2018, and there is a lack of approaches of guiding software development with it, the method we presented would significantly promotes any utilization of ALA.

6.3.4 Maintainability Improvement Evaluation of ALA in Commercial Software Code Base

Spray and Sinha (2018) describe that ALA improves the maintainability of commercial code bases in the long run by integrating the best practices from real software development experiences. This research provides empirical evidence on the maintainability

improvement of ALA by evaluating and comparing the re-developed ALA code base with the legacy one in commercial environment.

The result demonstrates that *Modularity, Reusability, Analysability and Testability* of the ALA code base is higher than the legacy one. However, the *Modifiability* of the ALA code base is surprisingly low. We analysed the reasons and concluded that it is mainly because the maintenance has not reached the point that no new abstractions need to be constructed. Due to time limitations, we were not able to implement more user stories to reach that point. Nevertheless, regardless of the new abstractions, our opinion is that the overall *Maintainability* of the ALA code base was considered to improve with long-term maintenance being carried out.

6.4 Future Works

There are multiple potential directions for future research, and some of them are discussed as follows.

6.4.1 Ongoing Maintainability Observations of ALA Datalink

The re-developed Datalink is considered as a successful and significant utilization of ALA. The reproduced functionalities of ALA Datalink were approximately restored comparing with the legacy one. However, the accomplishment of ALA Datalink development in this research does not put an end to the project. The re-developed Datalink was originally planned to replace the existing one, and Datamars has already hired three summer students to carry on the maintenance.

Hence, it would be interesting to track the ongoing activities, on one hand it would produce more practice of optimizing the method of ALA application development, and providing evidence of maintainability performance for any possible additional analysis on the other hand.

6.4.2 Utilization of ALA on Other Platforms

The ALA application in this research was specifically developed with C# and WPF (Windows Presentation Foundation). Further on that, we have used many advanced features of C# e.g. *Reflection* (Draheim, Lutteroth & Weber, 2005), *Lambda Expression* (Schildt, 2010) and *Linq* (Pialorsi & Russo, 2007). These features are not the key factors that make ALA work, but also play important roles in the application.

A possible research direction is to explore the feasibility and maintainability of ALA on other platforms with other programming languages. Spray and Sinha (2018) describe that ALA considers any applications with some degree of data flows, events and user interactions, and it also works for pure algorithm problems. Thus, there exists a wide range of possible programs that can be developed by ALA.

6.4.3 Exploring Approach of Optimizing ALA's Application Layer

When we investigated the causes for ALA's low *Modifiability* in Section 5.7.1, one reason is the increasing size and complexity of the application file. Currently, it is easy to maintain this file by following the way abstraction wired in the design documentation. However, as we can foresee, the growing functionalities of a commercial project would continuously increase the size and complexity of the application file. Theoretically, this file might become a god class at some point which includes a very large number of objects that is impossible to maintain.

Therefore, it would be significantly valuable to explore the approach to simplify and share the responsibility of ALA's application layer. However, such approach should still follow the design notions of ALA and its mechanisms, and does not decrease the maintainability of ALA.

6.5 Final Thoughts

It was quite challenging to undertake this research at the beginning, but it is proved to be worthy and valuable to organize the knowledge learned from the papers and books to carry out the whole research. The objective is mainly exploring the maintainability of ALA by re-developing the existing Datalink, and comparing it with the original code base. Based on the re-development process, we also explored the method of developing ALA applications. The result demonstrates that the *Modularity, Reusability, Analysability and Testability* of ALA is particularly high, whereas the *Modifiability* is low at the initial stage, but gradually increases with more and more abstractions being implemented.

The five research questions were properly answered with supporting empirical evidence. This research and results should provide meaningful and valuable approaches, references and confidence for any possible projects that aim to develop with ALA. We are delighted with the research procedure and the contributions made within such a limited time frame.

References

Abreu, F. B. & Carapuça, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th international conference on software quality* (Vol. 186, pp. 1–8).

Adamov, R. & Baumann, P. (1987). *Literature review on software metrics* (Vol. 87). Institut für Informatik, Universität Zürich-Irchel.

Aggarwal, K., Singh, Y., Kaur, A. & Malhotra, R. (2006). Application of artificial neural network for predicting maintainability using object-oriented metrics. *Transactions on Engineering, Computing and Technology*, *15*, 285–289.

Ahn, Y., Suh, J., Kim, S. & Kim, H. (2003). The software maintenance project effort estimation model based on function points. *Journal of Software maintenance and evolution: Research and practice*, *15*(2), 71–85.

Albrecht, A. J. (1979). Measuring application development productivity. In *Proc. joint share, guide, and ibm application development symposium, 1979.*

Albrecht, A. J. & Gaffney, J. E. (1983). Software function, source lines of code, and development effort prediction: a software science validation. *IEEE transactions on software engineering*(6), 639–648.

Arafat, O. & Riehle, D. (2009). The commenting practice of open source. In *Proceedings of the 24th acm sigplan conference companion on object oriented programming systems languages and applications* (pp. 857–864).

Balaji, S. & Murugaiyan, M. S. (2012). Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, *2*(1), 26–30.

Baldwin, C. Y. & Clark, K. B. (2000). *Design rules: The power of modularity* (Vol. 1). MIT press.

Banker, R. D., Kauffman, R. J. & Kumar, R. (1991). An empirical test of object-based output measurement metrics in a computer aided software engineering (case) environment. *Journal of Management Information Systems*, *8*(3), 127–150.

Basili, V. R., Briand, L. C. & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, *22*(10), 751–761.

Baskerville, R. L. (1999). Investigating information systems with action research. *Communications of the association for information systems*, *2*(1), 19.

Bass, L., Clements, P. & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

Belady, L. & Lehman, M. (1972). *An introduction to growth dynamics in statistical computer performance evaluation.* Academic Press: New York NY.

Belady, L. A. & Lehman, M. M. (1976). A model of large program development. *IBM Systems journal*, *15*(3), 225–252.

Bengtsson, P. & Bosch, J. (2003). Architecture level prediction of software maintenance. *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*, 139–147. doi: 10.1109/csmr.1999.756691

Bengtsson, P. O., Lassing, N., Bosch, J. & Van Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, *69*(1-2), 129–147. doi: 10.1016/S0164-1212(03)00080-3

Bennett, K. (1993). An overview of maintenance and reverse engineering. In *The redo compendium* (pp. 13–34).

Bettany-Saltikov, J. (2012). *How to do a systematic literature review in nursing: a step-by-step guide*. McGraw-Hill Education (UK).

Boehm, B. (1996). The cocomo 2.0 software cost estimation model. *American Programmer*.

Briand, L. C., Wüst, J., Daly, J. W. & Porter, D. V. (2000). Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, *51*(3), 245–273.

BSI ISO. (2011). BS ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. *BSI Standards Publication*. Retrieved from `https://bsol-bsigroup-com.ezproxy.aut.ac.nz/PdfViewer/Viewer?pid=000000000030215101`

BSI ISO. (2016). BS ISO/IEC 25023:2016 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality. *BSI Standards Publication*. Retrieved from `https://bsol-bsigroup-com.ezproxy.aut.ac.nz/PdfViewer/Viewer?pid=000000000030280200`

Cataldo, M., Mockus, A., Roberts, J. A. & Herbsleb, J. D. (2009). Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, *35*(6), 864–878.

Chapin, N. (1979). A measure of software complexity. *Proceedings of the 1979 NCC*, 995–1002.

Chen, J. & Lu, J. (1993). A new metric for object-oriented design. *Information and software technology*, *35*(4), 232–240.

Chen, L., Babar, M. A. & Nuseibeh, B. (2012). Characterizing architecturally significant requirements. *IEEE software*, *30*(2), 38–45.

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), 476–493.

Chu, W. C., Chih-Wei Lu, Chih-Hung Chang, Yeh-Ching Chung, Yueh-Min Huang & Baowen Xu. (2002, Aug). Software maintainability improvement: integrating standards and models. In *Proceedings 26th annual international computer software and applications* (p. 697-702). doi: 10.1109/CMPSAC.2002.1045083

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J. & Little, R. (2002). *Documenting software architectures: views and beyond*. Pearson Education.

Coleman, D., Lowther, B. & Oman, P. (1995). The application of software maintainability models in industrial software systems. *Journal of Systems and Software*, *29*(1), 3–16.

Creswell, J. W. (2014). The selection of a research approach. *Research design: Qualitative, quantitative, and mixed methods approaches*, 3–24.

Creswell, J. W. & Creswell, J. D. (2017). *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

Desharnais, J., Pare, F., Maya, M. & St-Pierre, D. (1997). Implementing a measurement program in software maintenance–an experience report based on basili's approach. In *Ifpug conference, cincinnati, oh.*

Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A. & De Lucia, A. (2018). Detecting code smells using machine learning techniques: are we there yet? In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)* (pp. 612–621).

Draheim, D., Lutteroth, C. & Weber, G. (2005). Generative programming for c#. *ACM SIGPLAN Notices*, *40*(8), 29–33.

Dresch, A., Lacerda, D. P. & Antunes, J. A. V. (2015). Design science research. In *Design science research* (pp. 67–102). Springer.

Dubey, S. K. & Rana, A. (2011). Assessment of maintainability metrics for object-oriented software system. *ACM SIGSOFT Software Engineering Notes*, *36*(5), 1–7.

e Abreu, F. B. (1995). The mood metrics set. In *proc. ecoop* (Vol. 95, p. 267).

e Abreu, F. B. & Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3rd international software metrics symposium* (pp. 90–99).

Easterbrook, S., Singer, J., Storey, M.-A. & Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering* (pp. 285–311). Springer.

Fenton, N. E. & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on software engineering*, *25*(5), 675–689.

Fontana, F. A., Mäntylä, M. V., Zanoni, M. & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*(3), 1143–1191.

Fraenkel, J. R., Wallen, N. E. & Hyun, H. H. (2011). *How to design and evaluate research in education*. New York: McGraw-Hill Humanities/Social Sciences/Languages.

Gable, G. G. (1994). Integrating case study and survey research methods: an example in information systems. *European journal of information systems*, *3*(2), 112–126.

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Garcia, J., Popescu, D., Edwards, G. & Medvidovic, N. (2009). Toward a catalogue of architectural bad smells. In *International conference on the quality of software*

architectures (pp. 146–162).

Goel, B. M. & Bhatia, P. K. (2012). Analysis of reusability of object-oriented system using CK metrics. *International Journal of Computer Applications*, *60*(10), 32–36.

Grady, B. (1998). Object-Oriented Analysis and Design with Applications. 2-nd ed. In *Library of congress cataloging-in-publication data.*

Halstead, M. H. et al. (1977). *Elements of software science* (Vol. 7). Elsevier New York.

Harrison, R., Counsell, S. J. & Nithi, R. V. (1998). An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, *24*(6), 491–496.

Harrison, W. A. & Magel, K. I. (1981). A complexity measure based on nesting level. *ACM Sigplan Notices*, *16*(3), 63–74.

Henry, S. & Kafura, D. (1981). Software structure metrics based on information flow. *IEEE transactions on Software Engineering*(5), 510–518.

IEEE Standard for a Software Quality Metrics Methodology. (1993, March). *IEEE Std 1061-1992*, 1-96. doi: 10.1109/IEEESTD.1993.115124

Ionita, M. T., Hammer, D. K. & Obbink, H. (2002). Scenario-based software architecture evaluation methods: An overview. In *Workshop on methods and techniques for software architecture review and assessment at the international conference on software engineering* (pp. 19–24).

Johnson, B. R., De Li, S., Larson, D. B. & McCullough, M. (2000). A systematic review of the religiosity and delinquency literature: A research note. *Journal of Contemporary Criminal Justice*, *16*(1), 32–52.

Jorgensen, M. (1995). Experience with the accuracy of software maintenance task effort prediction models. *IEEE Transactions on software engineering*, *21*(8), 674–681.

Kaur, U. & Singh, G. (2015). A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, *118*(1).

Kazman, R., Bass, L., Abowd, G. & Webb, M. (1994). Saam: A method for analyzing the properties of software architectures. In *Proceedings of 16th international conference on software engineering* (pp. 81–90).

Keele, S. et al. (2007). *Guidelines for performing systematic literature reviews in software engineering* (Tech. Rep.). Technical report, Ver. 2.3 EBSE Technical Report. EBSE.

Khan, K. S., Ter Riet, G., Glanville, J., Sowden, A. J., Kleijnen, J. et al. (2001). *Undertaking systematic reviews of research on effectiveness: Crd's guidance for carrying out or commissioning reviews* (No. 4 (2n). NHS Centre for Reviews and Dissemination.

Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, *33*(2004), 1–26.

Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J. & Linkman, S. (2009). Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology*, *51*(1), 7–15.

Kitchenham, B., Pfleeger, S. L. & Fenton, N. (1995). Towards a framework for software measurement validation. *IEEE Transactions on software Engineering*, *21*(12), 929–944.

Kitchenham, B., Pfleeger, S. L., McColl, B. & Eagan, S. (2002). An empirical study of maintenance and development estimation accuracy. *Journal of systems and software*, *64*(1), 57–77.

Kruchten, P. B. (1995). Architectural Blueprints - The 4+ 1 view model of architecture. *Software, IEEE*, *12*(6), 42–50. Retrieved from `http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=469759`

Kulkarni, U. L., Kalshetty, Y. R. & Arde, V. G. (2010, Nov). Validation of ck metrics for object oriented design measurement. In *2010 3rd international conference on emerging trends in engineering and technology* (p. 646-651). doi: 10.1109/ICETET.2010.159

Kumar, B. (2012, Sep.). A survey of key factors affecting software maintainability. In *2012 international conference on computing sciences* (p. 261-266). doi: 10.1109/ICCS.2012.5

Laing, V. & Coleman, C. (2001). Principal Components of Orthogonal Object-Oriented Metrics. *Software Assurance Technology Center, White Paper SATC-323-08-14, NASA Goddard Space Flight Center, Greenbelt, Maryland*, *20771*.

Li, W. (1998). Another metric suite for object-oriented programming. *Journal of Systems and Software*, *44*(2), 155–162.

Li, W. & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software*, *23*(2), 111–122.

Lippert, M. & Roock, S. (2006). *Refactoring in large software projects: performing complex restructurings successfully*. John Wiley & Sons.

Lorenz, M. & Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

Loucopoulos, P. & Karakostas, V. (1995). *System requirements engineering*. McGraw-Hill, Inc.

Lynch Jr, J. G. (1982). On the external validity of experiments in consumer research. *Journal of consumer Research*, *9*(3), 225–239.

Malhotra, R. & Chug, A. (2016, 10). Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering*, *26*, 1221-1253. doi: 10.1142/S0218194016500431

Malhotra[1], R. & Chug, A. (2012). Software maintainability prediction using machine learning algorithms. *Software Engineering: An International Journal (SEIJ)*, *2*(2).

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Mason, R. L., Gunst, R. F. & Hess, J. L. (2003). *Statistical design and analysis of experiments: with applications to engineering and science* (Vol. 474). John Wiley & Sons.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*(4), 308–320.

Mulrow, C. & Oxman, A. (1997). Cochrane collaboration handbook. *The Cochrane collaboration Handbook (Version 3.0). San Antonio Cochrane collaboration*.

Muthanna, S., Kontogiannis, K., Ponnambalam, K. & Stacey, B. (2000). A maintainability model for industrial software systems using design level metrics. In *Proceedings seventh working conference on reverse engineering* (pp. 248–256).

Nicolau, A. (1989). Run-time disambiguation: coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, *38*(5), 663–678.

Noor, K. B. M. (2008). Case study: A strategic research methodology. *American journal of applied sciences*, *5*(11), 1602–1604.

Oman, P. & Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *Proceedings conference on software maintenance 1992* (pp. 337–344).

Paré, G., Trudel, M.-C., Jaana, M. & Kitsiou, S. (2015). Synthesizing information systems knowledge: A typology of literature reviews. *Information & Management*, *52*(2), 183–199.

Petticrew, M. & Roberts, H. (2008). *Systematic reviews in the social sciences: A practical guide*. John Wiley & Sons.

Pialorsi, P. & Russo, M. (2007). *Introducing microsoft® linq*. Microsoft Press.

Pohl, K. (2010). *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated.

Pree, W. & Gamma, E. (1995). *Design patterns for object-oriented software development* (Vol. 183). Addison-wesley Reading, MA.

Prieto-Diaz, R. & Freeman, P. (1987). Classifying software for reusability. *IEEE software*, *4*(1), 6.

Rasool, G. & Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process*, *27*(11), 867–895.

Rombach, H. D. (1987). A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*(3), 344–354.

Rombach, H. D. (1990). Design measurement: Some lessons learned. *IEEE Software*, *7*(2), 17–25.

Rosenthal, R., Rosnow, R. L. et al. (1985). *Contrast analysis: Focused comparisons in the analysis of variance*. CUP Archive.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. E. et al. (1991). *Object-oriented modeling and design* (Vol. 199) (No. 1). Prentice-hall Englewood Cliffs, NJ.

Ryan, G. (2010). Guidance notes on planning a systematic review. *James Hardiman Library (dostęp na http://www. library. nuigalway. ie/media/jameshardimanlibrary/-content/documents/support/Guidance% 20on% 20planning% 20a% 20systematic% 20review. pdf)*.

Saraiva, J., Soares, S. & Castor, F. (2013). Towards a catalog of object-oriented software maintainability metrics. In *2013 4th international workshop on emerging trends in software metrics (wetsom)* (pp. 84–87).

Sarkar, S., Rama, G. M. & Shubha, R. (2006). A method for detecting and measuring architectural layering violations in source code. In *2006 13th asia pacific software engineering conference (apsec'06)* (pp. 165–172).

Schensul, S. L., Schensul, J. J. & LeCompte, M. D. (1999). *Essential ethnographic methods: Observations, interviews, and questionnaires* (Vol. 2). Rowman Altamira.

Schildt, H. (2010). *C# 4.0: The complete reference*. Tata McGraw-Hill Education.

Schwaber, K. (1997). Scrum development process. In *Business object design and implementation* (pp. 117–134). Springer.

Sharma, T., Fragkoulis, M. & Spinellis, D. (2016). Does your configuration code smell? In *2016 ieee/acm 13th working conference on mining software repositories (msr)* (pp. 189–200).

Sharma, T., Mishra, P. & Tiwari, R. (2016). Designite: a software design quality assessment tool. In *Proceedings of the 1st international workshop on bringing architectural design thinking into developers' daily activities* (pp. 1–4).

Smacchia, P. (2007). Ndepend. *Product description on company website at http://www.ndepend.com*.

Spray, J. & Sinha, R. (2018). Abstraction layered architecture: Writing maintainable embedded code. In *European conference on software architecture* (pp. 131–146).

Steidl, D., Hummel, B. & Juergens, E. (2013). Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)* (pp. 83–92).

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd international conference on software engineering* (pp. 492–497).

Visser, E. (2007). Webdsl: A case study in domain-specific language engineering. In *International summer school on generative and transformational techniques in software engineering* (pp. 291–373).

Wake, S. & Henry, S. (1988). A model based on software quality factors which predicts maintainability. In *Proceedings. conference on software maintenance, 1988.* (pp. 382–387).

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Wu, W., Cai, Y., Kazman, R., Mo, R., Liu, Z., Chen, R., … Zhang, J. (2018). Software architecture measurement—experiences from a multinational company. In *European conference on software architecture* (pp. 303–319).

Xiao, Y. & Watson, M. (2019). Guidance on conducting a systematic literature review. *Journal of Planning Education and Research*, *39*(1), 93–112.

Zhou, Y. & Leung, H. (2007). Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of systems and software*, *80*(8), 1349–1361.

Zohrabi, M. (2013). Mixed method research: Instruments, validity, reliability and reporting findings. *Theory & practice in language studies*, *3*(2).

Appendix A

The Design of ALA Datalink

A.0.1 Domain Abstractions and Programming Paradigms of ALA Datalink

This phase was about getting fully understanding of the requirements and design the domain abstractions. We obey the rules of the ALA architecture and decomposed the system into 4 layers from top to bottom.

- The application layer which depends on the domain abstraction layer and the programming paradigms only.
- The domain abstraction layer which depends on the programming paradigms and the language layer only.
- The programming paradigms layer which depends on the language layer only.
- The programming language layer which we do not have to concern too much about because it is decided by the operating system and it is stable to use.

A.0.2 Notion of the Designed Programming Paradigms

- IUI - Hierarchical containment structure of the UI.

- IEvent - Events or observer pattern (publish/subscribe without data. It can be asynchronous or synchronous. No data, can be bidirectional. Analogous to Reactive Extensions without the duality with iteration - the flow only uses hot observable, and never completes.

- IDataFlow - A data flow with a single scalar value with a primitive data type and a "OnChanged' event. OR think of it as an event with data, the receivers are able to read the data at any time. Or think of it as an implementation of a global variable and an observer pattern, with access to the variable and observer pattern restricted to the line connections on the diagram. Unidirectional - every line is one direction implying sender(s) and receiver(s). You can have multiple senders and receivers. The data is stored in the wire so receivers that don't act on the event can read its value at any time. Receivers cant change the data or send the event. Analogous to Reactive Extensions without the duality with Iteration - unlike RX the flow only uses hot observable, and so never completes. It is just a connection point for drawing convenience and has no special meaning. Lines may have a > or < sign to indicate the direction.

- IRequestResponseDataFlow - Another kind of data flow. However, it is asynchronous.

- ITableDataFlow - A data flow connection that carries rows and self-describing columns that have names and types. Sometimes one directional and sometimes bi-directional depending on the operators (some are bidirectional). Data only moves when an operator called a transact is fired. (Called transact because it transfers the whole table at once so it remains self-consistent). Transactions can be triggered in either direction. Transact can be near the source, or the destination, or anywhere in-between. Thus if the transaction is initiated at the source it is analogous to RX (Reactive extensions) with a hot observable. If the

transaction is at the destination, it is analogous to a query. Some abstractions are sources (SessionListSCP), some are destinations (read-only grid) and many are both (FileReaderWriter, DatabaseTable, SessionDataSCP). Some abstractions transform data and are not considered a source or destination, so they send the data through, transformed, when a transaction occurs anywhere in their stream. The ITableDataFlow interface also has a 'current row'. At the logical level, all rows are transmitted on a transaction. For example, several of these interfaces can be connected directly to a single destination and the destination will show the last transaction sent from any of the sources. In practice only the rows that are actually needed are transferred, so a grid actually requests the data it needs for the display even if the transact itself is at the source. This will cause less data to be brought off the device or from the database. If the sink is say a file or a website, then all data must be queried. When this happens for a slow device, the data is queried in chunks that can be adjusted for efficiency. Many decorators can be implemented such as the equivalent of Select (Map), Where (Filter), Aggregate (Reduce). When the inputs change, a new transaction will be sent from that point. For example when a filter input port changes, a new transaction is sent logically consisting of the new set of rows. Three or more ITableDataFlows can be connected to a single point. Transacts or inactive Gates will block them. If there is more than one active destination, the data flows to all destinations. When there are multiple active sources it will cause an error. Rule: When you have more than two ITableDataFlows connected to a single point, all but one must have a Transact so it is clear which two the data is flowing through on a transaction. Every column has a visibility control.

A.0.3 Notion of the Designed Domain Abstractions

- PopupWindow - A GUI element that is a stand alone window. Implements a IEvent on the left which causes the window to open. On the right it has a list of IUI.The window tells all the IUI in the list to display and arranges them vertically. It has a IEvent port for closing the window. Click events are sent to the contained widget at the location of the click.
- Menubar - Menubar as found on most applications Displays Menu's horizontally. Implements IUI. Has a list of Menus which are IUI.
- Menu - One Menu that sits on a Menubar. Has a list of MenuItems which are IUIs. Displays MenuItems vertically when the menu is clicked. Implements IUI.
- MenuItem - A menu item of a menu that can be clicked. Has a IEvent port on the RHS. When the item is clicked, it generates and event.
- Toolbar - A toolbar typically has tool icons pictures layout horizontally. Implements IUI. Has a list of Tools which are IUIs.
- Tool - A tool used by the the Toolbar. Implements IUI. When clicked, generates an event on its RHS port.
- Horizontal - Arranges contained UI elements horizontally Automatically sizes them.
- Vertical - Arranges contained UI elements vertically. Automatically sizes the widths to be the same and the heights to be shared according to the contain elements fixed size, else, minimum size if there is room, otherwise equally shared.
- Wizard - A window with a list of WizardItems which display as radio buttons, and Buttons for Next, Cancel, and optionally Back. Each WizardItem has it's

own boolean output. When the Next key is pressed, the selected WizardItem's output goes true and emits an event. The Wizard hides but remains in an 'active' state which holds that output. The Back key, which only appears if it is connected somewhere, hides and deactivates the wizard. It typically connects to the left input of a previous wizard (which causes it to re-display as it is usually active but hidden). The Cancel port is a bidirectional IEvent. When the Cancel key is pressed, or an event is received on this port, the wizard hides and deactivates, which releases any held output. See Macro for implementation. Wizard instance builds the other instances according to the macro diagram and information from the list of WizardItems.

- WizardItem - One of the radio buttons of a Wizard. Boolean data output that is true for the selected radio button and false for the rest, and an event. Goes false when the wizard is cancelled or the whole operation completes.

- Panel - A rectangular container of other UI elements with a title.

- Picture - An IUI abstraction that display an image with the input of the image path.

- OptionBox - A UI element that presents a drop down list of options. The set of options are a list of OptionBoxItems. This list is type IDataFlow with a string type, so the OptionBox can directly read the string values of the options. OptionBox itself has two output ports of type IDataFlow: - The string value of the selected option. - the index number of the selected option. The OptionBoxItem has a IDataFlow output port of type boolean, which outputs true while that option is selected, and emits an event when first selected. This provides a convenient way of getting separate boolean outputs for each option when they are needed for

example to control gates. The OptionBox sends true/false events to the OptionBoxItems to tell them they are selected or not selected. An unresolved problem is that IDataFlow interface does not support these events, so another interface type is needed between the OptionBox and the OptionBoxItem. All the other similar pair situations such as MenuBar/Menu, Menu/MenuItem, Toolbar/Tool, RadioButtons/RadioButtonItem, Tabs/Tab, Wizard/WizardItem also have restricted grammar (must be used in pairs, even though other IUI types may actually work.) So they perhaps should all be different interfaces based on IUI, even if they are functionally identical to IUI. Every one of these has a concept of either being in focus or being selected. So the OptionBox/OptionBoxItem pair seems like an unusual case because it doesn't use the IUI interface.

- OptionBoxItem - This type could just be a LiteralString except that we want it to have a boolean output port.
- LiteralString - use StringFormat with nothing connected to the list of inputs.
- Text - Display any kind of text information on UI.
- RightJustify - Layout the sub-elements at the right side of a parent element.
- StringFormat - Really just like a formatted string function found in languages. Takes a string property which contains C-Sharp style data insertion points e.g. "Data=1, Data2=2". Has a port which is a list of IDataFlows that are converted to strings and inserted at the insertion points according to their index numbers, so the ordering of the connections shown in the diagram are important.
- Map - A ITableDataFlow decorator that maps one column in a data stream to a new type/value Column is the column to be mapped Lambda is the conversion expression. A variation is to map all the columns into a new set of columns (like

a C-Sharp RX Select statement can map to a new result class using fields from the source class.)

- Filter - A ITableDataFlow decorator that filters rows Lambda is the expression that can use any of the columns but must result in a boolean.
- Select - Takes an ITableDataFlow and keeps certain columns.
- Gate - Can block a data stream. One version for each of ITableDataFlow, IDataFlow, IEvent, IUI. Has an input data stream and an output data stream. Has a control port that must be a IDataFlow with a boolean type.
- Grid - UI element with rows and columns. The Port Left is an IUI so it can be connected to anything requiring an IUI, such as a window, panel, horizontal, vertical. Port 1 must be an ITableDataFlow which is the data source. The Grid displays the data and allows the user to select a cell and change the data. The ITableDataFlow defines the columns that are displayed. The Primary key column is not displayed by default. The user can select a row and column. Port 2 is an output, the current row Primary Key.
- Transact - Decorator of ITableDataFlow that does a data transfer, either left or right. Matches the column names and copies the data in the rows. If a column does not exist on the destination, then attempts to create that column. If columns have different compatible types (e.g. string with any other type) then tries to convert the data on a row by row basis. If columns have incompatible types (e.g. date and number) then doesn't transfer data, and generates an error. Matches rows by ID's. Error messages are output as rows on an Error Port which is a ITableDataFlow. Error messages can be for a whole columns, a whole row, or a specific cell.
- RowButton - Just a button that looks like a row of a grid.

- SessionDataSCP - Sends SCP commands to it connected serial port to get session data off of an SCP based device SCP commands are like FN, device responds with like [982000000123456, 123.5, 2018-11-23].
- LifeDataSCP - similar to SessionDataSCP except send a command to select life data first.
- SCPDeviceSense - sends ZN command which returns a device name like [3000].
- Iterator - Takes a ITableDataFlow on it RHS port. When a transaction occurs on this port, it starts iterating through the rows. It has a output called CurrentRow which is a ITableDataFlow with a single Row and the same columns as the input table. It generates a Transact operation. It has a output port called Started of type IEvent. It has an output port called Index of type IDataFlow of type number which outputs the index of the current row. It has an input port called Next or type IEvent that causes it to go to the next row of the input table. It has an output called Complete of type IEvent. It has an input called Stop or type IEvent, which can be used to stop the iterator before it finishes all the rows.
- Count - To get the number of the elements in a container.
- ConvertToEvent - Converts the activation event within the IUI interface to an IEvent. Converts a IDataFlow event to an IEvent.
- ConvertTableToDataFlow - Select a specific row of a table and pick the given column to get the cell data then output it as a string data flow.
- Equals - Outputs a true when the input equals the configured property value. Used frequently for recognizing connected device names and generates a true output that turns on many gates for that device. Use generics to implement? e.g. Equals<string>.

- Value - Can have multiple IDataFlow inputs and outputs. As implied by the way the IDataFlow interfaces works, stores the value when an event comes from an input, and emits events to the outputs. All the outputs can read the value at any time.

Appendix B

User Stories and Task Decomposition

B.1 The Selected User Stories For Comparative Experiments

We did not aim to select any specific user stories when we carried out the experiments. The product manager of Datamars has proposed more than 30 user stories, which were supposed to be added on the legacy Datalink. However, due to the low maintainability of it, these user stories were put off until a proper technical solution is found.

In this case, we merely selected the first six user stories for the experiments, which are listed as follows. Among them US1 and US2 were completely implemented, while the rest four ones were measured by estimations.

US1. When a new device is connected for the first time the desktop application will ask the user if they would like to automatically download sessions from this device.

US2. If the user chooses yes, the desktop application will automatically download any new sessions on the device and display them to the user. When the device is disconnected these will still be available in a sessions view.

US3. If the user chooses no, the desktop application will only download those sessions the user clicks on to view.

US4. The user can change this auto-download setting for a device at any time.

US5. Sessions that have been downloaded will automatically sync to MiHub Livestock can be saved to a custom location on the users' computer for emailing etc, and can be sent to national traceability programs, and can be deleted from the desktop application.

US6. When viewing sessions on a device the user can choose to delete sessions off the device - either some sessions, or all sessions. This makes it simple to delete many devices at once, rather than doing it one-by-one on the device UI.

B.2 Task Decomposition for Implemented User Stories in ALA Datalink

In this section, the way that the user stories were decomposed as tasks are explained. As the maintainer of the legacy Datalink aimed to implement the user stories as a whole, here we merely show the decomposition of the two implemented user stories in ALA Datalink, as illustrated in table B.1.

US1 was decomposed as three tasks, among which one new abstractions were required and it is generally a simple user story that advantages the ALA code base. US2 was decomposed as eight tasks, among which four new abstractions were required, and the new abstractions were not easy to implement. This story is considered to disadvantage the ALA Datalink since it has to construct four new abstractions and one new programming paradigm. Moreover, it changes the way the existing Iterator works, so some old code needs to change to fit this new design.

Story	Task	Description
US1	T1. new abstraction: DeviceIdSCP	T1. Get the unique ID of a connected device.
	T2. refine abstraction: Filter	T2. Refine existing abstraction to make them more reusable.
	T3. wiring	T3. wire abstractions to make them work
US2	T1. new interface: IIterator	T1. A new programming paradigm for iterating.
	T2. new abstraction: ListOfFiles	T2. List the files with a given folder name.
	T3. new abstraction: SelectExternal	T3. Dynamically select the fields and combine them with give parameters.
	T4. new abstraction: FileSessions	T4. Cache the session data read from the csv files.
	T5. new abstraction: ConvertIteratorToTable	T5. Convert iterator to table with a given column name.
	T6. refine abstraction: Select, CSVFileReaderWriter	T6. Make it able to read header information from a csv file
	T7. refine Iterator	T7. Change the way iterator works and re-wiring the existing logic that relates to Iterator
	T8. wiring	T8. Wire the new and existing abstraction to satisfy the user story

Table B.1: User Story Decomposition of ALA