Abstract

Software is contributing a substantial part of new functionality and innovations in safety-critical systems. These systems put a huge demand on software reliability, because a minor error can produce failure of a complete system. The evolution of software requires continuous development and maintenance. With size and complexity of safety-critical software growing as time goes, additional challenges arises, including implicit assumptions of *technical debt*. *Technical debt* refers to the compromises that are made in software development and maintenance in order to meet a short-term business goal. *Design debt* is an instance of technical debt. As software systems evolve, their design tend to decay over time, hence accumulating design debt. Consequently, software design becomes more difficult to maintain. Therefore, developers need to understand the reason for design debt accumulation so they can take proactive steps that may potentially reduce the debt in the future.

The main goal of this thesis is to empirically investigate design debt in safety-critical systems. The goal is reflected in our attempt to answer the following research questions:

- **RQ1:** How can design debt be identified?
- **RQ2:** What kind of design debt can be found in embedded systems?
- **RQ3:** What are the effects of design debt?
- **RQ4:** How to pay design debt?

A case study has been conducted in order to answer our research questions. The case study involves an analysis of a safety-critical system developed by Autronica Fire and Security AS. The system is written in C/C++. We used object-oriented metrics to identify classes that are most likely to pose problems for the system. Qualitative data were collected and analyzed using descriptive statistics. A set of thresholds for the metrics were derived to identify classes that have higher metric values than threshold values. In addition, automatic static analysis tools were applied in order to detect code smells.

This work contributes mainly to the improvement in software metrics and software quality. The main contributions of this work:

- **C1:** Empirical knowledge about identification of design debt in safety-critical systems by object-oriented metric analysis and code smell detection.
  - **C1.1:** A set of threshold values for the measured object-oriented metrics
- **C2:** Empirical knowledge about the different types of design debt in safety-critical systems
- **C3:** Empirical knowledge about the effects of having design debt in safety-critical systems

Sammendrag

Programvare spiller en

Preface

This thesis was written as a part of my MSc degree at Norwegian University of Science and Technology (NTNU). It was written in the spring of 2016 at the Department of Computer and Information Science (IDI) at NTNU.

This thesis is an extension of the work done in the specialization project - ”Managing Technical Debt in Embedded Systems” that was carried out by the same author in the fall of 2015. The supervisor for this project was Carl-Fredrik Sørensen.

Acknowledgements

This work has been supervised by Carl-Fredrik Sørensen at the Norwegian Univserity of Science and Technology (NTNU). I would like to express my sincere thanks to him for giving me the opportunity to work on this exciting project and for his invaluable guidance throughout the process. Without his time and valuable feedback, this work would not have been complete.

I would like to show my gratitude to Ingar Kulbrandstad from Autronica for providing the case study. I would also like to thank Øyvind Teig from Autronica for helpful proof-reading and grammar correction during the final phases of this thesis.

Lastly, I would like to thank my family and friends that have supported me all the way throughout my studies.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Report Overview

- ***1. Introduction***
- *2. State-of-the-Art*
- *3. Research Methodology*
- *4. Results*
- *5. Discussion*
- *6. Conclusion*

This chapter provides an introduction to this masters thesis. We begin with outlining the motivation for this research. Then a brief description of the research context and the research questions is presented. Lastly, we present the thesis outline.

1.1 Motivation

Successful embedded systems continuously evolve in response to external demands for new functionality and big fixes [1]. Evolution occurs as software changes over time. One consequence of such evolution is an increase of issues in design, development, and

maintainability [2]. Software code often ends up not contributing to the mission of the original intended software architecture or design.

The main challenge with software evolution is the technical debt that is not paid by the organization during software development and maintenance. Technical debt addresses the debt that software developers accumulate by taking shortcuts in development in order to meet the organizations business goals. For example, a deadline may lead developers to create ”non-optimal” solutions in order to deliver on time. As technical debt keeps accumulating, systems can become unmanageable and eventually unusable. Even more resources have to be spent during software maintenance on paying off the interest, i.e., the cost of having the debt. According to Gartner [3], the cost of dealing with technical debt threatens to grow to $1 trillion globally by 2015. That is the double of the amount of technical debt in 2010. Furthermore, many embedded systems are getting interconnected within existing Internet infrastructure, further known as the Internet of Things. Such devices are threatened by security issues. For example, Matthew Garret got access to the electronic equipment in every hotel room in a hotel located in London[1]. These equipments were connected to a network. Furthermore, two research discovered the possibility to start Tesla Model S using a laptop[2].

Several studies have classified the metaphor of technical debt into different types of debt that are associated with the different phases of software development [4–9]. Design debt is an instance of technical debt. Design debt accumulates when compromises are made in the software design. Software design plays a significant role in the development of large systems [10]. Unlike code-level debt, design debt usually has more significant consequences on software evolution by making the system more complex and harder to maintain over time [11, 12]. The quality of the software is very important, and without it, there will be more accidents and system failures. This is even more crucial for the case of safety-critical software, which failure can endanger human lives.

1.2 Research Context

This master thesis builds upon our previous study ”Managing Technical Debt in Embedded Systems” [13], a prestudy that was carried out in the fall of 2015. The written assignment for the specialization project had the following definition:

[1] The Internet of dangerous, broken things: http://www.zdnet.com/article/the-insecurity-of-the-internet-of-things/

[2] Researchers Hacked a Model S, But Tesla’s Already Released a Patch: http://www.wired.com/2015/08/researchers-hacked-model-s-teslas-already/

Managing Technical debt in embedded systems
”This task is related to management of software in embedded systems, as well as evolution of such software over time. Embedded systems have often a long lifetime and it is thus important to find out best practices and tools for this management since it is necessary to cope with architectural and design decisions which were made perharps decades ago, as well as clearly find out how present decisions may affect future maintenance and operation. This is called technical debt since all decisions will have a future cost related to them. Such decisions are often not documented, the people that made the software is not available 10-20 years after the implementation, the Internet of Things make all kind of embedded systems accessible from the Internet and thus posing security threats.

The project may take different directions based on the students interests and motivation. Industrial companies are very interested in this topic, so it is possible to study industrial systems, both past and current., make suggestions and implement them, make tools, make processes, make best practice etc.”

In our previous research, we did a pre-study of the field of technical debt in embedded systems, where we investigated the reasons for companies to incur technical debt and the different strategies for managing it. Data was collected by conducting semi-structured interviews. After completing the study, we had a desire to look into a more narrow field of the concept technical debt by conducting a deeper case study for our upcoming master thesis. An interest was to study an industrial system.

The work in our master thesis has been done in collaboration with Autronica Fire and Security AS, a global provider of safety solutions which includes fire safety equipment, marine safety monitoring, and surveillance equipment. Their main office is located in Trondheim, one of the largest cities in Norway. We have performed a deeper study on one of the company’s fire detection systems software for approximately six weeks. The study explored their source code in order to investigate design debt.

1.3 Research Design and Questions

Being able to identify design debt helps the software engineering field to get one step closer on solving the problems that are faced by the software industry today. The goal of the analysis is to investigate design debt in safety-critical systems. Design debt is a problem for many software projects today. Although design debt is recognized in many studies, there is lack of focus on design debt in safety-critical systems.

The relevant research methods in software engineering can be survey, design and creation, case study, experimentation, action research, and ethnography [14]. In this study, literature review and case study have been used to answer our research questions. Literature review was a part of the pre-study and has been used to get familiar with the term design debt and to define the research questions. Case studies are empirical methods used to investigate a single entity or phenomenon within a specific time space [15], which fits our desire to study an industrial system. Case studies can be both qualitative and quantitative [14, 16]. The research process we have chosen to adopt in this study follows the principles of the six steps defined by Soy [17]: *Determine and Define The Research Questions, Select the Cases and Determine Data Gathering and Analysis Techniques, Prepare to Collect Data, Data Collection, Evaluate and Analyze Data, and Prepare the Report*.

The main research questions investigated in this thesis are:

- **RQ1**: How can design debt be identified?
- **RQ2**: What kind of design debt can be found in embedded systems?
- **RQ3**: What are the effects of design debt?
- **RQ4**: How to pay design debt?

1.4 Contrubution

This work contributes mainly to the improvement in software metrics and software quality. The main contributions of this work are:

- **C1:** Empirical knowledge about identification of design debt in safety-critical systems by object-oriented metric analysis and code smell detection.
  - **C1.1:** A set of threshold values for the measured object-oriented metrics
- **C2:** Empirical knowledge about the different types of design debt in safety-critical systems
- **C3:** Empirical knowledge about the effects of having design debt in safety-critical systems

1.5 Thesis Structure

The thesis is structured into several chapters with sections and subsections. The outline of the thesis is as follows:

- **Chapter 1**: Introduction contains a brief and general introduction to the study and the motivation behind it.
- **Chapter 2**: State-of-the-Art looks at important aspects of the research question.
- **Chapter 3**: Research Method describes how the literature review was carried out throughout the research, as well as a description of the case study to be performed.
- **Chapter 4**: Results presents the results from the case study, and takes a closer look at the findings from the case study.
- **Chapter 5**: Discussion contains a summarized look at the findings from the case study, and connects it with the literature review and to the research questions. An evaluation of the research is also given in this chapter.
- **Chapter 6**: Conclusion concludes the research by providing a summary of the most important points of the results and discussion chapter. Additionally, in outlines possible routes to take in the research field.

CHAPTER 2

STATE-OF-THE-ART

Report Overview

- *1. Introduction*
- ***2. State-of-the-Art***
- *3. Research Methodology*
- *4. Results*
- *5. Discussion*
- *6. Conclusion*

This chapter presents the relevant state-of-the-art for this thesis. Section 2.1 presents the metaphor of technical debt.

2.1 Technical Debt

The metaphor of technical debt was first introduced by Ward Cunningham in 1992 to communicate technical problems with non-technical stakeholders [18]. To deliver business functionality as quick as possible, *'quick and dirty'* decisions are often made. These

decisions may have short-term value, but it could affect future development and maintenance activities negatively. Cunningham was the first one who drew the comparison between technical complexity and financial debt in a 1992 experience report [18]:

> *"Shipping first time code is like going into debt. A little debt speeds up the development as long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise." - Ward Cunningham, 1992.*

The concept refers to the financial world where going into debt means repaying the loan with interest [19]. Like financial debt, technical debt accrues interest over time. Interest is defined as the extra effort that has to be dedicated in the future development in order to modify the part of the software that contains technical debt [4, 20, 21]. Unmanaged technical debt can cause projects to face significant technical and financial problems, which ultimately leads to increased maintenance and evolution costs [22].

2.1.1 Definitions of Technical Debt

Several researchers have attempted to give us a clear picture of what technical debt is [10, 23, 24]. Fowler [23] presents a technical debt quadrant which consists of two dimensions: *reckless/prudent* and *deliberate/inadvertent* [23]. Technical debt quadrant in Figure 2.1 indicates four types of technical debt: *reckless/deliberate*, *reckless/inadvertent*, *prudent/deliberate*, and *prudent/inadvertent*. Reckless/Deliberate debt is usually incurred when technical decisions are taken intentionally without any plans on how to address the problem in the future. A team may know about good design practices, but still implements *'quick and dirty'* solutions because they think they cannot afford the time required to write clean code. The second type is reckless/inadvertent. It is incurred when best practices for code and design are being ignored, ultimately leading to a big mess of spaghetti code. Prudent/Deliberate debt occurs when the value of implementing a 'quick and dirty' solution is worth the cost of incurring the debt to meet a short-term goal. The team is fully aware of the consequences, and have a plan on how to address the problem in the future. At last, we have prudent/inadvertent debt. This type of debt occurs when a team realizes that the design of a valuable software could have been better after delivering it. A software development process is as much as learning as it is coding.

McConnell [24] classified technical debt as intentional and unintentional debt. Intentional debt is described as debt that is incurred deliberately. For example, an organization

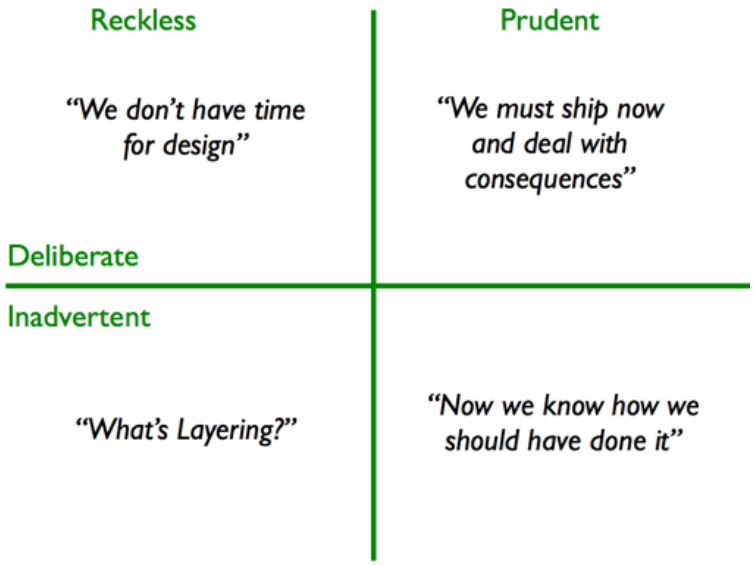

Figure 2.1: Fowler's Technical Debt Quadrant

makes a strategic decision that aims to reach a certain objective by taking a shortcut they are fully aware of. Intentional debt can further be viewed as short-term and long-term debt [5, 25]. Short-term debt is usually incurred reactively, for tactical reasons. Long-term debt is usually incurred pro-actively, for strategic reasons. Unintentional debt is described as debt that is incurred inadvertently due to lack of knowledge or experience. For example, a junior software developer may write low quality code that does not conform with standard coding standard due to low experience.

Krutchen et al. [10] presented a technical debt landscape for organizing technical debt. They distinguished visible elements such as new functionality to add or defects to fix, and the invisible elements that are only visible to software developers. On the left side of Figure 2.2, technical debt affects evolvability of the system, while on the right side, technical debt mainly affects software maintainability.

2.1.2 Classification of Technical Debt

Technical debt can accumulate in many different ways, and therefore it is important to distinguish the various types of technical debt. Multiple studies [4–9] have pointed out several subcategories of technical debt based on its association with traditional software life-cycle phases; architectural debt, code debt, defect debt, design debt, documentation debt, infrastructure debt, requirements debt, and test debt. Table 2.1 lists the different

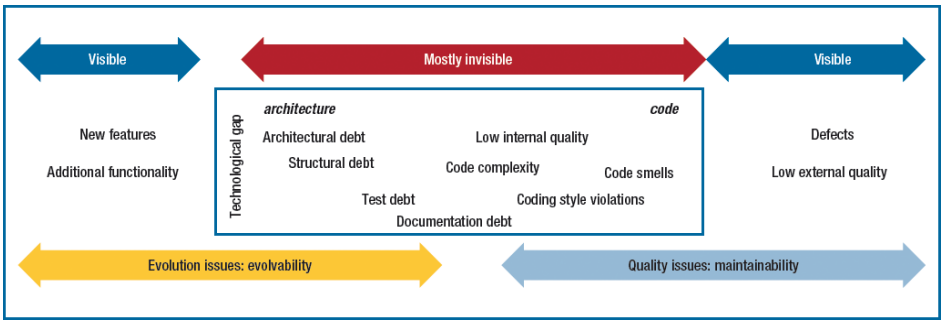

Figure 2.2: Technical Debt Landscape

Table 2.1: Types of Technical Debt

Subcategory	**Definition**
Architectural debt [4–6]	Architectural decisions that make compromises in some of the quality attributes, such as modifiability.
Code debt [4, 6, 7]	Poorly written code that violates best coding practices and guidelines, such as code duplication.
Defect debt [4, 7]	Defect, failures, or bugs in the software.
Design debt [4, 6, 8]	**Technical shortcuts that are taken in design**.
Documentation debt [4, 6, 9]	Refers to insufficient, incomplete, or outdated documentation in any aspect of software development.
Infrastructure debt [4, 5, 7]	Refers to sub-optimal configuration of development-related processes, technologies, and supporting tools. An example is lack of continuous integration.
Requirements debt [4, 9]	Refers to the requirements that are not fully implemented, or the distance between actual requirements and implemented requirements.
Test debt [4, 6, 9]	Refers to shortcuts taken in testing. An example is lack of unit tests, and integration tests.

subcategories of technical debt.

2.1.3 Causes and Effects of Technical Debt

Several researchers have investigated the reasons to incur technical debt. Klinger et al. [21] conducted an industrial case study at IBM where four technical architects with different backgrounds were interviewed. The goal was to examine how decisions to incur debt were taken, and the extent to which the debt provided leverage [21]. The study revealed that the company failed to assess the impact of intentionally incurring debt on projects. Decisions regarding technical debt were rarely quantified. The study also revealed big organizational gaps among the business, operational, and technical stakehold-

ers. When the project team felt pressure from the different stakeholders, technical debt decisions were made without quantifications of possible impacts.

Lim et al. [26] pointed out that technical debt is not always the result of poor developer disciplines, or sloppy programming. It can also include intentional decisions to trade off competing concerns during business pressure. Furthermore, Li et al. explains that technical debt can be used in short term to capture market share and to collect customers feedback early. In the long term, technical debt tended to be negative. These trade-offs included increased complexity, reduced performance, low maintainability, and fragile code. This led to bad customer satisfaction and extra working hours. In many cases, the short term benefits of technical debt outweighed the future costs.

Guo et al. [27] studied the effects of technical debt by tracking a single delayed maintenance task in a real software project throughout its life-cycle, and simulated how managing technical debt can impact the project result. The results indicated that delaying the maintenance task would have almost tripled the costs, if it had been done later.

Siebra et al. [28] carried out an industrial case study where they analyzed documents, emails, and code files. Additionally, they interviewed multiple developers and project managers. The case study revealed that technical debt were mainly taken by strategic decisions. Furthermore, they commented out that using a unique specialist could lead the development team to solutions that the specialist wanted and believe were correct, leading the team to incur debt. The study also identified that technical debt can both increase and decrease the amount of working hours.

Zazworka et al. [29] studied the effects of *God Class* code smell and technical debt on software quality. *God Class* are examples on bad coding, and therefore includes a possibility for refactoring [8]. The results indicated that *God Class* code smell requires more maintenance effort including bug fixing and changes to software that are considered as a cost to software project. In other words, if developers desire higher software quality, then technical debt needs to be addressed closely in the development process.

Buschmann [30] explained three different stories of technical debt effects. In the first case, technical debt accumulated in a platform started had growth to a point where development, testing, and maintenance costs started to increase dramatically. Additionally, the components were hardly usable. In the second case, developers started to use shortcuts to increase the development speed. This resulted in significant performance issues because an improper software modularization reflected organizational structures instead of the system domains. It ended up turning into economic consequences. In the last case, an existing software product experienced increased maintenance cost due to architectural erosion. However, management analyzed that re-engineering the whole software would

cost more than doing nothing. Management decided not to do anything to technical debt, because it was cheaper from a business point-of-view.

Codabux et al. [5] carried out an industrial case study where the topic was agile development focusing on technical debt. They observed and interviewed developers to understand how technical debt is characterized, addressed, prioritized, and how decisions led to technical debt. Two subcategories of technical debt were commonly described in this case study; infrastructure and automation debt.

These studies indicates that the causes and effects of technical debt are not always caused by technical reasons. Technical debt can be the result of intentional decisions made by the different stakeholders. Incurring technical debt may have short-term positive effects such as time-to-market benefits. Not paying down technical debt can result economic consequences, or quality issues in the long-run. The allowance of technical debt can facilitate product development for a period, but decreases the product maintainability in the long-term. However, there are some times where short-term benefits overweight long-term costs [27].

2.1.4 Identification of Technical Debt

Technical debt accumulation may cause increased maintenance and evolution costs. At worst, it may even cancel out projects. The first step towards managing technical debt is to properly identify and visualize technical debt items.

According to Zazworka et al. [31], there are four main techniques for identifying technical debt in source code: modularity violations, design patterns and grime buildup, code smells, and automatic static analysis issues.

Modularity Violation

Software modularity determines software quality in terms of evolveability, changeability, and maintainability [32], and the essence is to allow modules to evolve independently. However, in reality, two software components may change together though belonging to distinct modules, due to unwanted side effects caused by *'quick and dirty'* solutions [31, 33]. This causes a violation in the software designed modular structure, which is called a modularity violation. Wong et al. [33] identified 231 modularity violations from 490 modification requests in their experiment using Hadoop. 152 of the 490 identified violations were confirmed by the fact that they were either addressed in later versions of Hadoop, or recognized as problems by the developers. In addition, they identified 399

modularity violation from 3458 modification request of Eclipse JDT [33]. Among these violations, 161 were confirmed. Zazworka et al. [31] revealed that the average number of modularity violations per class in release 0.2.0 to release 0.14.0 of Hadoop ranged from 0.04 to 0.11. They identified 8 modularity violations in the first release of Hadoop and 37 in the last one. In addition, they revealed that modularity violations are strongly related to classes with high defect- and change-proneness.

Design Pattern and Grime Buildup

Patterns are known to be general solutions to recurrent design problems. They are commonly used to improve maintainability, flexibility, and architecture design of software systems by reducing the number of defects and faults. Twenty-three design patterns are widely used in software development and is classified into three types [34]: *creational*, *behavioral*, and *structural*. Creational design patterns are all about class instantiation. This pattern can be divided into class-creation patterns and object-creational patterns. Creational patterns include patterns such as Abstract Factory, Builder, Factory Method, Object Pool, Prototype, and Singleton [34]. Structural design patterns are all about Class and Object composition. They include patterns such as Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy [34]. Lastly, behavioral design patterns are all about class's objects communication, and are concerned with communication between objects. Behavioral patterns include patterns such as Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template Method, and Visitor [34].

Software continuously evolve in response to external demands for new functionality. One consequence of such evolution is software design decay. Izurieta et al. [12] defines decay the deterioration of the internal structure of system designs. Furthermore, they define design pattern decay as deterioration of the structural integrity of a design pattern realization. As a pattern realization evolves, its structure and behavior tend to deviate from its original intent. Changes in the code base could lead to code ending up outside the pattern. This is known as design grime, non-pattern related code [12]. Moreover, design patterns can also rot, when changes break the structural and functional integrity of a pattern [12]. Izurieta et al. [35] examined the extent to which software designs actually decay, rot, and accumulate grime in three open-source systems. They found no evidence of design patterns rot, but they found evidence of pattern decay due to grime. Furthermore, Izurieta et al. [36] also studied the consequences of grime buildup on testability of general purpose design patterns. Their result reveals that as systems age, the growth of grime and the appearance of anti-patterns increase testing requirements.

Code Smell

Some forms of technical debt accumulate over time in the form of source code [31]. Fowler et al. [37] describes the concept of code smells as choices in object-oriented systems that do not comply with the principles of good object-oriented design and programming practices. They are an indication of that some parts of the design is inappropriate and that it can be improved. Code smells are usually removed by performing one or more refactoring [37]. For instance, one such smell is ”Long Method”, a method with too many lines of code. This type of code smell can be refactored by ’Extract Method’, by reducing the length of the method body [37].

Mäntylä et al. [38] proposes a taxonomy based on the criteria on code smells defined by Fowler et al. [37]. The taxonomy categories code smells into seven groups of problems: bloaters, object-oriented abusers, change preventers, dispensables, encapsulators, couplers, and others. The first class, Bloaters, represents large pieces of code that cannot be effectively handled. Object-oriented abusers are related to cases where the solution does not exploit the the possibilities of object-oriented design. Change preventers refers to code structure that considerably hinder the modification of software. Dispensables represent code structure with no value. Encapsulators deal with data communication mechanism or encapsulation. Couplers refer to classes with high coupling. The last group of problem is Other, which refers to code smells that do not fit into any of the other categories. This includes *Incomplete Library Class* and *Comments*. Table 2.2 lists all the code smells that are presented by Fowler et al. [37].

Several studies have been conducted to investigate the relationship between code smell and change-proneness of classes in object-oriented source code. A study by Olbrich et al. [39] revealed that different phases during evolution of code smells could be identified, and classes infected with code smells have a higher change frequency; such classes seem to need more maintenance than non-infected classes. Khomh et al. [40] investigate if classes with code smells are more change-prone than classes without smells. After studying 9 releases of Azureus and 13 releases of Eclispe, their findings show that classes with code smells are more change-prone than others.

Multiple approaches have been proposed for identifying code smells, ranging from manual approaches to automatic. Manual detection of code smells can be done by code inspections [41]. Travassos et al. [41] present a set of reading techniques that gives specific and practical guidance for identifying defects in Object-Oriented design. However, Marinescu [42] argue that manual code inspection can be time expensive, unrepeatable, and non-scalable. In addition, it is often unclear what exactly to search for when inspecting code [43]. Moreover, a study by Mäntylä revealed more issues regarding manual

Table 2.2: Code Smell Taxonomy

Code Smell	**Group**
Long Method	Bloaters
Large Class	Bloaters
Primitive Obsession	Bloaters
Long Parameter List	Bloaters
Data Clumps	Bloaters
Switch Statements	O-O Abusers
Temporary Field	O-O Abusers
Refused Bequest	O-O Abusers
Alternative Classes with Different Interfaces	O-O Abusers
Parallel Inheritance Hierarchies	O-O Abusers
Divergent Change	Change Preventers
Shotgun Surgery	Change Preventers
Lazy Class	Dispensables
Data Class	Dispensables
Duplicated Code	Dispensables
Speculative Generality	Dispensables
Message Chains	Encapsulators
Middle Man	Encapsulators
Feature Envy	Couplers
Inappropriate Intimacy	Couplers
Comments	Other
Incomplete Library Class	Other

inspection of code. He states that manual code inspection is hard due to conflicting perceptions of code smells among the developers, causing a lack of uniformity in the smell evaluation.

Automatic approaches for identifying code smells reduce the effort of browsing through large amounts of code during code inspection process. Ciupke [43] propose an approach for detecting code smells in object-oriented systems. In this approach, code smells to be identified are specified as queries. The result of a queries is a piece of design specifying the location of the code smell in the source code. This approach was applied to several case studies, both in academical and industrial context. Their findings revealed that code smell detection can be automated to a large degree, and that the technique can be effectively applied to real-world code.

Another method for automatic detection of code smells is done by using metrics. Marinescu [42] propose a general metric-based approach to identify code smells. Instead of a purely manual approach, the use code metrics were proposed for detecting design flaws in object-oriented systems. This approach were later refined, with the introduction of detection strategies [44]. Based on their case study, the precision of automatic detection of code smells is reported to be 70%. Furthermore, a study by Schumacher et al. [45] investigated how human elicitation of technical debt by detecting god class code smells compares to automatic approaches by using a detection strategy for god classes. Their findings show that humans are able to detect code smells in an effective way if provided with a suitable process. Moreover, the the findings revealed that the automatic approach yield high recall and precision in this context.

Automatic Static Analysis Issues

The identification of software design and code issues can be done with automatic static analysis code tools. Automatic static analysis tools looks for violations of recommend programming practices on source code line level that might cause faults or degrade some parts of software quality [31]. Tools are able to alert software developers of potential problems in the source code, and they may suggest refactoring solutions to avoid future problems. Several tools for detecting code and design issues have been proposed in the literature [46]. These tools have been applied by several researchers [31, 46–48], and the overall finding is that a small set of automatic static analysis issues are related to defects in the software. However, the set of issues depends on the context and type of software analyzed. Despite the fact that a tool indicates a potential problem in the source code, it takes human judgment to determine if something could be problematic down the road.

2.1.5 Strategies and Practices for Managing Technical Debt

Managing technical debt compromises the actions of identifying the debt and making decisions about which debt that should be repaid [8, 10, 37]. There are three high-level steps that are required to manage technical debt [49]: *Increasing awareness of technical debt*, *detecting and repaying technical debt*, and *prevent accumulation of technical debt*. *Increasing awareness of technical debt* is the first step towards technical debt management. This includes awareness of the concept of technical debt, its different forms, the impact of technical debt, and the factors that contribute to technical debt. Awareness will help an organization taking the right decisions to achieve their goals. The second step is *detecting and repaying technical debt*. This step is focused on determining the extent of technical debt in the software. Identifying specific instances of debt and their impact can help us to prepare a plan to recover from the debt. The final step, *prevent accumulation of technical debt*, ensures that technical debt does not increase and remains manageable in the future. Best practices such as refactoring, re-engineering, and testing is necessary to manage the debt [5]. This step also involves collaboration between all stakeholders to collectively track and monitor the debt. For example, using a backlog during release and iteration planning to list debt-related tasks [10]. Guo et al. [20] suggest using a portfolio for technical debt management. This approach collects technical debt items to a *"Technical Debt List"*, which may potentially be used to pay technical debt based on its cost and value. Tools may also be used for technical debt management. SonarQube is an open source application for quality management [50]. It can assess the debt in a software project by performing automatic static code analysis. Each automatic static analysis issue is assigned a score based on how much work it requires to fix that error. The analysis gives the total sum of technical debt for the entire product.

2.2 Design Debt

Design debt is one of the instances of technical debt [4, 6, 8]. It is concerned with the design aspects of technical debt. Design debt includes design smells and violations of design rules. Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality [49]. A possible symptom of design debt is when code structures drift away from good object-oriented design principles [29]. There are various reasons for software projects to run into design debt. For example, a common object-oriented design principle tells that a class should have a single purpose and should not implement many functions of the system. A class that tries to accomplish too many purposes is known as *God Class* code smell. As

we have mentioned earlier, code smells are an example design flaws in object-oriented design, which may potentially lead to maintainability issues in future evolution of the software system [39].

Suryanarayana et al. [49] present six common causes of design debt: *violation of design principles*, *inappropriate use of patterns*, *language limitations*, *procedural thinking in object-oriented paradigm*, *viscosity*, and *non-adherence to best practices and processes*. Design principles provide guidance to designers in creating high quality software. *Violation of design principles* are manifested as smells. For example, code smells are symptoms of design violations in the source code. *Inappropriate use of patterns* is related to the use of patterns without fully understanding the context, and the finer aspects and implications of using a particular variation of a pattern which. In some cases, misuse of a design pattern may potentially lead to an antipattern. *Language limitations* is related to deficiencies in programming languages. *Procedural thinking in object-oriented paradigm* is related to using procedural programming techniques in object-oriented paradigm. For example, using imperative names for classes, functional decomposition, and missing polymorphism with explicit type checks results in design smells in an object-oriented context. *Viscosity* is one of the reasons to use hacks instead of adopting a systematic process to achieve a particular requirement. Lastly, *Non-adherence to practices and processes* is related to best practices and processes that are not followed correctly or completely.

2.3 Software Quality

According to the IEEE Standard Glossary of Software Engineering Terminology [51], the quality of a software is defined as *1) the degree to which a system, component, or process meets specified requirements*, and *2) the degree to which a system, component, or process meets customer or user needs or expectations*. ISO/IEC 9126:2001 offer a valuable conceptual framework for software quality, where a distinction between *quality in use*, *external quality*, and *internal quality* is made [52]. *Quality in use* refers to the users view of system quality. *External quality* reflects the dynamic aspect of a software application, and is subdivided into six quality attributes. *Internal quality* refers to the criteria of each quality attribute. Software quality can be measured by using a product quality model. ISO/IEC 9126:2001 classifies software quality in a structured set of six quality attributes [52]. Bass et al. [53] describes these characteristics as software quality attributes. Table 2.3 present the ISO/IEC 9126:2001 quality model, with the quality attributes, and their criteria.

2.4 Object-Oriented Metrics

Object-Oriented metrics have been proposed as a quality indicator for object-oriented software systems. There are three traditional metrics that are widely used, and are well understood by researchers and practitioners [54]. These metrics are: *cyclomatic complexity*, *size*, and *comment percentage*. *Cyclomatic complexity* evaluates the complexity of an algorithm in a method. It is recommended that the *cyclomatic complexity* for a method should be below 10 [54]. *Size* of a method is used evaluate the understandability of the code. *Size* is measured in many different ways, including all physical lines of code, lines of statements, and number of blank lines. *Comment percentage* measures the number of comments in percent by counting total number of comments divided on total lines of code minus number of blank lines.

Chidamber and Kemerer [55] proposed a set of six software metrics to identify certain design traits of a software component. These metrics are: *Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Lack of Cohesion in Methods (LCOM), Coupling Between Objects (CBO), and Response For a Class (RFC)*. The *WMC* is used to count the number of methods in a class, or to count the of sum of complexities of all methods in a class. The complexity of a method is measured by *cyclomatic complexity*. This metric measures understandability, maintainability, and reusability [54]. The *DIT* metric measure the maximum number of steps from a class node to the root in the inheritance hierarchy. The deeper a class is withing the hierarchy, the greater number of methods it is likely to inherit, making it more complex to predict its behavior [54]. Moreover, this metric are related to efficiency, reusability, understandability, and testability [54]. *NOC* metric measures the number of subclasses of a class in a hierarchy. The greater number of children may be an indication of subclasses misuse or improper parent abstraction. This metric evaluates efficiency, reusability, and testability [54]. The *LCOM* is used to measure the lack of cohesion in methods of a class. It measures the dissimilarity of methods in a class by looking at the instance variables or attributes used by the methods. High cohesion indicates good subdivision, while low cohesion increase the complexity of a class. This metric measures efficiency and reusability [54]. The *CBO* metric counts the number of other classes to which a class is coupled. Excessive coupling is detrimental to modular design and prevents reuse [54]. Larger number of coupled objects indicates higher sensitivity to changes in other parts of the design. *CBO* evaluates efficiency and reusability [54]. The *RFC* metric counts the total number of methods in a class that can be invoked in a response to a message sent to an object. This metric includes all methods accessible within the class hierarchy. *RFC* metric evaluates understandability, maintainability, and testability [54]. Basili et

al. [56] investigated Chidamber and Kemerer's suite of metrics. Their result claim that several of Chidamber and Kemerer's object-oriented metrics appear to be useful to predict fault-prone classes during early phases of the software life-cycle. Similarly, Li et al. [57] conclude that object-oriented metrics are able to predict maintenance effort.

2.5 Software Evolution and Maintenance

Increasingly, more and more software developers are employed to maintain and evolve existing systems instead of developing new systems from scratch [58]. Lehman [59] introduced the study of software evolution. Software evolution is a process that usually takes place when the initial development of a software project is done and was successful [60]. The goal of software evolution is to incorporate new user requirements in the application, and adapt it to the existing application. Software evolution is important because it takes up to 85-90% of organizational software costs [58]. In addition, software evolution is important because technology tend to change rapidly.

Software maintenance is defined as *modifications of a software after delivery to correct faults, ti improve performance or other attributes, or to adapt the product to a modified environment* [61]. Maintenance can be classified into four types [60,61]: *adaptive maintenance*, *perfective maintenance*, *corrective maintenance*, and *preventive maintenance*. *Adaptive maintenance* is the modification of a software product performed after delivery to keep the computer program usable in a changed or changing environment. *Perfective maintenance* is the modification of a software product after delivery to improve performance and maintainability. *Corrective maintenance* is the reactive modification of a software product performed after delivery to correct discovered faults. Lastly, *Preventive maintenance* is the maintenance performed for the purpose of preventing problems before they occur.

Van Vliet [62] states that the real maintenance activity is corrective maintenance. 50% of the total software maintenance is spent on perfective, 25% on adaptive maintenance, and 4% on preventive maintenance. This leads to that 21% of the total maintenance activity is corrective maintenance, the 'real' maintenance [62]. This has not changed since the 1980s when Lientz and and Swanson conducted a study on software maintenance [63]. The study points out that most severe maintenance problems were caused by poor documentation, demands from users for changes, poor meeting scheduled, and problems training new hires.

2.6 Software Reuse

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch [64,65]. There are many various techniques that can be used to achieve software reuse: *abstraction*, *compositional reuse*, *generative reuse*, and *generation vs. composition* [66]. *Compositional reuse* is based on the idea of reusable components. It supports bottom-up development of systems where low-level components are available. *Compositional reuse* is also known as ad-hoc reuse or individual reuse. *Generative reuse* is based on the reuse of a generation process. It is often domain-specific, and is concerned with artifacts such as requirements, designs, and subsystems [67]. *Generative reuse* is also known as systematic reuse [67]. *Generation vs. composition* is a combined approach.

There are several ”assets” from a software project that can be reused, including *architectures, source code, data, designs, documentation, estimates, human interfaces, plans, requirements*, and *test cases* [65].

The most common benefits of software reuse are quality improvement and effort reduction. Software reuse includes lower costs, shorter development time, higher quality of reusable components, and higher productivity [68, 69]. The quality of the software increases every time an asset is reused, because errors are discovered more frequently, thus making it easier to keep the artifact more stable [66]. However, software reuse may cause problems as well. A case study on selected feature from self-driving miniature car development revealed that reuse of legacy, third party, or open-source code, was one of the root causes of technical debt accumulation [70]. Furthermore, Morisio et al. [71] identified three main causes of failures associated with software reuse: *not introducing reuse-specific processes*, *not modifying non-reuse processes*, and *not considering human factors*. These causes were combined with lack of commitment by top management. Organizations attempting to implement generative reuse face both technical and non-technical problems [72]. Technical problems includes challenges with tools, standards and technology. Non-technical problems include the human factor, cultural issues, economical issues, and organizational issues.

2.7 Refactoring

Fowler defines refactoring as a process of changing the software system in such way that it does not alter the external behavior of the code yet improves its internal structure [37]. Refactoring increases the code readability and maintainability by cleaning up code in a

way such that changes of defects is reduced [37]. Refactoring is an act of improving the design and quality of an existing system [62]. It is believed that refactoring improves software quality and developer productivity by making it easier to understand and maintain software source code [73]. According to Opdyke [74], refactoring can be categorized into low-level and high-level refactoring. Low-level refactoring is related to changing a program entity, such as renaming a member variable. High-level refactoring is usually a sequence of low-level refactorings, such as defining an abstract superclass. In addition, Opdyke presents twenty-six low-level refactorings and three high-level refactorings. Fowler et al. [37] have summarized the work of Opdyke by cataloging his refactorings along with many other refactoring techniques that have been collected over the years. Refactoring can be applied to three types of software artifacts: *programs, designs*, and *software requirements*. *Programs* include refactoring at source code or program level. For example, *Extract Method* [37] is a refactoring technique that can be applied at this level. *Designs* include refactoring at design level, for example in the form of UML models. Design pattern is an example of an artifact at this level. Lastly, *software requirements* include refactoring at the level of requirements specification. For example, decomposing requirements into a structure of viewpoints.

2.8 Embedded Systems

The *IEEE Standard Glossary of Software Engineering Terminologies* define embedded system as *a computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in aircraft or rapid transit system* [51]. Embedded systems are special-purpose computer systems designed to perform certain dedicated functions under certain constraints. According to Koopman [75], there are four types of embedded systems: *general computing*, *control systems*, *signal processing*, and *communication and networking*. The various types of embedded systems share common requirements such as: *real-time requirements*, *resource consumption*, *dependability*, and *life-cycle properties* [76]. Embedded systems are supposed to be failure-free [77], but these requirements may hinder embedded systems to deliver reliable service given a disturbance to its services [78].

Software constitutes only one part in embedded systems. Embedded software is defined as software part of a larger system which performs some of the requirements of that system [51]. Developing embedded software has proven to be difficult [79, 80]. Embedded software not only have to meet the functional requirements of ”what”, but also the non-functionality or ”quality” requirements expected of them [80]. Non-functional

requirements have come to be referred to as software quality attributes. Ebert et al. [79] state that quality of software in embedded systems is difficult to measure. Most behaviors for embedded systems are related to run-time quality attributes, depending on the system domain. For instance, reliability and safety qualities are important in the automotive domain [81], while performance and reliability is important in mission critical systems [82]. Other quality attributes such as maintainability and usability, are often compromised in favor of run-time quality attributes.

2.8.1 Safety-Critical Systems

Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment [83]. These systems put a huge demand on software reliability, because a minor error in safety-critical software can produce failure of a complete system [79, 81, 82]. Consider the role of safety-critical systems is often critical, the importance of managing software quality is therefore necessary to deliver software in a useful, safe, and reliable way. Given the lack of focus on non-functional requirements, the amount of technical debt in safety-critical systems grow continuously. Insufficient requirements and testing are two major cost drivers in embedded software development [1, 79]. 40% of all software defects results from insufficient requirements, especially non-functional requirements[6,7,45] [1, 79, 84]. Testing code consumes up to 40% of development resources. In addition, testing may potentially require 15-50% of total project duration [79].

Table 2.3: Software quality attributes, criteria, and description (ISO/IEC 9126)

Quality Attribute	**Criteria**	**Description**
Functionality		Systems ability to do work for which it was intended.
	Suitability	
	Accuracy	
	Interoperability	
	Security	
	Compliance	
Reliability		Systems ability to keep operating over time under certain conditions.
	Maturity	
	Fault Tolerance	
	Recoverability	
	Compliance	
Usability		Systems ability to be understood, learned, and used by others.
	Understandability	
	Learnability	
	Operability	
	Attractiveness	
	Compliance	
Efficiency		Systems ability to provide appropriate performance relative to the amount of resources used under stated conditions.
	Time behaviour	
	Resource utilization	
	Compliance	
Maintainability		Systems ability to be modified in the future.
	Analyzeability	
	Changeability	
	Stability	
	Testability	
	Compliance	
Portability		Systems ability to be transferred from one environment to another.
	Adaptability	
	Installability	
	Co-existence	
	Replaceability	
	Compliance	

CHAPTER 3

RESEARCH METHODOLOGY

Report Overview

- *1. Introduction*
- *2. State-of-the-Art*
- ***3. Research Methodology***
- *4. Results*
- *5. Discussion*
- *6. Conclusion*

The nature of this thesis makes it suitable as an empirical research. To answer the research questions that was stated in Chapter 1, Section 1.3, an empirical research needs to be carried out in order to collect some data. This chapter provides a brief introduction to research methods in software engineering, and describes the research conducted in the thesis. Section 3.1 describes the relevant research methods in software engineering. Section 3.2 describes the research method that was chosen for this study. Section 3.4 presents the research process we have followed throughout this thesis, which includes our research design.

3.1 Research Methods in Software Engineering

This section presents relevant research methods applied in software engineering research.

Research is believed to be the most effective way of coming to know what is happening in the world [16]. Empirical software engineering is a field of research based on empirical studies to derive knowledge from an actual experience rather than from theory or belief [85]. Empirical studies can be explanatory, descriptive, or exploratory [15].

There are two types of research paradigms that have different approaches to empirical studies [15]; the qualitative, and the quantitative paradigm. Qualitative research is concerned with studying objects in their natural setting [15]. It is based on non-numeric data found in sources as interview tapes, documents, or developers' model. Quantitative research is concerned with quantifying a relationship or to compare two or more groups [15]. It is based on collecting numerical data.

To perform research in software, it is useful to understand the different research strategies that are available in software engineering. Oates [14] presents six different research strategies; survey, design and creation, case study, experimentation, action research, and ethnography.

Survey focuses on collection data from a sample of individuals through their responses to questions. The primary means of gathering qualitative or quantitative data are interviews or questionnaires. The results are then analyzed using patterns to derive descriptive, exploratory, and explanatory conclusions.

Design and creation focuses on developing new IT products, or artifacts. It can be a computer-based system, new model, or a new method.

Case study focuses on monitoring one single 'thing'; an organization, a project, an information system, or a software developer. The goal is to obtain rich, and detailed data.

Experimentation are normally done in laboratory environment, which provides a high level of control. The goal is to investigate cause and effect relationships, testing hypotheses, and to prove or disprove the link between a factor and an observed outcome.

Action research focuses on solving a real-world problem while reflecting on the learning outcomes.

Ethnography is used to understand culture and ways of seeing of a particular group of people. The researcher spends time in the field by participating rather than observing.

3.2 Choice of Research Method in this Thesis

The main purpose of this research project is to gain an understanding about the nature of technical debt in software design and architecture, and its potential sources in embedded systems in order to improve the management of software evolution. Based on the research questions stated in Chapter 1, we applied the case study approach to our research. Case studies provide both quantitative and qualitative information about the system [14], depending on the approach the case study is taking.

3.2.1 Case Study Method

Case study is an empirical method to investigate a single phenomenon within a specific time space in real-life context [15]. Case studies excels at bringing an understanding of why or how certain phenomena occur or to add strength to what is already known through previous research [15, 17]. Runeson et al. [86] suggests case study as the most appropriate research method to use when exploring how a problem behaves in a real life context. In addition, they conclude that case study is suitable for software engineering research. There has been suggested systematic approaches for organizing and conducting a research successfully [17,86]. According to Yin [87], a research design is an action plan from getting here to there, where here is defined as the initial set of questions answered, and there is some set of conclusions about these questions. Moreover, a research design can be seen as a blueprint of research, dealing with at least four problems: what questions to study, what data are relevant, what data to collect, and how to analyze the results [87]. Soy [17] proposes six steps that can be used when carrying out a case study:

1. *Determine and Define the Research Questions*: The first step involves establishing a research focus by forming questions about the problem to the studied. The researcher can refer to the research focus and questions over the course of study.

2. *Select the Cases and Determine Data Gathering and Analysis Techniques*: The second step involves determining what approaches to use in selecting single or multiple real-life cases cases to examine, and which instruments and data gathering approaches to use. (whom we want to study, the case, cases, sample. and how we want to study it, design).

3. *Prepare to Collect Data*: The third step involves a systematic organization of the data to be analyzed. This is to prevent the researcher from being overwhelmed by the amount of data and to prevent the researcher from losing sight of the research focus and questions.

4. *Collect Data in the Field*: This step involves collecting, categorizing, and storing multiple sources of data systematically so it can be referenced and sorted. This makes the data readily available for subsequent reinterpretation.

5. *Evaluate and Analyze the Data*: The fifth step involves examining the raw data in order to find any connections between the research object and the outcomes with reference to the original research questions.

6. *Prepare the Report*: In the final step, the researcher report the data by transforming the problem into one that can be understood. The goal of the written report is to allow the reader to understand, question, and examine the study.

3.3 Case Context

We have chosen to study a commercial system by conducting a case study to study the consequences of design debt in this thesis. The conducted case study took place at Autronica Fire and Security AS. Autronica is a leading innovator, manufacturer, and supplier of fire safety equipment and marine safety monitoring and surveillance equipment. Their headquarters are based in Trondheim, Norway. AutroSafe, a high-end distributed fire alarm system, is one of the products they offer. The product was first released around year 2000, and has been on sale since. The product is mainly based of C/C++ source files. Project ”Firmus” is the project name for the next generation AutroSafe. ”Firmus” is a Latin word, which in English means: *solid, firm, strong, steadfast, steady, stable, reliable, and powerful*. The goal with ”Firmus” is to adopt newer technologies and technology standards that are used today. We had the opportunity to conduct our analysis on Project ”Firmus”. The project is still in the development phase. The goal of the analysis is to identify design debt before it gets worse. As we mentioned, the product is mainly based of C/C++ source code files. The software architecture of the Project ”Firmus” is component-based, where the different source files are divided into each their component. In total, the system consists of 13 components, and 461 source code files. Test files can be found inside each component. To ensure reuse of code and libraries, the company has developed a library that is used by this system and other systems as well. However, we have decided not to include library files in our analysis. Table 3.1 summarizes the system metrics, which includes the test files.

Table 3.1: System Metrics for Project "Firmus"

Project "Firmus"	
Lines	88465
Lines of Code	49287
Lines of Comments	23017
Components	13
Files	461
Number of Classes	339

3.4 Research Process

A research process provides a systematic approach on how to fulfill the goal of a research. We have chosen to follow the principles of the six steps defined by Soy [17] in this study: *1) Determine and Define The Research Questions, 2) Select the Cases and Determine Data Gathering and Analysis Techniques, 3) Prepare to Collect Data, 4) Data Collection, 5) Evaluate and Analyze Data, and 6) Prepare the Report*.

3.4.1 Determine and Define the Research Questions

The first step of this study is to define the research goal, and the research questions. In our previous research [13], we stated that we are interested in getting a deeper insight into the field of technical debt. As mentioned in Section 2.1.2, many subcategories of technical debt exists. With regards to that, we have chosen to investigate design debt in embedded systems.

An analysis of the state-of-the-art was carried out to determine what prior studies have determine about the topic of design debt. The goal with the analysis is to determine and define the research questions. *Google Scholar, ACM Digital Library, Scopus, and IEEE Xplore Digitar Library* were tremendously used during the analysis to find research papers that are relevant for our thesis. The research questions were defined after getting familiar with the topics of area for this study. The research questions will be our primarily driving force though this research. We have defined four research questions, **RQ1-4**, which we have summarized in Table 3.2.

Table 3.2: Research Questions

RQ1	How can design debt be identified?
RQ2	What are the effects of design debt?
RQ3	What kind of design debt can be found in embedded systems?
RQ4	How to pay design debt?

3.4.2 Select the Cases and Determine Data Gathering and Analysis Techniques

A representative context has be chosen to analyze and investigate the research questions. We have chosen to conduct an exploratory and descriptive case study in real-life context to obtain knowledge about the problem to be studied. The case study took place at Autronica Fire and Security for approximately six weeks. A brief description of Autronica can be found in Section 3.3. We were provided with a workspace and multiple data sources, including access to the software's source code, issue lists for the project, system requirements, and documentation for system design and code. Data were mainly extracted from these sources.

The first step is to find the structural code and design attributes of the software system. Design attributes can be identified by measuring the object-oriented metrics. In addition, object-oriented metrics can be used to assess the quality of the software. Furthermore, as we mentioned in Section 2.2, code smell detection can be used to identify design violations in the source code. A part of the literature review was to get familiar with existing tools that has been used to address similar problems. There is a wide set of tools which enables software metrics measurement and automatic static analysis of code. Some of the tools are open-source, while other contain strict licensing. However, a few of the commercial tools provide license for academical purposes. We will only focus on the tools providing measurement for C/C++. Table 3.3 list tools that have been used to extract relevant data in this case study.

Table 3.3: Tools used in the thesis

3.4.3 Prepare To Collect Data

The third step in this case study is about preparing for data collection. *Doxygen* is used by the company to generate and keep system documentation up-to-date. We spent some time analyzing the system documentation to get familiar with the system. Moreover, *Doxygen* has the ability to generate various diagrams, including inheritance diagrams, and dependency graphs. However, a downside with *Doxygen* is that it does not allow us to interact with the diagrams. *Doxygen* allows us to specify depth of the graphs that are being generated, but output can be very large, hence we had some troubles understanding the graphs. Furthermore, *Doxygen* can generate a dependency graph for each file in a component, but it does does not provide full dependency graphs for a component. There are many tools that offers reverse engineering of C/C++ source code, so we decided to try out a few of them, including *ArgoUML*, *Enterprise Architect*, and *Understand*. *Understand*, *CppClean*, *CppDepend*, and *CppCheck* will be used to extract design problems at code level by analyzing the source code.

A word document is created to keep track of the extracted data so we can review it later for analysis.

Metrics Selection

Various software metrics exists for system measurement. Unfortunately, there is no well known standardized set of software metrics aimed to measure safety-critical systems. Choosing right metrics for measuring safety-critical software is preferable because this kind of software is responsible for many things, such as human lives. To enchance quality in software systems, object-oriented metrics were established. They measure characteristics of object-oriented systems in a way to improve them. Many aspects has been defined to improve the quality of code using these metrics. We have chosen to use the metric suite defined by Chidamber and Kemerer [55]. This suite of metrics is widely cited and has been used by many researchers. For example, Rosenberg et al. [89] have applied this set of metric in evaluation of many NASA projects. The projects were written in both C++ and Java. In addition to Chidamber and Kemerer's metrics, we have measured two additional metrics, one counting the number of instance methods, and the other counting the number of instance variables. The following metrics have been measured in this study:

- LCOM (Percent Lack of Cohesion): A method is cohesive when it performs a single task. Low cohesion increases complexity, and will increase the likelihood for errors during development process. In general, the desirable value for LCOM is

to be lower. The average percentage of class methods using a given class instance variable

- DIT (Max Depth Inheritance Tree): The longest path from a given class to the root class in the inheritance hierarchy.
- CBO (Coupling Between Object classes): Number of other classes that are coupled to this class. Desirable value is lower.
- NOC (Number of Children): Number of subclasses from a given class.
- RFC (Response For a Class): The sum of the number of methods that can be potentially executed in response to a message by an object of a class.
- NIM (Number of Instance Methods): Number of instance methods in a class, which is methods that are accessible through an object of that class. Non-static methods.
- NIV (Number of Instance Variables): Number of instance variables in a class, that is, variables that are only accessible through an object of that class. This variable is used to measure LCOM values. non-static variables.
- WMC (Weighted Methods per Class): This metrics sums the cyclomatic complexity of each class by counting the cyclomatic complexity for each method.

3.4.4 Data Collection

The fourth step of the research process is to execute the plan that was created in step three. During the case study, data is collected from two different sources by using multiple tools to improve the reliability of the study. The first source of design flaws it to identify for code smells in the source code. Table 2.2 in Section 2.1.4, Chapter 2, summarizes the code smells that are presented by Fowler et al. [37]. By using automatic static analysis tools, we were able to identify multiple code smells in the system. Most of the code smells were manually verified by the researcher by inspecting the class and dependency diagrams for the class in which code smell exists. For instance, *Duplicated Code* code smell were identified using *SonarQube*. We inspected each file with duplicated code to verify the results. Another example of a code smell we identified is the Long Method code smell. Long Method code smell was identified using CppDepend and Understand. The results were verified by reverse engineering the source code to generate UML class diagrams. At first, we used the built-in functionality in Doxygen to generate the class diagrams. However, Doxygen was not able to provide full class diagrams. Therefore, we had too look for other options. We came across ArgoUML, an open source alternative to

generate UML diagrams by reverse engineering C/C++ code. After comparing some of the results with snippets from Doxygen class diagrams, we noticed that ArgoUML failed to reverse engineer some of the classes and their corresponding relations to other classes. This led us to look for commercial software. Using Understand, we were able to extract the class diagrams for the system by using their built-in reverse engineering functionality. UML class diagrams were also used to verify Large Class code smell and Long Method code smell.

The second source of design flaw extraction identification is to measure the object-oriented metrics for the system. Object-oriented metrics are used to manage, predict, and improve the quality of a software product [90]. Understand C++ was used for measuring all the metrics values of the system. The reason we chose to use this tool is because it provides wide set of metrics to be measured. In addition, Scitools, the developers behind Understand C++, offer a 15 days trial for academical academical purposes. The metrics for C++ are divided into four groups: file, class, project, and method. We have mainly focused on extracting class metrics, including LCOM, DIT, CBO, NIV, NIM, NOC, and WMC. In our measurements, we have decided to exclude the test classes as they may potentially affect the metrics in both positive and negative way. After analyzing our project, we had the opportunity to extract the measured in HTML format. We had to create a script to convert the HTML tables to Excel tables in order to compute descriptive statistics and graph creation. These tables were imported to both Excel 2013 and Google Spreadsheet. After metrics extraction, descriptive statistics were computed for the whole project and each component using formulas in Google Spreadsheet. These statistics aims to give a measure of the value of the metrics for all the classes, which we can use to identify classes with weak metric values. In addition, graphs were created using *Google Spreadsheet* and Microsoft Excel 2016.

The descriptive statistics that we have measured are:

- Minimum: The minimum value of a metric.
  Google spreadsheet formula: *MIN(value1; [value2; ...])*
- Maximum: The maximum value of a metric.
  Google spreadsheet formula: *MAX(value1; [value2; ...])*
- Sample Mean: The mean of the metric, that is, the average value of a metrics. It can be used to measure the center of the data.
  Google spreadsheet formula: *AVERAGE(value1; [value2; ...])*
- Median: Value in the middle of a given data set.
  Google spreadsheet formula: *MEDIAN(value1; [value2; ...])*

- Standard Deviation: A measure of how spread out the numbers are. Higher values indicates greater spread.
  Google spreadsheet formula: *STDEV(value1; [value2; ...])*
- Kurtosis: A measure of whether the data set have a peak or not. A positive kurtosis value indicates peaked data distribution, while negative value indicate flat data distribution.
  Google spreadsheet formula: *KURT(value1; [value2; ...])*
- Skewness: A measure which gives information about the symmetry of data distribution. Normal distribution has a skewness equal zero. Positive skewness value indicate longer right tail with sample mean placed on the right side of the peak value. Negative skewness value indicate longer left tail with sample mean placed on the left side of the peak value.
  Google spreadsheet formula: *SKEW(value1; [value2; ...])*

Code smell data were collected using automatic static analysis tools. In addition, we have manually verified some of the detected code smells in order to determine whether the hits are false negative or false positive. For example, to verify Long Method code smell, we manually inspected the class in which the method is located.

3.4.5 Evaluate and Analyze the Data

The fifth step of the case study will examine the data collected that have been collected in the fourth step. Although object-oriented metrics are well-known, there is still a discussion about identifying their threshold and usage [91]. In the previous step, we collected data from multiple object-oriented metrics. The data can be used to measure the software quality from different points of views, search for potential problems in the source code, and identify possible candidates in which inspection may potentially be needed. However, the usage of object-oriented metrics alone do not tell us which classes have acceptable metric values, and which classes need an inspection. In addition to object-oriented metric measurement, we also need to determine when the metric values represent a positive characteristic, and when the values becomes some sort of warning sign for possible unwanted aspect of the software. The boundaries between two sets of values are known as thresholds [92]. Our goal with the analysis part is to identify the classes with metric values above a defined threshold.

Threshold values will be identified using statistical information method [93]. For each metric, we used our descriptive statistics, the sample mean and the standard deviation. There are three different threshold values we will identify. The first value corresponds

with the sample mean and represents the most typical value in the data set. The second value is calculated as the sum of the sample mean and the standard deviation. It represents high values, but acceptable in some cases. The last value is the second value multiplied with 1.5 [93]. It represents extreme values and should not be present in the data set.

3.4.6 Prepare the Report

Lastly, the methods conducted in this research will be reported through this thesis. This involves all the steps that we have gone through this research, including stating the problem, performing a literature review, listing the research questions, explaining data gathering and analysis techniques used, and a conclusion where research questions are answered and suggestions are made for further research. The report also includes findings from the literature review, and how they are related to our findings from the case study. Lastly, the report conclusion makes suggestions for further research, so that other researchers may apply these techniques in some other context to determine whether findings are similar to our research or not.

3.5 Summary of the Research Design

The research questions described in Subsection 3.4.1 were answered using the case study methodology and a literature review. The literature review helped us getting familiar with the topic of area, and to identify the tools needed to mine necessary data. A case study was conducted in collaboration with Autronica Fire and Security AS, where we analyzed a system.

CHAPTER 4

RESULTS

Report Overview

- *1. Introduction*
- *2. State-of-the-Art*
- *3. Research Methodology*
- ***4. Results***
- *5. Discussion*
- *6. Conclusion*

This chapter presents the results of this study. We evaluated the design and software quality of the system by measuring object-oriented metrics related to classes, and by identifying code smells in the system. Section 4.1 presents the object-oriented metrics we have measured for Project ”Firmus”. In Subsection 4.1.1, we present the object-oriented metrics for each component in Project ”Firmus”. Finally, Section 4.2 summarizes the code smells we have detected using automatic static analysis tools.

Table 4.1: Object-oriented metrics and descriptive statistics for Project ”Firmus”

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	100	55	42.711	32.961	-1.501	-0.274
DIT	0	4	1	1.066	1.069	-0.676	0.647
CBO	0	30	5	6.178	5.162	2.093	1.214
NOC	0	20	0	0.467	1.866	61.606	7.064
RFC	0	115	10	15.991	8.769	9.279	2.936
NIM	0	48	7	8.458	7.006	8.117	2.515
NIV	0	18	1	2.195	2.828	5.131	2.003
WMC	0	325	10	20.293	31.934	41.063	5.314

4.1 Object-Oriented Metrics in Project ”Firmus”

The metrics that have been used to measure the quality of the code is mostly based on the work of Chidamber and Kemerer. [55]. They have proposed a set of static metrics that are designed to measure the quality of object-oriented software. These metrics are widely known, and their metrics suite is the deepest research in object-oriented metrics investigation and the measurements we have are the following: *Weighted Method per Class (WMC)*, *Depth of Inheritance Tree (DIT)*, *Number of Children (NOC)*, *Lack of Cohesion in Methods (LCOM)*, *Response For a Class (RFC)*, and *Coupling between Object Classes (CBO)*. In addition to these metrics, we have chosen to count the number of instance variables and instance methods in each class. A short description of each metric is provided in Section 3.4. We present their descriptive statistics which includes the minimum, maximum, median, sample mean, standard deviation, kurtosis, and skewness values for the whole system. In addition, we present descriptive statistics for each component in the system. We have decided to exclude the tests from object-oriented metrics analysis. A total of 317 files were analyzed. These files contains 226 classes, and 31204 lines of code. Table 4.1 presents descriptive statistics for class level metrics for the whole project.

LCOM: A class is cohesive if LCOM is low. In this analysis, LCOM is measured in percent. Our data reveals that LCOM value lies between a range from 0 to 100, indicating that there are classes with high and low cohesion. Figure 4.1 shows the frequency distribution of LCOM values. There are 77 classes with LCOM value of 0, indicating that these classes have high cohesion. However, 119 classes have a value of LCOM larger than 50. Among these, 7 classes have a value of LCOM larger than 90, where 2 classes have a LCOM value of 100. The normal distribution is almost perfectly symmetrical. In addition, the sample mean is very close to the peak. Classes with low cohesion increase the complexity of the software, and may therefore increase the likelihood of errors during development. In order to improve the class design, it is necessary to split one class to

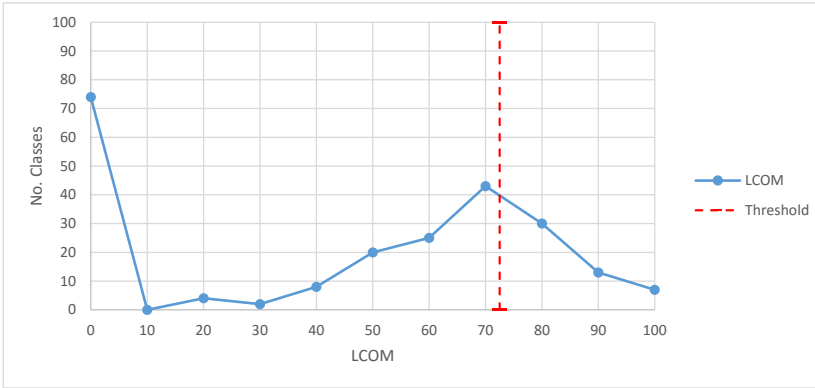

Figure 4.1: Frequency chart of the LCOM metric

two or more classes to make them more cohesive.

CBO: In general, higher values of CBO indicates fault prone classes. Our analysis show that 194 classes have a value of CBO less than 10, and that 4 classes have a value larger than 20. The maximum value of CBO is 30. This class is an example of a class that is hard to understand, harder to reuse, and more difficult to maintain.

DIT and NOC: DIT value appears to be generally low in the captured statistics. A class with DIT value of 0 is the root in a class hierarchy. Figure 4.4 shows that 89 classes have a DIT value of 0, and 68 classes have a DIT value of 1. The median value suggests that most classes tend to be close to the root in the inheritance hierarachy. According to Chidamber and Kemerer [55], DIT metric can be used to determine whether the design is top heavy or bottom heavy. A design is top heavy if there are too many classes near the root, and bottom heavy if most classes are near the bottom of the hierarchy. By observing the empirical data of DIT, the system appears to be top heavy, which indicates that there may be lack of reuse through inheritance. Classes with a DIT value of 2 and 3 indicates higher degree of reuse through inheritance. In total, we identified 70 classes with DIT value of 2 and 3. The maximum value of DIT is 4. These values show that inheritance is used in most of the classes to an optimal level. However, there may be some possibilities

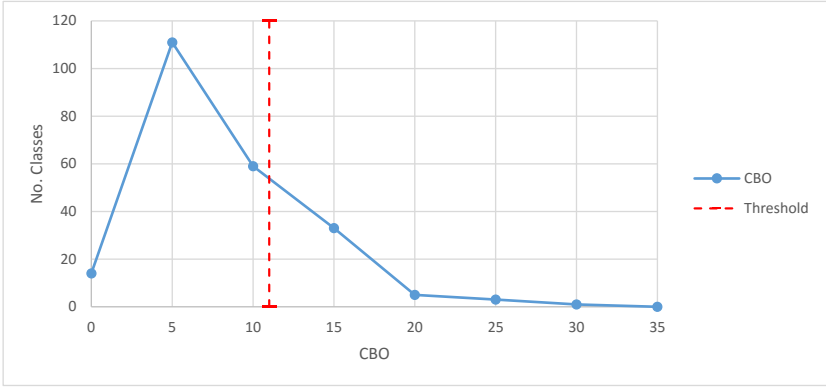

Figure 4.2: Frequency chart of the CBO metric

for improvements for classes with DIT value of 0.

NOC metric measures the number of subclasses of a class. The median value of NOC implies that most half of the classes has no immediate children, which indicates that inheritance may not be used enough. According to the Figure 4.3, approximately 86% of the classes seem to have no subclasses, hence affecting the normal distribution due to a high value of kurtosis. In addition, the skewness value indicate that long tail is on the positive side of the peak (i.e., on the right side). Furthermore, the max value of NOC is 20, which may indicate a misuse of subclassing. Classes with high NOC value are difficult to modify, and they usually require more testing because of the effects on changes on all the children.

RFC: Classes with large RFC tends to be complex and have decreased understandability. Testing classes with large RFC is more complicated. The RFC statistics reveals that majority of the classes have a RFC of less than 20. There are only 22 classes with a value of RFC larger than 30, where 2 of them have a value larger than 100. The maximum value of RFC in this system is 115. In addition, most of the classes have a WMC of less than 7, but there are a few classes with more larger values. Classes with large WMC values may indicate Large Class code smell, and these classes are candidates for inspection and

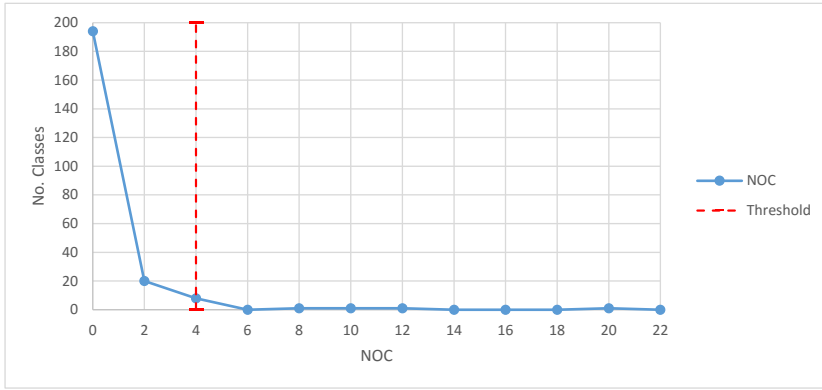

Figure 4.3: Frequency chart of the NOC metric

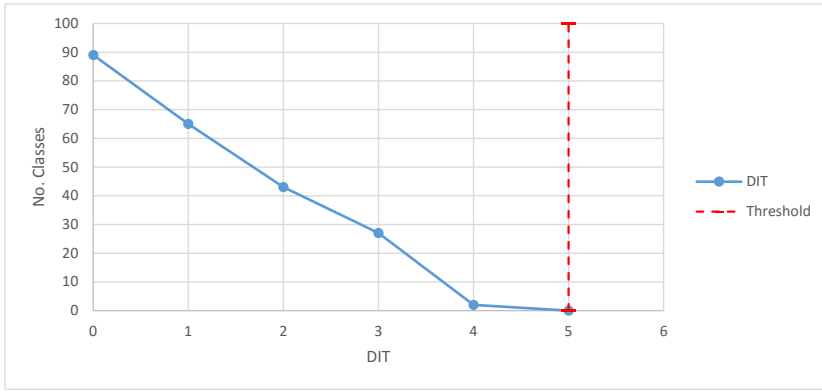

Figure 4.4: Frequency chart of the DIT metric

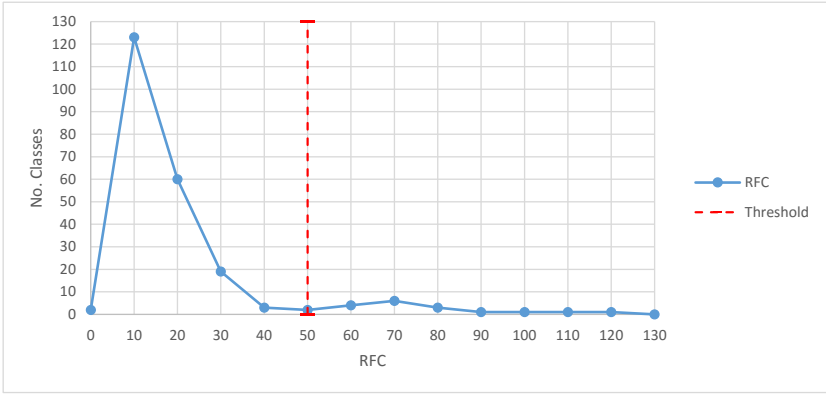

Figure 4.5: Frequency chart of the RFC metric

eventually refactoring.

WMC By examining at Figure 4.6, we observe that majority of the classes have a value of WMC less than 10. More precisely, 123 classes has a value of WMC less than 10. Moreover, 5 classes have a value of WMC2 larger than 100. The maximum value of WMC is set to 325.

NIM and NIV: NIM and NIV metric reports the number instance methods and instance variables in a class. Our analysis show that most classes are small. The sample mean of NIV tells us that each class has an average of 2 instance variables. The maximum value of NIV is 18, indicating that there is at least one class that contains 18 instance variables. The sample mean of NIM show us that each class has an average of 8 instance methods. More precisely, there are 170 classes with value of NIM less than 10. The maximum value of NIM is 48, which indicates that there is at least one class contains 48 methods.

4.1.1 Object-Oriented Metrics for the Components

Descriptive statistics in Table 4.1 reveals statistics for class level metrics for the whole project. However, the statistics does not say anything about class level metrics in the

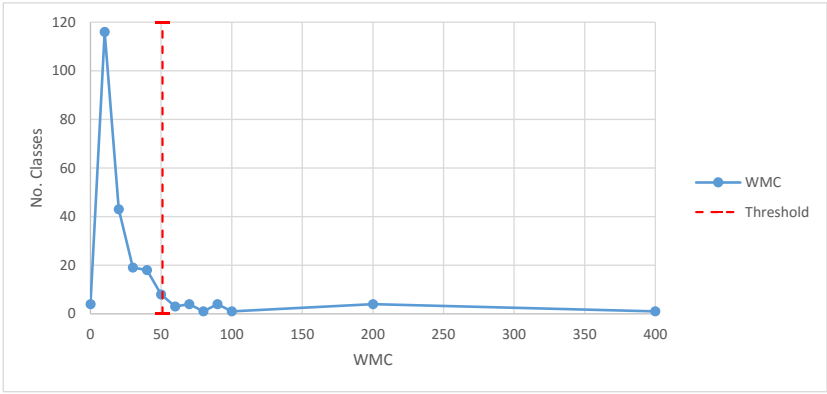

Figure 4.6: Frequency chart of the WMC metric

different components. Some of the components may have good object-oriented metric values, while other components have bad statistics. In order to identify weak components, we calculated descriptive statistics for each component.

Component A

Component A contains 53 files. Among these files, we identified 37 classes and 5427 lines of code. Figure 4.7 presents the frequency distribution of the measured object-oriented metrics. Table 4.2 presents common descriptive statistics of the metric distribution.

The DIT values indicate that inheritance hierarchies is somehow flat. Classes with flat inheritance hierarchy usually hints that reuse through inheritance is not used. There are approximately 8 classes with flat inheritance hierarchy. Rest of the classes inherits for at least one class. The max value captured show that some classes have deep hierarchy. Higher values for DIT indicates higher degree of reuse, but as trade off, it may potentially increase the complexity of the class. Moreover, the results indicate that most classes only have a few subclasses. There are 32 classes with no children. However, one class has

Table 4.2: Object-oriented metrics and descriptive statistics for Component A

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	94	62	46.405	34.321	-1.477	-0.472
DIT	0	4	2	1.567	1.167	-0.707	0.270
CBO	0	30	6	6.758	6.639	4.00	1.766
NOC	0	8	0	0.758	1.706	8.508	2.743
RFC	6	115	38	43.649	31.515	-0.708	0.559
NIM	4	40	10	13.135	8.891	3.674	1.991
NIV	0	12	1	2.270	2.969	3.694	1.917
WMC	3	194	19	31.081	36.554	10.362	2.821

NOC value of 8.

The results show that 40.5% (i.e. 15 classes) of all classes in Component A are strongly cohesive, which implies that more than half of the classes show lack of cohesion. By examining Figure 4.7, we notice that two classes has LCOM values larger than 90, indicating loose class structures. The kurtosis value of LCOM is negative, so we can consider LCOM having a flat distribution, which can be seen Figure 4.7.

Most classes have small CBO values, indicating that most classes are self-contained. However, the frequency distribution shows that few of the classes are strongly coupled. One of the classes in this component has a CBO value of 29, indicating a possible fault-prone class which affects its reusability and maintainability. This particular class has LCOM value of 62, WMC value of 95, and RFC value of 25, hence being a possible Large Class code smell.

The results show that each class have at least two methods. More than half of the classes have low RFC values, which indicates greater polymorphism. However, there are few classes in this component that has larger values of RFC. The maximum RFC value is 115, and classes with high RFC are usually difficult to maintain and test.

The values of WMC ranges from 3 to 194. The median value indicate that at least 50% of the classes have a cyclomatic complexity of 17 or less. However, the sample mean is revealed to be larger than the median value. This implies that there are few classes with large values of WMC, which is evident by inspecting the standard deviation value.

Classes with large number of instance variables are few. At least 50% of the classes have one instance variable or less. The largest number of instance variables in a class is 12, revealing that software system does not apply information hiding principle appropriately for this class. The same class have its LCOM value at 90. Furthermore, each class is revealed to have at least four instance methods. At least 50% of the classes have 10 instance methods or less. This means that rest of the classes have more than 10 instance methods, which indicates that classes may potentially provide several services to other

classes. This could be the reason behind large values of LCOM. The maximum value of NIM captured is 40. There are two classes with NIM value of 40, one having a WMC value of 194 and the other having a WMC value of 72. Both classes have a LCOM value larger than 90.

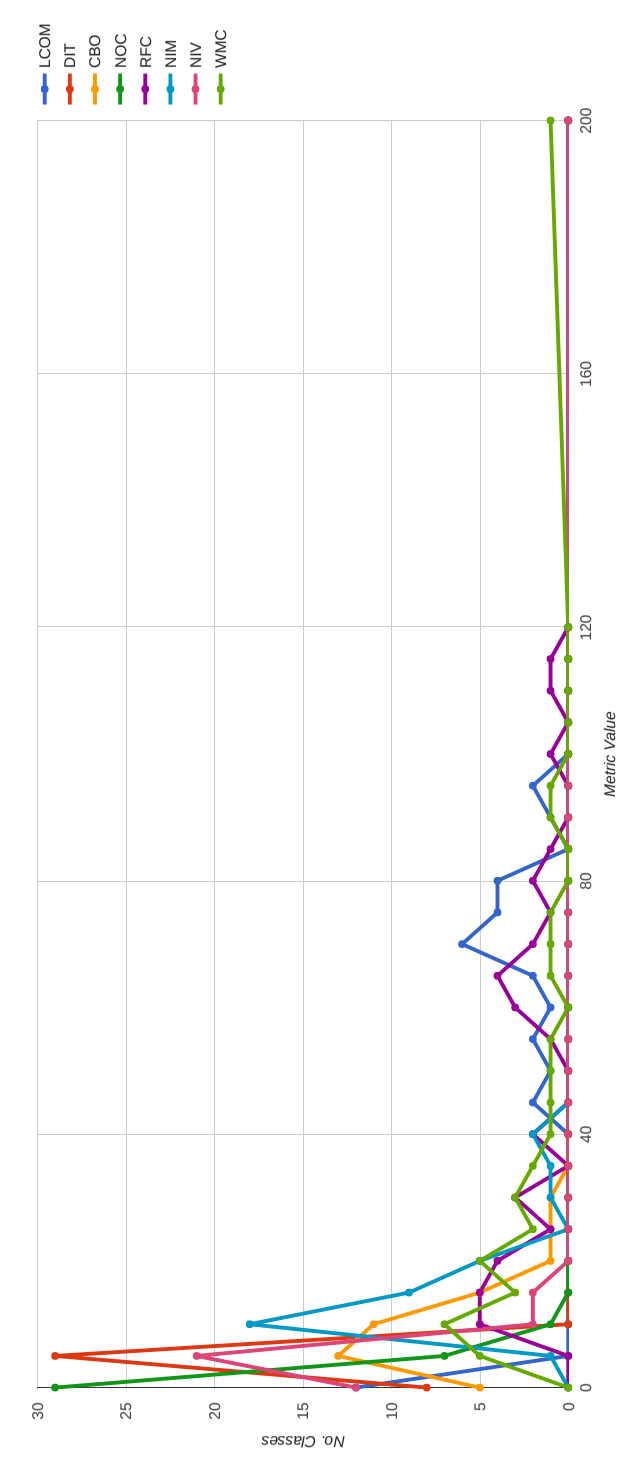

Figure 4.7: Frequency distribution of object-oriented metrics in Component A

Table 4.3: Object-oriented metrics and descriptive statistics for Component B

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	22	91	66	63.348	15.916	1.179	-0.761
DIT	0	1	1	0.609	0.499	-1.950	-0.477
CBO	1	21	5	6.13	4.799	2.653	1.350
NOC	0	1	0	0.087	0.288	8.605	3.140
RFC	6	48	9	14.217	11.977	2.867	1.831
NIM	6	48	9	13.434	10.974	3.822	1.986
NIV	1	10	2	3	2.504	2.51	1.711
WMC	3	325	19	35.826	66.796	17.288	3.964

Component B

We identified 23 classes in Component B. These classes are spread across 42 files, which in total contains 3905 lines of code. Figure 4.8 presents a frequency chart of the object-oriented metric results for Component B. Table 4.3 presents descriptive statistics of the analyzed metrics.

The values of LCOM in Component B range from 22 to 91, which indicates that none of the classes in Component B are cohesive. There are only two classes with LCOM values below 50, both having low values in terms of cyclomatic, method count, coupled objects, and instance variables. The WMC metric values range from 3 to 325. We decided to examine the class with max value of WMC. This class has a LCOM value of 74, and a CBO value of 10. Its RFC value is 44, while NIM is 36. This class has no subclasses, but it does inherit methods and variables from one superclass. In terms of cyclomatic complexity, this class is revealed to be the most complex class in the system. Moreover, there are 8 classes in this component with a LCOM value of 66. By examining the metrics of these classes, we did notice that 4 of these classes has identical DIT, CBO, RFC, NIM, NIV, and WMC values. Manual inspection of the classes did not reveal any duplicated code. In terms of refactoring, we do think that all these classes need the same effort.

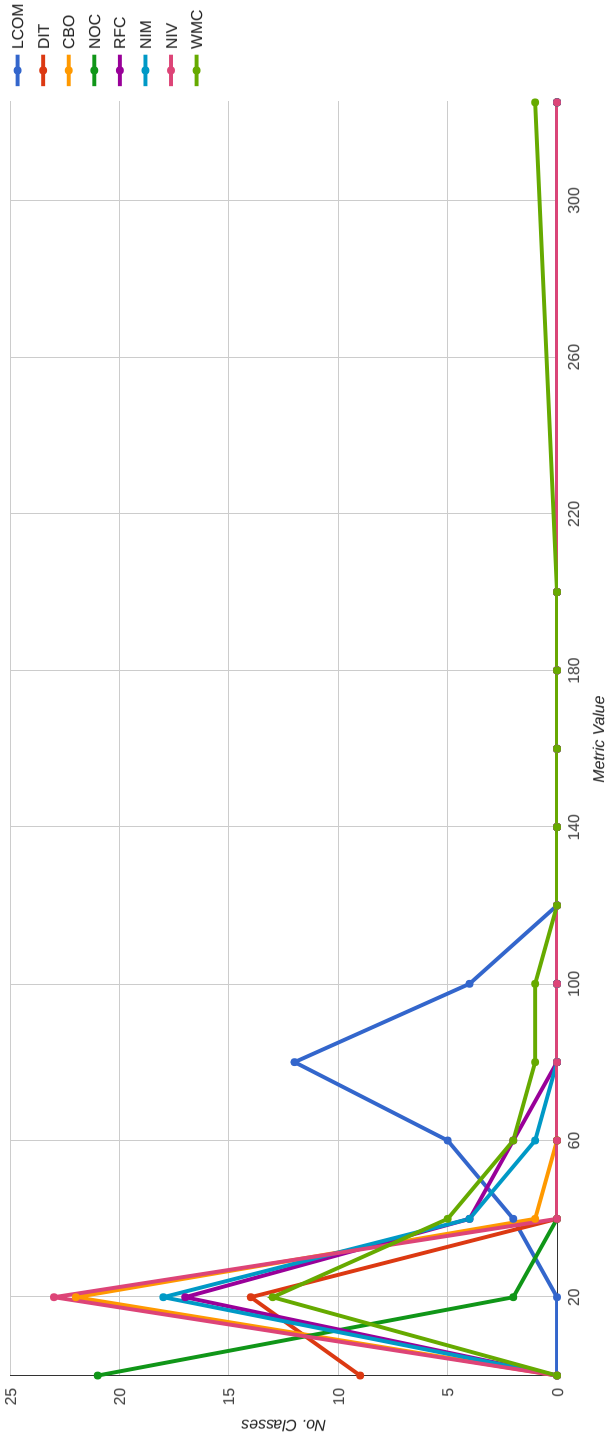

Figure 4.8: Frequency distribution of object-oriented metrics in Component B

Table 4.4: Object-oriented metrics and descriptive statistics for Component C

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	89	61	54.8	22.139	1.015	-1.089
DIT	0	1	0	0.1	0.308	7.037	2.888
CBO	1	19	4.5	6.5	5.587	-0.045	1.030
NOC	0	1	0	0.05	0.223	20	4.472
RFC	3	26	8.5	10.3	5.741	1.825	1.363
NIM	3	26	8.5	9.85	5.153	3.967	1.626
NIV	0	9	2	3.15	3.013	-0.217	1.084
WMC	3	106	12.5	27.65	30.975	1.472	1.544

Component C

Our analysis shows that component C contains 29 files, 20 classes, and 4792 lines of code. We present the descriptive statistics for component C in Table 4.4, and the frequency distribution of the metrics in Figure 4.9.

LCOM value of Component C ranges from 0 to 89. The median value reveal that more than half of the classes have LCOM value of 60 or more, indicating possibilities for design improvements by splitting up the classes. DIT and NOC metric values is very low, implying that inheritance may potentially not be used enough. Moreover, CBO values ranges from 1 to 19. Each class has a CBO value of 5 in average. One of the classes have a CBO value of 18. This class prevents reuse due to its modular design. Strong coupling complicates a system since a class is harder to understand and modify. The WMC metric values ranges from 3 to 106. The median value suggests that complexity in this component is well managed for most classes. However, we identified one class with a WMC value of 106. This particular class has a LCOM value of 63, and is coupled to 18 other objects. RFC, and NIM values of this class is set to 26, which are the maximum values that we analyzed for the corresponding metrics. All these values are an indication of a possible fault-prone and complex class. In addition, this class may potentially be affected by the Large Class code smell.

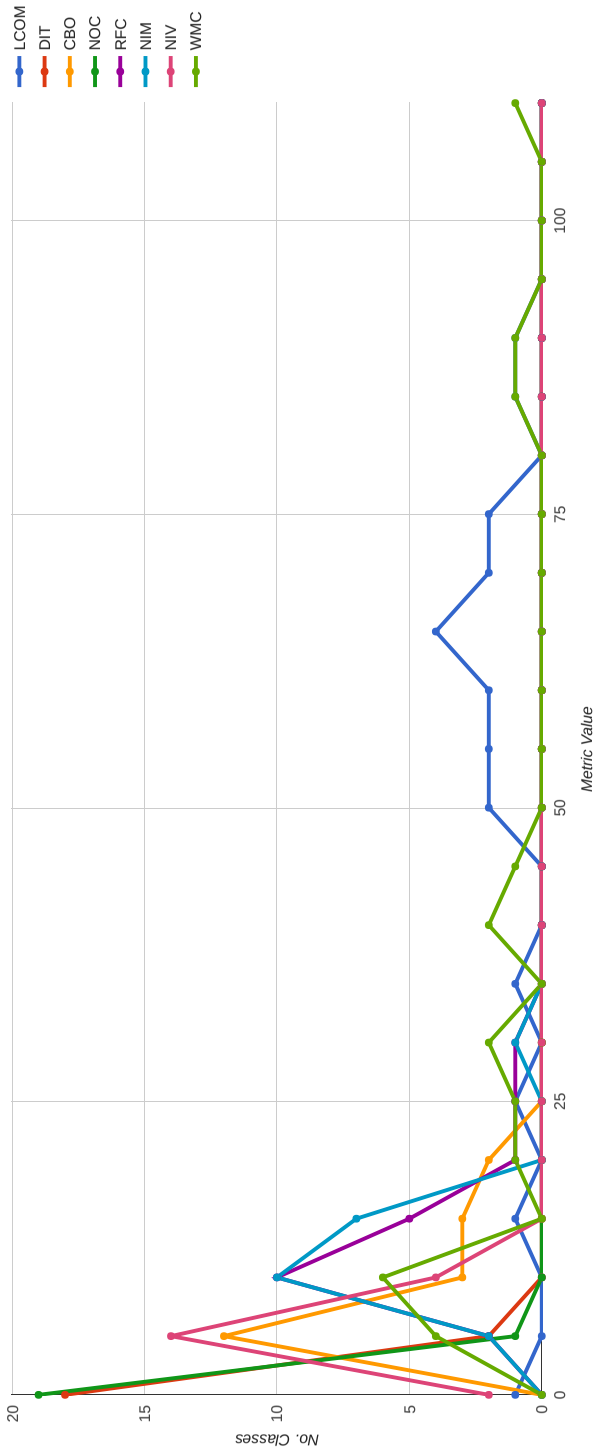

Figure 4.9: Frequency distribution of object-oriented metrics in Component C

Component D

Our analysis show that Component D consists of 13 files and 1647 lines of code. Among these files, we identified only one class. This class has LCOM value of 68. Furthermore, our results show a DIT value of 1 for this class, and a NOC value of 0. The CBO value is set to 10, implying that this class is coupled to 10 objects. The values of RFC, and NIM have a value of 8. Moreover, the class has one instance variable. The sum of complexity in this class is 17.

Component En

Similar to Component D, our analysis identified only one class among 3 files and 367 lines of code in Component En. The metrics are very similar to Component D metrics. The class has a LCOM value of 62, indicating low cohesion in some of the methods. However, compared to Component D, this class is only coupled to one object. The sum of complexity of methods in this class is 15.

Component Ex

Our analysis found 48 files in Component Ex. These files consist of 4089 lines of code, and among these, we identified 86 classes. 38% of the number of classes of the system is located in this component. Table 4.5 present the descriptive statistics for Component Ex, while Figure 4.10 present the frequency distribution of the measured metrics.

This component has classes with low and high cohesion. Despite the fact that at least 50% of the classes have high cohesion, there are still many classes with low cohesion. Our results show that there are 22 classes with LCOM value larger than 50, whereas 6 classes have a LCOM value of 80 or more. There are 2 classes with LCOM value of 100. Moreover, the average cyclomatic complexity of the classes have a value of 8.744, while the median has a value of 7. These values imply that most classes may have more polymorphism and less complexity. Figure 4.10 shows us that there are only one 8 classes with value of WMC larger than 20, where the maximum value is 41. These values indicate that the complexity is well managed to this point. By examining the CBO values, we observe that only 7 classes are self-contained. Rest of the classes are coupled to other objects. The majority of the classes has a CBO value of 5 or less. There is only one class with CBO value of 16, indicating that this class may potentially be difficult to understand and maintain.

Table 4.5: Object-oriented metrics and descriptive statistics for Component Ex

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	100	0	26.302	33.082	-0.971	0.756
DIT	0	3	2	1.581	1.121	-1.313	-0.234
CBO	0	16	4	5.314	4.459	-0.511	0.783
NOC	0	20	0	0.744	2.736	32.072	5.338
RFC	0	28	8	10.279	6.030	0.800	0.87
NIM	0	22	3	4.907	3.846	4.082	1.785
NIV	0	10	0	1.209	2.098	5.887	2.415
WMC	0	41	7	8.744	8.117	3.91	1.882

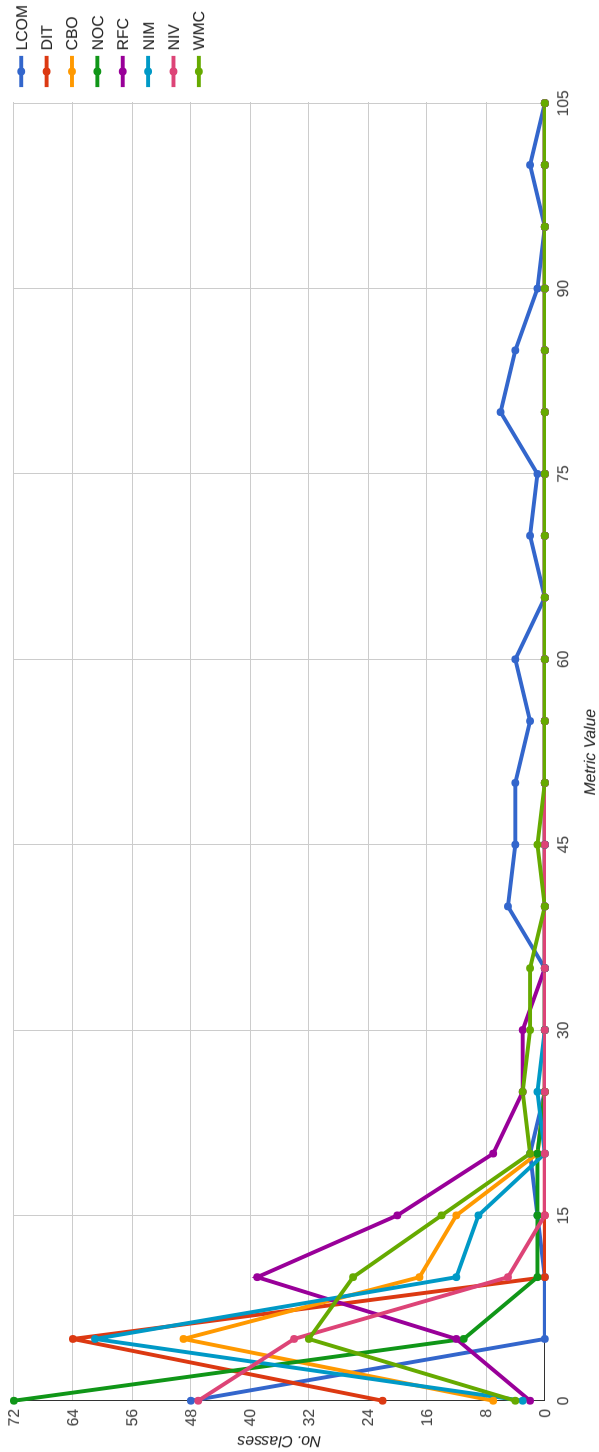

Figure 4.10: Frequency distribution of object-oriented metrics in Component Ex

Table 4.6: Object-oriented metrics and descriptive statistics for Component G

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	94	60	50.25	31.236	-0.767	-0.794
DIT	0	2	1	0.625	0.609	-0.582	0.399
CBO	0	23	5.5	6.312	4.987	1.898	1.152
NOC	0	2	0	0.25	0.622	4.231	2.357
RFC	2	30	9	10.187	6.382	2.178	1.356
NIM	0	29	7	8.437	5.459	5.669	1.745
NIV	0	18	2	3.062	3.926	5.983	2.225
WMC	1	123	12	19.437	24.794	9.572	2.824

Component G

Component G consists of 59 files. These files include 3701 lines of code and 32 classes. Descriptive statistics for Component G is summarized in Table 4.6, and its frequency distribution of the metrics are presented in Figure 4.11.

Overall, the statistics may indicate that there are some accumulated design debt in this component. The values of LCOM ranges from 0 to 94, where 8 classes has a LCOM value of 0. However, values of LCOM for rest of the classes are larger than 50. Moreover, the statistics show that at least 16 classes have a DIT value of larger than 0, which indicates a higher degree of reuse. There are only 5 classes with subclasses in this component. The WMC metric values in this component ranges from 1 to 123, where only class has a value of WMC larger than 100. We decided to examine the class with WMC value of 123. The metric of that class reveal a LCOM value of 92, CBO value of 22, RFC value of 30, NIM value of 9, and NIV value of 18. These values show that this class is probably influenced by the Large Class code smell, and is a candidate for inspection and possible refactoring.

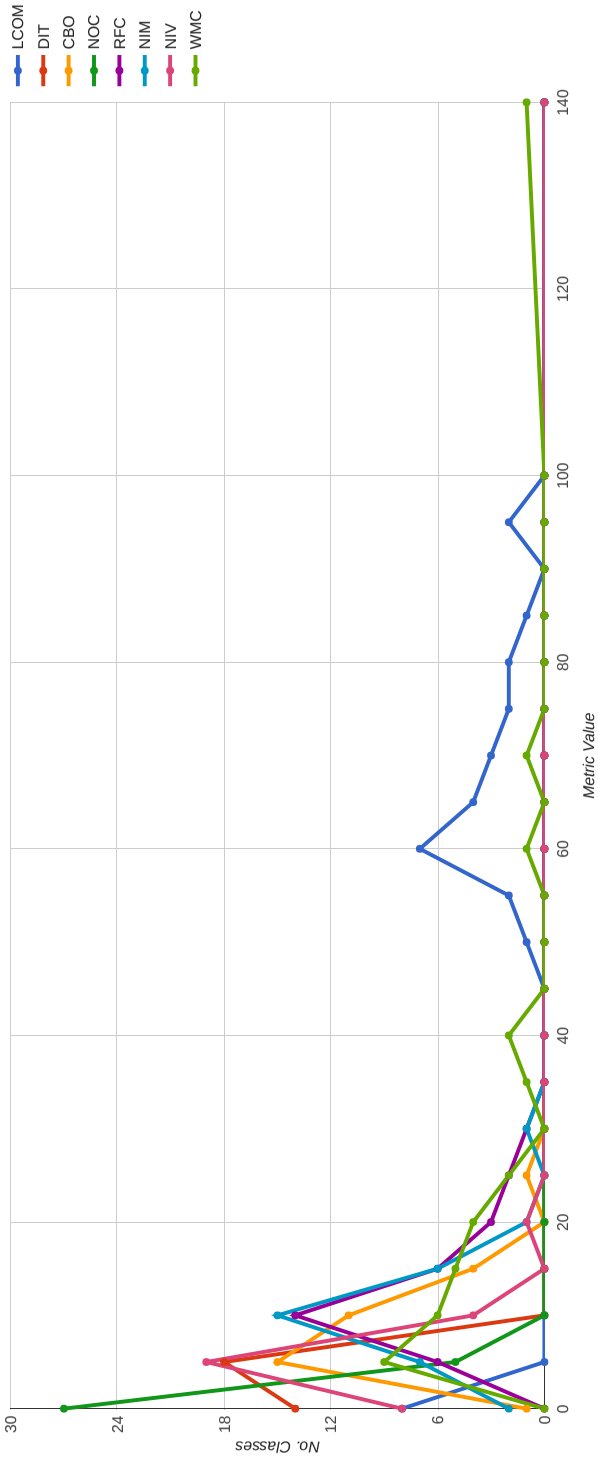

Figure 4.11: Frequency distribution of object-oriented metrics in Component G

Table 4.7: Object-oriented metrics and descriptive statistics for Component L

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	80	68	50.857	35.130	-0.908	-1.135
DIT	0	1	1	0.571	0.534	-2.8	-0.374
CBO	1	14	6	6.714	4.609	-0.712	0.327
NOC	0	0	0	0	0	N/A	N/A
RFC	5	12	9	8.571	2.936	-2.012	-0.239
NIM	3	12	9	8.286	3.402	-1.221	-0.555
NIV	0	5	1	1.571	2.070	-0.535	1.120
WMC	4	42	11	19.571	14.524	-1.521	0.577

Component L

Component L consist of 16 files, 849 lines of code. Among these, we identified 7 classes. Figure 4.12 presents the frequency distribution of the metrics, while Table 4.7 presents the descriptive statistics for this component. The values of LCOM range from 0 to 80. There are only 2 classes with LCOM value at 0. However, rest of the classes are showing lack of cohesion. These classes are candidates for inspection, and might eventually be split up into multiple classes. Moreover, 4 classes have a DIT value of 1. WMC metric value ranges from 4 to 42. There are only 2 classes with WMC values larger than 30.

Figure 4.12: Frequency distribution of object-oriented metrics in Component L

Table 4.8: Object-oriented metrics and descriptive statistics for Component N

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	79	70.5	54.5	33.899	-0.083	-1.378
DIT	0	1	0	0.25	0.463	0	1.440
CBO	3	19	12.5	11.5	4.536	1.941	-0.410
NOC	0	1	0	0.125	0.353	8	2.828
RFC	6	32	9	11.625	8.568	6.227	2.413
NIM	6	21	8.5	9.75	5.036	3.976	1.896
NIV	0	8	5.5	4.375	3.068	-1.142	-0.630
WMC	4	125	32	40.375	36.707	5.172	2.092

Component N

Component N consists of 17 files. In total, there are 1839 lines of code spread across these files. We identified 8 classes in this component. Descriptive statistics for Component N are presented in Table 4.8, while the frequency distribution of the metrics are presented in Figure 4.13.

LCOM metric value ranges from 0 to 79. The median value shows that more than half of the classes has LCOM value of 70 or more, indicating that this component should be split into more classes to increase the cohesion of each class. By taking a closer look at Figure 4.13, we identify two classes with LCOM values interval from 75 to 80. More precisely, one class has LCOM value of 78 while the other class has LCOM value of 79. We decided to examine the class with LCOM value of 79, and identified that this class has WMC value of 125, RFC value of 32, CBO value of 19, and NIM value of 21. These values tells us that this class may be affected by Large Class code smell.

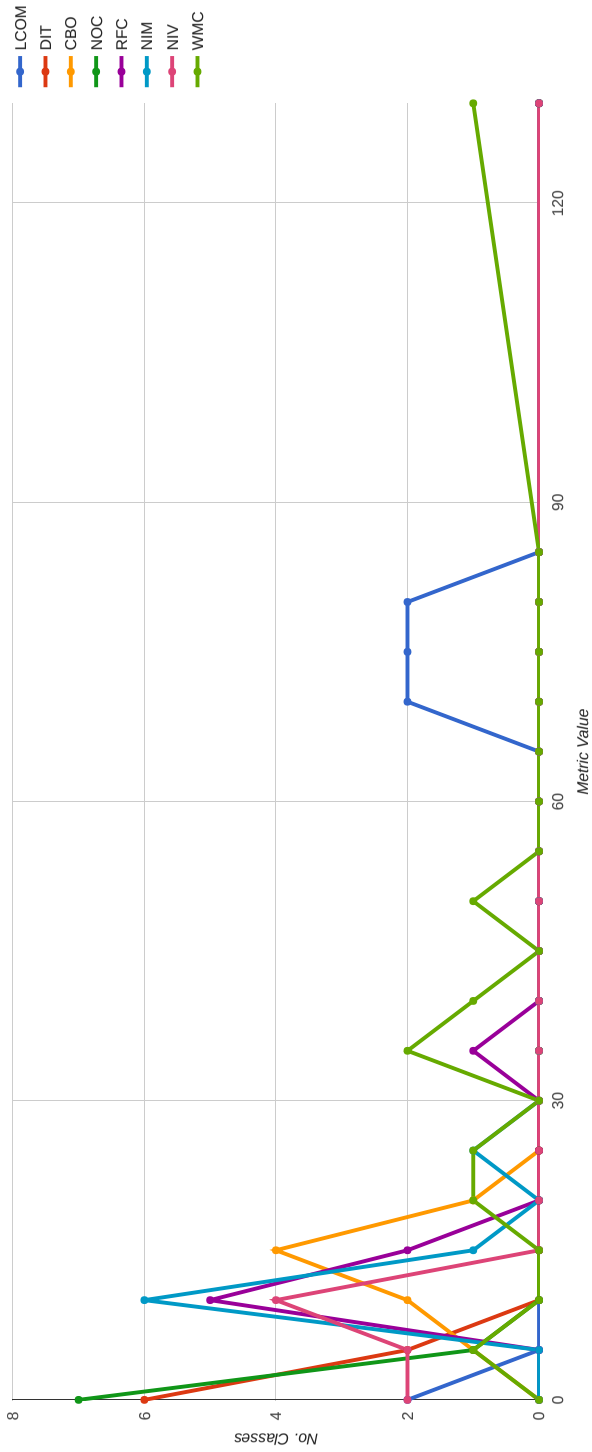

Figure 4.13: Frequency distribution of object-oriented metrics in Component N

Table 4.9: Object-oriented metrics and descriptive statistics for Component P

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	0	81	63	57.25	25.409	4.625	-1.987
DIT	0	1	0	0.25	0.463	0	1.440
CBO	0	13	6	6	4.472	-0.645	0.255
NOC	0	1	0	0.125	0.353	8	2.828
RFC	2	14	6.5	7.375	3.502	1.523	0.647
NIM	2	13	6.5	6.875	3.044	2.986	0.765
NIV	0	6	3.5	3.125	2.031	-0.886	-0.223
WMC	1	47	8	15.75	15.809	-1.037	1.350

Component P

Descriptive statistics calculated for Component P are presented in Table 4.9. The frequency distribution of the metrics are presented in Figure 4.14. We identified 12 files in Component P, consisting of 12 files, 722 lines of code, and 8 classes.

In case of measurement for cohesion, LCOM values in Component P ranges from 0% to 81%. A median value shows the level of cohesiveness in the system. In this context, the median value shows that more than half of the classes have large LCOM values, implying that these classes are improperly designed and should be split up to make them more cohesive. By examining the component, we identified three classes with values of LCOM larger than 70. The DIT results range from 0 to 1, implying that most classes have flat inheritance hierarchies. There are only 2 classes having a DIT value of 1. Despite the fact that 6 of the classes have DIT metric of 0, they alone may not tell us if classes are part of an inheritance tree or if they are root classes. By examining the NOC results, we see that only one class has a NOC value of 1. Six classes have both NOC and DIT values of zero, indicating that they are not part of an inheritance hierarchy. CBO values ranges from 0 to 12, with a mean and median of 5.5. WMC ranges from 1 to 47.

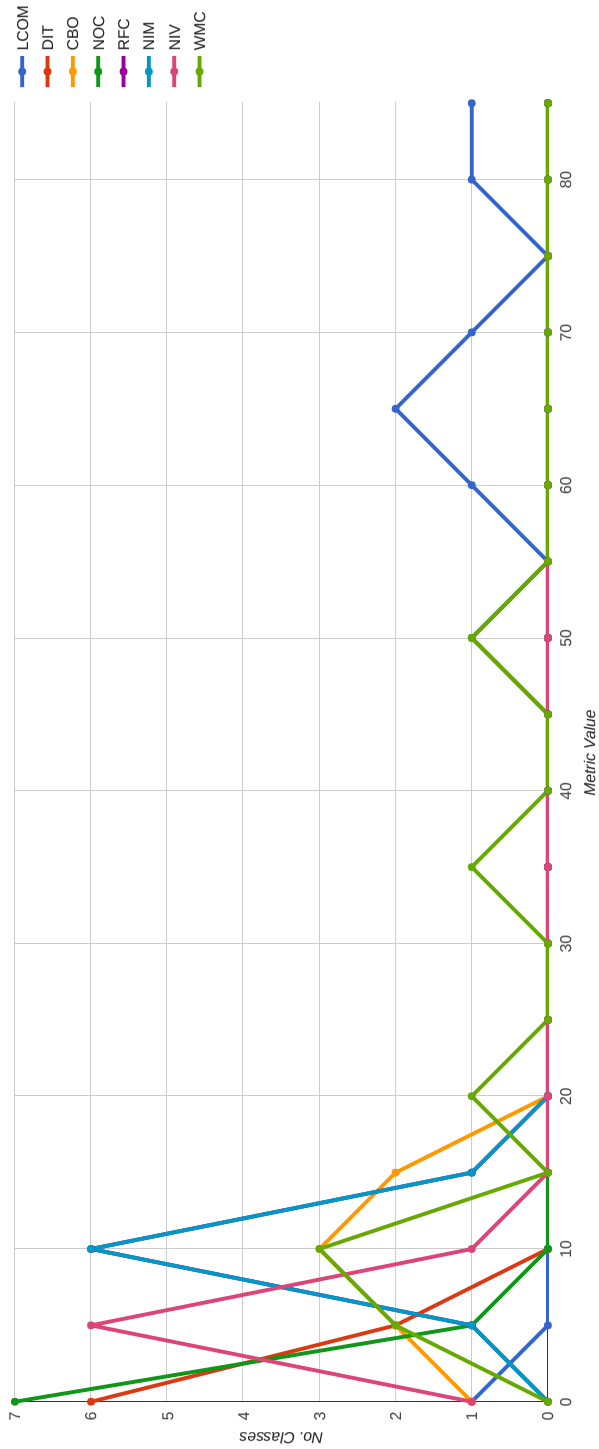

Figure 4.14: Frequency distribution of object-oriented metrics in Component P

Table 4.10: Object-oriented metrics and descriptive statistics for Component S

Metric	Min	Max	Median	Sample Mean	Standard Deviation	Kurtosis	Skewness
LCOM	33	50	41.5	41.5	12.021	N/A	-0.472
DIT	0	1	0.5	0.5	0.707	N/A	-0.472
CBO	3	7	5	5	2.829	N/A	-0.472
NOC	0	0	0	0	0	N/A	-0.472
RFC	6	9	7.5	7.5	2.121	N/A	N/A
NOM	6	9	7.5	7.5	2.121	N/A	N/A
NIM	6	9	7.5	7.5	2.121	N/A	N/A
NIV	1	1	1	1	0	N/A	N/A
WMC	6	19	12.5	12.5	9.192	N/A	N/A

Component S

Table 4.10 presents the descriptive statistics for Component S. Component S contains 4 files, which consists of 223 lines of code and 2 classes. The LCOM values show that there are possibilities to improve the design of this component by splitting up methods that does not share fields with each other to separate classes. Moreover, the results point out that none of the classes has any subclasses. However, the results reveal that one of the classes inherits methods and variables from a superclass. Moreover, WMC metric values indicate that the classes have less complexity and greater polymorphism.

Component W

Component W contains only one file. This file consists of 69 lines of code. In this component, we identified one class. This class has a LCOM value of 58, indicating that some instance variables are not shared across the member functions. Moreover, DIT and NOC values of 0 indicates that this class does not inherit from a superclass, or has any subclasses. The results reports a CBO value of 4, implying that this class is not tightly coupled with other objects. There are only 2 instance variables in this class, and 6 methods, in which all are instance methods. WMC value of this class is set to 8.

4.2 Identifying Code Smells using Automatic Static Analysis Tools

Code smells are used to find problematic classes. As we explained in Chapter 2, one of the ways to identify design debt is to look at the number of code smell in the source code. Code smell is an indicator of software design flaws in object-oriented systems that

Table 4.11: Number of Code Smells detected

Code Smell	Detected
Long Method	10
Long Parameter List	15
Duplicated Code	Approximately 5% of the source code.
Speculative Generality	1153
Dead Code	151

can decrease software maintainability which may lead to issues in further evolution of the system [39]. The analysis of code smells in the system is based on automatic static analysis tools. We investigated each tool that we could use in this project, and evaluated the smells they are able to detect. Table 4.11 describes the number of code smells that were identified using automatic static analysis tools.

Duplicated Code

Duplicated code is found by looking for pieces of code that appears at multiple places in the source code, both internally in a file or in another file. A piece of code is considered duplicated if the piece of code contains at least 10 lines of code and occurs at multiple places in the source code. Table 4.11 reports the number of duplicated code found by SonarQube, expressed as a percentage value. Including the test files, the results show that roughly 5% of the source code contains duplicated code. This corresponds to 4395 lines of code, which affects 39 files across the system. By examining the results, we identified that roughly 54% of the duplicated code is located in Component A. The other duplicated lines are spread across Component B, N, P, C, D, L, Ex, S, and G. Table 4.12 summarizes duplicated lines in the different components. The first column contains the name of the components, while the second column summarizes number of files affected by duplicated code and the number of duplicated lines among these files.

Long Method

Understand for C++ considers a Long Method as code smell if lines of code in method exceeds 200 lines. We identified 10 long methods, spread across six different files. 7 of 10 long methods are located in the test files.

Table 4.12: Duplication in Project Firmus

Component	Information
Component A	12 files, 2400 LOC
Component B	6 files, 366 LOC
Component C	3 files, 284 LOC
Component D	2 files, 80 LOC
Component Ex	4 files, 305 LOC
Component G	4 files, 311 LOC
Component L	1 file, 30 LOC
Component N	3 files, 301 LOC
Component P	2 files, 124 LOC
Component S	2 files, 194 LOC

Table 4.13: Speculative Generality Results

Category	Hits
Unused Methods	794
Unused Local Variables	346
Unused Static Globals	13

Long Parameter List

Long Parameter List code smell is detected by comparing the total number of parameters in a method against a fixed threshold. The maximum number of parameters allowed in a method using CppDepend is set to 5. This means that 6 or more parameters in a method are considered as code smell. The results from CppDepend reports 15 hits of Long Parameter List code smell, where 3 hits are considered as critical. A Long Parameter List hit is critical when total of parameters in a method is higher than 8. The largest number of parameters in a method we identified was 12. These results were verified manually by examining the class diagrams for the corresponding methods.

Speculative Generality

Speculative Generality is detected by locating unused classes, methods, fields, or parameters. Table 4.13 summarizes Speculative Generality code smell that were identified through a code analysis using Understand. The results are divided into the categories unused methods, unused local variables, and unused static globals.

Table 4.14: Dead Code Results

Category	**Hits**
”Commented Out” Code	67
Unreachable Code	10
Unnecessary Includes in Header Files	74

Dead Code

Fowler and Beck [94] do not classify dead code as code smell. However, dead code should be classified as a code smell, as it is a quite common problem as it hinders code comprehension and makes the current program structure less obvious [38]. We examined three types of ”Dead Code” code smell in Project Firmus: ”Commented Out” Code, Unreachable Code, and Unnecessary Includes in Header Files. In total, we found 151 hits of ”Dead Code” code smell, which we have summarized in Table 4.14.

CHAPTER 5

DISCUSSION

Report Overview

- *1. Introduction*
- *2. State-of-the-Art*
- *3. Research Methodology*
- *4. Results*
- ***5. Discussion***
- *6. Conclusion*

This thesis has investigated design debt in embedded systems. Section 5.1 evaluates object-oriented metrics that have been measured by applying a set of threshold values to uncover possible insecure parts of the code.

5.1 Analysis of Object-Oriented Metrics by Applying Threshold Values

Software metrics measurement in object-oriented software is important in terms of quality management [91, 92], as software metrics can be used as predictors of fault-prone

classes in object-oriented systems [56]. A study by Basili et. al [56] assessed Chidamber and Kemerers [55] suite of object-oriented metrics as predicators of fault-prone classes. Their results implied that object-oriented metrics appear to be useful to predict class fault-proneness during early phases in the software life-cycle. However, it is hard to assess the quality of a software with single object-oriented metrics.

A metric is meaningless without its threshold values. According to Tarcisio [91], software metrics are not effectively used in software industry due to the fact that for the majority of metrics, thresholds are not defined. Threshold is defined as values used to set ranges of desirable and undesirable metric values for measured software [92]. Knowing thresholds for metrics allow us to assess the quality of a software, and we may be able to identify where in a design errors are likely to occur. Lanza et al. [93] presents two ways to identify major sources for threshold values; statistical information (i.e., thresholds that are based on statistical measurements), and general accepted semantics (i.e., thresholds that are based on information which is considered common).

We identified the following thresholds for the individual metrics using their descriptive statistics from Table 4.1. The goal with applying threshold values to the metrics is to identify the classes with major design violations. Threshold values have been identified using statistical information [93]. For each metric, we have used the the sample mean and standard deviation from Table 4.1. There are three different threshold values: *low/good*, *high/typical*, and *extreme/bad*. ***Low/good*** value corresponds with the mean, and represents the most typical value in the data set [95]. Another way to calculate *low/good* is by subtracting the standard deviation from the sample mean [93]. However, the standard deviation in the data set may be larger than the sample mean, which leads to negative threshold values. This is why we have chosen to use the median value. ***High/typical*** value is the sum of the sample mean and the standard deviation. Classes with *high/typical* values are acceptable, but they should be inspected if possible. ***Extreme/bad*** value is simply the second threshold value multiplied with 1.5 [93]. It is considered as extreme, and should not be included in the data set.

Table 5.1 presents the metrics and their threshold values. The table includes threshold values that has been proposed by other researchers for the selected metrics. Despite the fact we were able to derive threshold values, our study should still be viewed as evaluation of one software system. Although our techniques can be applied to another kind of software, it is necessary to review that our threshold represent only local data. For general usage, it is recommended to compare our threshold with the the threshold defined by other researchers.

Table 5.1: Thresholds for object-oriented software metrics

Metric	Observed Value	Low	High	Extreme	Recommended Value
LCOM	100	42	75	112.5 (100+)	72.5 [91]
DIT	4	1	2	3+	4 [91], 5 [89,96,97]
CBO	30	6	11	17+	5 [89,97], 14 [98,99]
NOC	20	0	2	4+	3 [91], 5 [97], 10 [96]
RFC	115	15	34	51+	50 [89,96]
NIM	48	8	15	23+	N/A
NIV	18	2	5	7+	N/A
WMC	325	19	51	76+	34 [91], 40 [89], 50 [99]

Depth in Inheritance Tree

DIT indicates how deep a class is in the inheritance tree. Classes with higher value of DIT are associated with higher defects [100]. It is evident that a deep inheritance makes software maintenance more difficult [101]. Moreover, higher degree of *DIT* indicates a trade-off between increased complexity and increase reuseability. By following Chidamber and Kemerer's [55] guide to interpreting *DIT* metric using descriptive statistics, a low median value indicates that at least 50% of the classes tend to be close to the root in the inheritance hierarchy. In their study, a low median value had a typical value of 1 and 3. However, if the majority of *DIT* values are below 2, it may represent a poor exploitation of the advantages of object-oriented design and inheritance, because a *DIT* value of 2 and 3 indicates higher degree of reuse.

Figure 4.4 in Chapter 4 show that approximately 39% of the classes have a *DIT* value of 0, while 26.7% of the classes have a *DIT* value of 1. These classes are considered to be close to the root in the inheritance tree, and there may be a probability of not exploiting the advantages of object-oriented methodologies. Classes with high values of *DIT* have shown to be very significant in identifying fault-prone classes [56, 102]. We have derived the following threshold values for the *DIT* metric: a *low/good value* of 1, a *high/typical* value of 2, and an *extreme/bad* value of 3. By applying these thresholds, we observe that at least 26.7% of the classes satisfies good value. However, the *extreme/bad* value of 3 does not comply with the median value of 3, which is considered as a good value [55]. Therefore, we have chosen to apply the recommended max value of 5. As shown in Figure 4.4, there are no classes with a *DIT* value of 5 or more. However, there are two classes with *DIT* value of 4. These results do indicate that reuse opportunities through inheritance is limited and perhaps compromised in favor of comprehensibility of the overall architecture. On the other hand, low values of *DIT* suggest that appropriate design preferences are being followed by the company [100].

Number Of Children

NOC measures the number of children a class has. The greater number of children a class has, the greater potential of reuse, since inheritance is a form of reuse. However, large values of *NOC* indicate a misuse of subclassing [56]. Moreover, classes may require more testing. Our results reveal a *NOC* median value is 0 (see Table 4.1). The distribution of *NOC* metric in Figure 4.3 shows that approximately 86.5% of all classes have no children. In addition, Figure 4.3 show that a small number of classes have many immediate subclasses. Both Chidamber and Kemerer [55] and Basili et al. [56] have observed the similar median values for *NOC* metric in their respective studies. These values suggests that designer may not be fully exploiting the advantages of inheritance as a basis for designing classes. Lack of communication could be another reason between class designers, which may hinder developers to reuse.

The following threshold values for *NOC* metric were derived from its descriptive statistics in Table 4.1: a *low/good value* of 0, a *high/typical* value of 2, and an *extreme/bad* value 4. These values can be compared to threshold values identified by Filo et al. [91]. The difference is that they classify *extreme/bad* value as 4. According to Figure 4.3, 91.2% of the classes are classified as *low/good*, 5.2% of the classes are classified as *high/typical*, and 3.6% of the classes are classified as *extreme/bad*. Classes classified as *extreme/bad* are an indication of classes that may be hard to understand and maintain, and are potential candidates for inspection. However, similar to *DIT* metric values, *extreme/bad* values indicate limitation in reuse opportunities in favor of comprehensibility of the overall architecture. This suggest that appropriate design preferences for the system are being followed.

Lack of Cohesion in Methods

LCOM is related to the counting of methods using common attributes in a class. As mentioned, smaller values of *LCOM* represent cohesive and independent classes, which is desirable since it promotes encapsulation. Larger values of *LCOM* increase the complexity of the class, hence increasing the likelihood of errors during the development process. Our derived threshold values for *LCOM* metric reveal a *low/good* value of 42, a *high/typical* value of 75, and an *extreme/bad value* of 112.5. However, as we see in Figure 4.1, there are no classes with LCOM value larger than 100. This result is influenced by the special shape of the data from LCOM metric. We clearly see that the normal distribution is not normal. The recommended threshold value for LCOM is 72.5. This value is similar to our definition of *high/typical*. We choose to apply the recommended

threshold value for *LCOM* metric. We identified 41 classes with larger *LCOM* than 72.5, which we have illustrated in Figure 4.1. Our data reveal that LCOM values seem to increase with the size of classes. Most classes with a high value of *LCOM* revealed to have large number of methods. These methods indicate higher disparateness in the functionality provided by the class. Chidamber and Kemererer [55] and Basili et al. [56] state that classes with large values of *LCOM* could be more error prone, and more difficult to test. A refactoring option would be to split the classes into two or more classes that are more well defined in terms of behavior. Moreover, *LCOM* metric can be used by the developers to keep track of whether the cohesion principle is adhered.

Coupling Between Object classes

CBO refers to the number couplings between object classes. Higher values of *CBO* indicate the extent of lack of reuse potential of a class, and that more effort may be required to maintain and test the class. Classes with higher values of *CBO* are associated with higher defects [100]. According to Chidamber and Kemererer [55], a low median had a typical value of 0, while a high median had a value of 9. A median value of 0 suggests that at least half of the classes are self-contained and do not refer to other classes. Furthermore, Basili et al. [56] state that *CBO* appears to be useful to predict class fault-proneness.

Table 4.1 reveal a *CBO* median value of 5, indicating that at least 50% of the classes refer to 5 or less object classes. The *CBO* value is generally less for most classes. Therefore, these classes are easy to understand, reuse, and maintain. However, we did notice that some of the classes with low CBO value had higher values in the other metrics, such as *WMC* and *LCOM*.

Threshold values for the *CBO* metric were derived, and the following values were identified: a *low/good* value of 6, a *high/typical* value of 11, and an *extreme/bad* value of 17 or more. Chidamber and Kemerer [55] classified a median value of 9 as high, which is similar to the recommended values in Table 5.1. Therefore, we have chosen to apply our derived *high/typical* threshold value of 11 on the *CBO* metric results. Figure 4.2 reveal that 11.3% of all classes have a coupling of 11 or more. *CBO* should be kept to a minimum in order to promote encapsulation and modularity for a class. Classes with large values of CBO are more sensitive to changes in other parts of the design, which affects the systems maintainability, reusability, and testability. Furthermore, the *CBO* metric can be used as a way to track whether the class hierarchy is losing its integrity, and whether different parts of the system are developing unnecessary relations between classes.

Response For Class

RFC is defined as the total number of methods that can be executed in response to a message to a class. This counts includes all methods available in the class hierarchy. Large values of *RFC* makes testing and debugging more complicated since it requires a greater level of understanding on the part of the tester [55]. In addition, large values of *RFC* may increase the complexity of the class. It can be hard to predict the behavior of the class since it requires a deep understanding of the potential interactions that the objects of the class have with the rest of the system, hence affecting the classes' testability. Our analysis reveal that most classes are able to invoke a small number of methods. According to Figure 4.5, 74.2% of the classes have a *RFC* value of less than 15.

The following threshold values were derived for the *RFC* metric: a *low/good value* of 15, a *high/typical* value of 34, and an *extreme/bad* value of 51 or more. The *extreme/bad* value is very similar to the recommended max value, although it is acceptable to have a RFC up to 100 [96]. However, it is recommended that a class does not have *RFC* value larger than 50. We applied the threshold value on Project *"Firmus"*, and identified 17 classes with a *RFC* value larger than 50. One of the class have its *RFC* at 115. Research point out that *RFC* has been found to be highly correlated with *WMC* and *CBO* [55]. Our results reveal that RFC is not correlated with either *WMC* or *CBO*. For example, the class with *RFC* value of 115 had a *CBO* value of 2 and *WMC* value of 22. However, its *DIT* value is 4 and may explain that the large values of *RFC* comes from inherited methods. We identified the same pattern with the class with *RFC* of 109. This particular class has a *CBO* value of 3 and a *WMC* value of 10. However, its *DIT* value is 4. This observation indicates that *RFC* is associated with *DIT*. Moreover, *RFC* values tend to be low because there are a number of classes that have no parents.

Weighted Methods per Class

WMC refers to the sum of complexity in each method. A greater value of *WMC* indicates a complex class. *WMC* is usually affected by the number of methods in a class, but there may be some cases where some methods may have low complexity while other methods have high complexity. Table 4.1 reports a median value of 10 and sample mean value of 19.707 on WMC metric. The results are quite similar to what Chidamber and Kemerer [55] and Basili et al. [56] measured in their respective studies. Another aspect of WMC data is the similarity of the frequency distribution of the metric values in both of their and our study. In Figure 4.6, we notice that approximately 60% of all classes have a WMC value of less than 10. This suggest that most classes have a small number

of methods. The following threshold values were derived for WMC metric: a *low/good* value of 19, *high/typical* value of 51, and an *extreme/bad* value of 76. According to Table 5.1, the recommended max value are ranging between 30-50, which is close to our *high/typical* value. Therefore, it seems like this value is more appropriate to use to identify classes with high values of *WMC*. We apply the threshold, and identify 7.4% (i.e., 17 classes) with a *WMC* value larger than 51. One of the classes has a *WMC* value of 325, which indicates that time and effort to develop and maintain this class may be high. The same class have 44 methods and a *CBO* value of 10. A class with many methods are most likely to be application specific, hence reducing its reuse potential. Moreover, an increase in number of methods is associated with an increase in defects [100]. Another class with a *WMC* value of 194 reveal to have 40 methods. This suggest that classes with more methods tend to have higher complexity, which indicates an increase in defects. In such cases, maintenance effort increase drastically. The 17 classes with large value of *WMC* are primary candidate classes in which code inspection and potentially refactoring is needed.

5.2 Research Evaluation

This thesis is focused on identifying design debt in embedded systems. In particular, we were able to identify design debt by measuring object-oriented metrics and detecting code smells. The software design was analyzed using a suite of object-oriented metrics proposed by Chidamber and Kemererer, one of the most used object-oriented measures. The result does indicate that embedded software developers accumulate technical debt, despite the fact that they cannot contain any errors [79, 81, 82]. However, it seems like the accumulated technical debt is manageable. An important aspect when delivering a product is to make sure the product is stable and reliable. According to our results, we can now answer the research questions that were stated in Chapter 1.

RQ1: How can design debt be identified?

The first research question is related to the techniques we have used to identify design debt in this research. Automatic static code analysis, and object-oriented metrics measurements using Chidamber and Kemerer's suite of metrics have proven to be useful in the context of design debt identification. We were able to collect a large amount of measurements that characterize the software by using various tools. Tools that have been used through this research are *Doxygen*, *Understand*, *SonarQube*, *CCCC*, *CppCheck*, *CppDepend*, and *Enterprise Architect*.

Code smells are an example of design flaws that can degrade maintainability of source code, which implies that code smells can be used as an indicator to identify fault-prone files in the system. Moreover, code smells are an indication of refactoring possibilities in code base. We have been able to identify *Duplicated Code*, *Long Method*, *Long Parameter List*, *Speculative Generality*, *Dead Code*, and *Large Classes* code smells in our analysis. *Duplicated Code* was identified using *SonarQube*. *SonarQube* analyzed the entire source code base, and identified similar code blocks that had an appearance in multiple places. *Long Method*, *Speculative Generality*, and *Dead Code* were identified using *CppDepend*, *Understand*, and *CCCC*. *Long Parameter List* code smell were detected using *CppDepend* and confirmed using *Enterprise Architect*.

Another approach to assessing the software quality is based on object-oriented metrics [48]. Object-oriented are able to predict maintenance effort more than traditional metrics can [57] by identifying design flaws, and defect-prone, change-prone, and fault-prone classes [56]. In addition, object-oriented metrics may potentially affect the quality attributes of a system. For example, large values of *WMC* will affect a system's maintainability and reusability [54]. Object-oriented metrics are calculated over data that are extracted from the systems source code. Chidamber and Kemerer's suite of metrics have been applied in empirical investigations of object-oriented systems by multiple researchers, including Basili et al. [56], Chidamber and Kemerer [55], Okike [103], and Bakar et al. [104]. We have applied this suite of object-oriented metrics to discover potential fault-prone classes.

Understand C++ is an integrated development environment that enables static code analysis through visualization, documentation, and metric tools. The software is capable of analyzing projects with multiple lines. An academical license tool was provided to us, and the tool was used in our case study to compute software metrics. Every file file in the system were analyzed, and metrics are then extracted from these files. The tool has been used by researchers. Understand have been proven to be useful for code analysis. Malhotra et al. [105] calculated threshold values of object-oriented metrics by using statistical models. *Understand* was used to extract relevant metric data from one of the systems. Furthermore, Codabux et al. [48] extracted class-level metrics for defect- and change-prone classes using *Understand*.

Threshold values were derived in this study to assist us on identifying classes with design flaws. Threshold can be defined as the upper bound value value for a metric. A metric value with a greater than its upper bound threshold value can be considered as problematic, while a metric value lower than its upper bound threshold value can be considered as acceptable. Both metric values and threshold values can be compared in design phase of software development to identify the metrics whose value is bigger than its threshold.

Results from our study suggests that refactoring can be applied to classes with larger metric value than its threshold. In other words, we were able to predict possible fault-prone classes by applying metric threshold. For instance, a class with a *WMC* value larger than its threshold value indicate high complexity. Fault-prone classes should be used as early quality indicators, and actions should be take based on extent of the problem. For example, the project team may choose to redesign the entire class in order to achieve the desired metric value.

RQ2: What are the effects of design debt?

The second research question is related to the consequences of having design flaws in safety-critical software. Object-oriented metrics have proven to be indicators of problems in software design. Classes with larger metric values than its threshold values may affect the quality attributes of a system. Our analysis reported multiple classes in which following metric values were larger than its threshold values; *CBO, RFC, WMC, LCOM, CBO, and NOC*.

Classes with large values of *WMC* are likely to be more application specific, hence affecting the software's understandability, reusability, flexibility, and maintainability quality attributes [106, 107]. Moreover, classes with large values of *RFC* may be harder to understand and test, which also affects the software's understandability, testability, maintainability [106]. In addition, *RFC* may affect system's functionality and reusability as objects communicates by message passing [107]. *LCOM* measures the cohesion of a system. Lack of cohesion in a class increases the classe's complexity, which may lead to errors during development. This metric affects the systems efficiency, reusability, and understandability [106, 107]. Large values *CBO* complicates a system, since a module is harder to understand, change, reuse, and maintain due to its excessive coupling with other classes. *CBO* evaluates the systems understandability, extendability, efficiency, reusability, testability, and maintainability of a class [106, 107]. DIT and NOC metric are related to inheritance which enables reuse. Large values of *DIT* indicates deep hierarchy which constitutes greater design complexity. Deep hierarchy enhances the potential reuse of inherited methods but in trade-off, complexity will increase which affects other quality attributes. *DIT* metric evaluates efficiency, reusability, understandability, and testability [106] of a software. *DIT* metric may also be related to flexibility, extendability, effectiveness, and functionality [107]. Furthermore, *NOC* primarily evaluates efficiency, testability, and resuability of a system [106], but it may also influence flexibility, understandability, extendability, and effectiveness of a system [107].

Code smells are manifestations of design flaws that can have negative influence on soft-

ware quality. Although the results did not investigate the effects of the identified code smells on the system studied, we have reviewed the consequences of code smell that other researchers have discovered. Sjoberg et al. [108] investigated the relationship between 12 different code smells and maintenance effort. Their result show that none of these code smells were associated with more maintenance effort. Similarly, Hall et al. [109] state that some smells indicate fault-prone code in some circumstances but that the effect these smells have on faults and software maintainability is small. Lindsay et al. [110] state that not all *Large Class* code smell are bad, some of them could be explained by decisions that perhaps are not entirely under the control of the developer. For example, a choice of particular design pattern may lead to this smell. On the other hand, both Li et al. [111] and Dhillon et al. [112] state that bad smells are positively associated with increased error rate in software projects. Furthermore, Olbrich et al. [113] proved that *God Class* code smell have a negative effect on software quality in terms of change frequency, change size, and number of weighted defects. Khomh et al. [40] provided evidence that classes with specific code smells are more subject to change than others.

These articles state that certain types of code smells can have minimal effects, while other types of code smells can have greater effects on the quality attributes of a system. For example, *God Class* code smell will create more maintenance effort than *Dead Code* code smell will. To determine the actual effects of having code smells, the file and class metric in which the code smells are located should be measured and an action should be based on those results.

RQ3: What kind of design debt can be found in embedded systems?

The third research question is related to our results from the case study. We were able to identify fault-prone classes by applying threshold on the derived object-oriented metrics, and code smells using automatic static analysis code tools during our case study. We have been able to identify *Duplicated Code*, *Dead Code*, *Speculative Generality*, and *Long Method* as the most common code smells in general. Furthermore, object-oriented metrics were useful for investigating each individual class and the software in general. For example, we identified possible *Large Class* code smell my examining the metrics. These smells indicate possible fault-prone classes.

RQ4: How to pay design debt?

The last research question is related to the management of design debt. A common approach to keep design debt from growing, or to pay back design debt, is to conduct refac-

toring and re-engineering. Codabux et al. [5] mention that refactoring is a common way to manage and ultimately get rid of technical debt. In addition, refactoring seem improve important internal measures for reusability of object-oriented classes [114]. Without refactoring, the design of the program will decay over time.

Our results show that 5% of the source code, including the test files, contain duplicated code. Removing duplicated code will reduce the number of lines of code by default. Some duplicated code were found in the same class, while other were spread across multiple classes. Therefore, they must be handled differently. In most cases during our data collection, the same block of code occurred in the same file or class. If the same code of block exists in two different methods of the same class, *Extract Method* refactoring technique should be applied. This technique creates a new method, which can be invoked from both methods that contained duplicated code [37]. In addition, we identified some cases where the same block of code occurred in different classes in the system. For example, two identical files were identified in both Component S and B. If that is the case, then *Extract Class* refactoring technique should be applied. This technique creates a new class, superclass, or a subclass, which can be reused by both of the components with duplicated code. However, if the method is a critical part of one class, then the method should be invoked by the other class.

Moreover, we identified 10 *Long Method* code smells. *Extract Method* can be applied to shorten a method. This technique finds parts the method that seem to go nicely together, and creates a new method [37]. In some cases, a method may have lots of parameters and eventually temporary variables. If the method has lots of temporary variables to hold the result of an expression, then *Replace Temp with Query* refactoring technique should be applied [37]. *Replace Temp with Query* extracts the expression into a method, and all references to the temporary variables will be replaced with the expression. Moreover, this will allow us to reuse the new method in other methods. A method with long parameter list can be slimmed down with *Introduce Parameter Object* and *Preserve Whole Object* [37].

Furthermore, we were able to identify 15 methods with the *Long Parameter List* code smell. Three of the methods were listed as critical. As mentioned in the paragraph above, Long Parameter List can be slimmed down with *Introducted Parameter Object*, and *Preserve Whole Object*. *Introduce Parameter Object* should be used when a group of parameters naturally go together, while *Presever Whole Object* should be applied when a whole object can be sent as parameter in a method call rather than several values from an object. In addition, *Replace Parameter with Method* refactoring technique can be applied when you can get the data in one parameter by making a request of an object you already know about [37].

Our results reveal 1153 hits of *Speculative Generality* code smell, including unused methods, local variables, and static globals. Fowler et al. [37] suggest that unused variables and static globals should simply be deleted. Refactoring unused methods can be done by either removing them, or by applying *Inline Method* refactoring technique if a method body is more obvious than the method itself. *Inline Method* will replace calls to the method with the method content and delete the method.

The last code smell we identified is *Dead Code*. This smell includes ”commented out” code, unreachable code, and unnecessary ”#includes in header files”. The quickest way to conduct refactoring on Dead Code code smell is to delete unused code and unneeded files. Notice that this will reduce the number of lines of code.

To summarize, design debt can be paid by conducting necessary refactoring. We believe that refactoring will keep the software quality stable, which ultimately mitigates design debt issues. However, it is worth noticing that refactoring not necessary is the solution to pay design debt. There may be some cases where refactoring is unlikely to reduce fault-proneness in classes, and may increase fault-proneness in a class instead [109]. This phenomenon affects some of the object-oriented metrics. A consideration should be taken where both metrics and other classes affected by the refactoring are involved. For example, refactoring code smells like *Long Method* and *Duplicated Code* may increase the number of methods and coupling. Code smells should be manged by prioritizing the most critical smells. If the goal is to maintain and improve a certain metric value, then remove the smell that allows to improve this metric. Furthermore, we do believe that improving the software metrics and other work practices may be better more beneficial than refactoring code smells in order to reduce the maintenance effort.

5.3 Threats To Validity

Validity is related to how much the results can be trusted [15]. We consider threats to the external, internal, and conclusion validity of this study.

This study was carried out in an industrial context. To validate this research, more experiments need to be done with different object-oriented languages on different systems, both commercial and open-source. Another limitation is that we selected bad smells based on the results from automatic static analysis tools. More statistical computation around the metrics could be done to get more precise results. Other tools should have been used, or a developed tool. Metrics were mainly dictated by what Understand could produce.

5.3.1 Internal Validity

Interval validity is the degree to which we can conclude that the dependent variable is accounted for by the independent variable [15]. In other words, it concerns any factors that could have influenced our study results.

We have used multiple tools to investigate design debt. In some cases, these tools may result false positives.

Moreover, the artifacts of the case study may be badly designed.

Another factor that may reduce the internal validity is the selection of subjects from a larger group [15]. In this research, we found our subject for the case study after being recommended by others. The internal validity would be stronger if the subjects had been selected using random sampling.

5.3.2 External Validity

External validity refers to the degree to which the results from the study can be generalized to the population [15]. The system investigated in this study consists of one simple size and a single application domain, hence increasing the threat to external validity. That said, our results may potentially not be generalized to other safety-critical systems, both open-source and commercial. Other than that, we have only analyzed a system written in C/C++, and therefore the results obtained may not be generalized to projects written in other languages.

5.3.3 Conclusion Validity

Conclusion validity refers to the degree in which correct conclusion can be drawn from the relationship between treatment and the outcome [15]. One important issue here was the sample size of the experiment. Our case study studied one system, so the statistical power is very low. Deeper studies need to be performed to confirm if our results have a more general applicability. In addition, we may have influenced the results by looking for a specific outcome. Another potential problem can be the lack of experience on conducing case studies.

CHAPTER 6

CONCLUSION

Report Overview

- *1. Introduction*
- *2. State-of-the-Art*
- *3. Research Methodology*
- *4. Results*
- *5. Discussion*
- ***6. Conclusion***

Design debt is an instance of technical debt which describes the problem of increasing complexity of software design, and deterioration of its maintainability. Design debt is recognized to be a major problem for many software projects today. As the debt increases, more time will be spent on maintaining the system which means that the software development process and software evolution will become less effective. This will delay other development processes, such as releasing new features. Being able to measure and monitor the quality of software design will help the development team to understand the current situation of having design debt by making the debt visible. Once the problems are visible, suitable actions may potentially be performed, which includes refactoring and eventually re-engineering.

This thesis presents the result of a case study that has been conducted in real-life context in collaboration with Autronica Fire and Security AS. The purpose of the study was to investigate design debt in safety-critical systems. Object-oriented metrics were measured, and descriptive statistics were computed to analyze and interpret the data. In addition, a set of threshold value were derived in order to identify classes that are most likely to pose problems for a system. Moreover, automatic static analysis tools were applied to detect code smells in the system. We do believe that these data provide enough information for the project team as a basis for further inspection. It is up to the team to determine the criticality of these classes to make the final determinitation.

The main contributions of this work are:

- **C1:** Empirical knowledge about identification of design debt in safety-critical systems by object-oriented metric analysis and code smell detection.
  - **C1.1:** A set of threshold values for the measured object-oriented metrics
- **C2:** Empirical knowledge about the different types of design debt in safety-critical systems
- **C3:** Empirical knowledge about the effects of having design debt in safety-critical systems

6.1 Future Work

Based on the results from this thesis, we outline some possibilities for future research.

- We have mainly focused on measuring Chidamber and Kemerer's suite of object-oriented metrics. A possibility for future research would be to measure other software metrics, such as cyclomatic complexity and concurrency.
- Implementation of a tool which can detect code smells and suggest possible refactoring options.
- A deeper study on a system by analyzing multiple releases of the system. The evolution of metrics and patterns over time can then be compared to see if the design is decaying or not.
- A study on the use of design patterns, design decay, design grime, and design rot in safety-critical systems.

- A study on differences between metrics thresholds measured in this study and data from other similar systems. For example, it may be possible to compare our results with the measurement of thresholds of open-source systems.

BIBLIOGRAPHY

[1] B. Graaf, M. Lormans, and H. Toetenel, "Embedded software engineering: the state of the practice," *Software, IEEE*, vol. 20, no. 6, pp. 61–69, 2003.

[2] P. Ray, R. Laupers, and G. Ascheid, "Compose: A composite embedded software synthesis approach," in *Innovations in Information Technology (IIT), 2015 11th International Conference on*, pp. 29–34, Nov 2015.

[3] A. Kyte, "Gartner estimates global it debt to be 500 billion dollars this year, with potential to grow to 1 trillion dollars by 2015," 2010.

[4] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[5] Z. Codabux and B. Williams, "Managing technical debt: An industrial case study," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, MTD '13, (Piscataway, NJ, USA), pp. 8–15, IEEE Press, 2013.

[6] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, (New York, NY, USA), pp. 47–52, ACM, 2010.

[7] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.

[8] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, (New York, NY, USA), pp. 39–42, ACM, 2011.

[9] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, (New York, NY, USA), pp. 42–47, ACM, 2013.

[10] P. Kruchten, R. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Software, IEEE*, vol. 29, pp. 18–21, Nov 2012.

[11] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic, "Mapping architectural decay instances to dependency models," in *Proceedings of the 4th International Workshop on Managing Technical Debt*, pp. 39–46, IEEE Press, 2013.

[12] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pp. 449–451, IEEE, 2007.

[13] S. K. Bhuiyan, "Managing technical debt in embedded systems," 2015.

[14] B. J. Oates, *Researching Information Systems and Computing*. Sage Publications Ltd., 2006.

[15] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[16] M. Bassey, "Case study research," *Educational Research in Practice*, pp. 111–123, 2003.

[17] S. K. Soy, "The case study as a research method," 1997.

[18] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess.*, vol. 4, pp. 29–30, Dec. 1992.

[19] E. Allman, "Managing technical debt," *Commun. ACM*, vol. 55, pp. 50–55, May 2012.

[20] Y. Guo and C. Seaman, "A portfolio approach to technical debt management," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, (New York, NY, USA), pp. 31–34, ACM, 2011.

[21] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, (New York, NY, USA), pp. 35–38, ACM, 2011.

[22] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pp. 91–100, IEEE, 2012.

[23] M. Fowler, "Technical Debt Quadrant," 2009.

[24] S. McConnell, "Technical debt," 2007.

[25] S. McConnell, "Managing technical debt." http://2013.icse-conferences.org/documents/publicity/MTD-WS-McConnell-slides.pdf, 2007. [Online; accessed 03-May-2016].

[26] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *Software, IEEE*, vol. 29, pp. 22–27, Nov 2012.

[27] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. Santos, and C. Siebra, "Tracking technical debt—an exploratory case study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 528–531, IEEE, 2011.

[28] C. S. A. Siebra, G. S. Tonin, F. Q. Silva, R. G. Oliveira, A. L. Junior, R. C. Miranda, and A. L. Santos, "Managing technical debt in practice: An industrial report," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, (New York, NY, USA), pp. 247–250, ACM, 2012.

[29] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, pp. 17–23, ACM, 2011.

[30] F. Buschmann, "To pay or not to pay technical debt," *Software, IEEE*, vol. 28, no. 6, pp. 29–31, 2011.

[31] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, *et al.*, "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[32] S. Huynh and Y. Cai, "An evolutionary approach to software modularity analysis," in *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, p. 6, IEEE Computer Society, 2007.

[33] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 411–420, ACM, 2011.

[34] SourceMaking, "Design patterns." https://sourcemaking.com/design_patterns. (Accessed on 06/07/2016).

[35] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal*, vol. 21, no. 2, pp. 289–323, 2013.

[36] C. Izurieta and J. M. Bieman, "Testing consequences of grime buildup in object oriented design patterns," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pp. 171–179, IEEE, 2008.

[37] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing programs," 1999.

[38] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 381–384, IEEE, 2003.

[39] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*, pp. 390–400, IEEE Computer Society, 2009.

[40] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 75–84, IEEE, 2009.

[41] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *ACM Sigplan Notices*, vol. 34, pp. 47–56, ACM, 1999.

[42] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, pp. 173–182, IEEE, 2001.

[43] O. Ciupke, "Automatic detection of design problems in object-oriented reengineering," in *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30 Proceedings*, pp. 18–32, IEEE, 1999.

[44] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 350–359, IEEE, 2004.

[45] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 8, ACM, 2010.

[46] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pp. 245–256, IEEE, 2004.

[47] T. Copeland, "Pmd applied," 2005.

[48] Z. Codabux and B. J. Williams, "Technical debt prioritization using predictive analytics," in *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 704–706, ACM, 2016.

[49] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 2014.

[50] S. SonarSource, "Sonarqube," tech. rep., Technical report, last update: June, 2013.

[51] J. Radatz, A. Geraci, and F. Katki, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std*, vol. 610121990, no. 121990, p. 3, 1990.

[52] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[53] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, 3rd ed., 2012.

[54] G. Quenel and H. Lövdahl, "Object oriented software quality models," *membres. multimania. fr/gquenel/site/files/doc/OOSoftQualMod. pdf*.

[55] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[56] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *Software Engineering, IEEE Transactions on*, vol. 22, no. 10, pp. 751–761, 1996.

[57] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, pp. 111–122, 1993.

[58] I. Sommerville, *Software Engineering (9th Edition)*. Pearson Addison Wesley, 2011.

[59] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[60] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, (New York, NY, USA), pp. 73–87, ACM, 2000.

[61] "IEEE Standard for Software Maintenance," *IEEE Std 1219-1998*, 1998.

[62] H. v. Vliet, *Software Engineering: Principles and Practice*. Wiley Publishing, 3rd ed., 2008.

[63] B. P. Lientz and E. B. Swanson, "Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations," 1980.

[64] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[65] W. Frakes and C. Terry, "Software reuse: metrics and models," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 415–435, 1996.

[66] J. Sametinger, *Software engineering with reusable components*. Springer Science & Business Media, 1997.

[67] W. Frakes, "Systematic software reuse: a paradigm shift," in *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pp. 2–3, IEEE, 1994.

[68] O. P. N. Slyngstad, A. Gupta, R. Conradi, P. Mohagheghi, H. Rønneberg, and E. Landre, "An empirical study of developers views on software reuse in statoil asa," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, (New York, NY, USA), pp. 242–251, ACM, 2006.

[69] W. C. Lim, "Effects of reuse on quality, productivity, and economics," *Software, IEEE*, vol. 11, no. 5, pp. 23–30, 1994.

[70] M. Al Mamun, C. Berger, and J. Hansson, "Explicating, understanding, and managing technical debt from self-driving miniature car projects," in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pp. 11–18, Sept 2014.

[71] M. Morisio, M. Ezran, and C. Tully, "Success and failure factors in software reuse," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 340–357, Apr 2002.

[72] W. B. Frakes and S. Isoda, "Success factors of systematic reuse," *Software, IEEE*, vol. 11, no. 5, pp. 14–19, 1994.

[73] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, (New York, NY, USA), pp. 50:1–50:11, ACM, 2012.

[74] W. F. Opdyke, *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[75] P. Koopman, "Embedded systems in the real world," *Course Material: Dependable Embedded Systems*, 1999.

[76] I. Crnkovic, "Component-based approach for embedded systems," in *Ninth International Workshop on Component-Oriented Programming (WCOP)*, 2004.

[77] Z. You, "The reliability analysis of embedded systems," in *Information Science and Cloud Computing Companion (ISCC-C), 2013 International Conference on*, pp. 458–462, IEEE, 2013.

[78] S. Patil and L. Kapaleshwari, "Embedded software-issues and challenges," tech. rep., SAE Technical Paper, 2009.

[79] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, no. 4, pp. 42–52, 2009.

[80] T. Sherman, "Quality attributes for embedded systems," in *Advances in Computer and Information Sciences and Engineering*, pp. 536–539, Springer, 2008.

[81] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *2007 Future of Software Engineering*, pp. 55–71, IEEE Computer Society, 2007.

[82] J. J. Trienekens, R. J. Kusters, and D. C. Brussel, "Quality specification and metrication, results from a case-study in a mission-critical software domain," *Software Quality Journal*, vol. 18, no. 4, pp. 469–490, 2010.

[83] J. C. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pp. 547–550, IEEE, 2002.

[84] H. Washizaki, Y. Kobayashi, H. Watanabe, E. Nakajima, Y. Hagiwara, K. Hiranabe, and K. Fukuda, "Quality evaluation of embedded software in robot software design contest," *Progress in Informatics*, vol. 5, pp. 35–47, 2007.

[85] M. A. B. Sarfraz Nawaz Brohi, "Empirical research methods for software engineering," 2001.

[86] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, pp. 131–164, Apr. 2009.

[87] R. K. Yin, "Case stydt research: Design and methods. volume 5," 2003.

[88] Tigris.org, "Welcome to argouml," 2001.

[89] L. Rosenberg, R. Stapko, and A. Gallo, "Risk-based object oriented testing," *24th SWE*, 1999.

[90] D. Rodriguez and R. Harrison, "An overview of object-oriented design metrics," 2001.

[91] T. G. Filo, M. A. Bigonha, and K. A. Ferrira, "A catalogue of thresholds for object-oriented software metrics," 2015.

[92] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[93] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[94] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[95] Š. Cais and P. Pícha, "Identifying software metrics thresholds for safety critical system," 2014.

[96] A. Brooks, "Metrics overview."

[97] Objecteering, "Objecteering metrics user guide."

[98] H. A. Sahraoui, R. Godin, and T. Miceli, "Can metrics help to bridge the gap between the improvement of oo design quality and its automation?," in *Software Maintenance, 2000. Proceedings. International Conference on*, pp. 154–162, IEEE, 2000.

[99] PHPDepend, "Software metrics supported by phpdepend."

[100] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, 2003.

[101] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "Evaluating inheritance depth on the maintainability of object-oriented software," *Empirical Software Engineering*, vol. 1, no. 2, pp. 109–132, 1996.

[102] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[103] E. Okike, "A pedagogical evaluation and discussion about the lack of cohesion in method (lcom) metric using field experiment," *arXiv preprint arXiv:1004.3277*, 2010.

[104] A. A. Bakar and N. Sham, "The analysis of object-oriented metrics in c++ programs," *Lecture Notes on Software Engineering*, vol. 4, no. 1, pp. 48–52, 2014.

[105] R. Malhotra and A. J. Bansal, "Fault prediction considering threshold effects of object-oriented metrics," *Expert Systems*, vol. 32, no. 2, pp. 203–219, 2015.

[106] L. H. Rosenberg, "Applying and interpreting object-oriented metrics," 1998.

[107] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on*, vol. 28, no. 1, pp. 4–17, 2002.

[108] D. I. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *Software Engineering, IEEE Transactions on*, vol. 39, no. 8, pp. 1144–1156, 2013.

[109] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 33, 2014.

[110] J. Lindsay, J. Noble, and E. Tempero, "Does size matter?: a preliminary investigation of the consequences of powerlaws in software," in *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, pp. 16–23, ACM, 2010.

[111] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[112] P. K. Dhillon and G. Sidhu, "Can software faults be analyzed using bad code smells?: An empirical study," *International Journal of Scientific and Research Publications*, vol. 2, no. 10, pp. 1–7, 2012.

[113] S. M. Olbrich, D. S. Cruze, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, IEEE, 2010.

[114] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?," in *Reuse of Off-the-Shelf Components*, pp. 287–297, Springer, 2006.