Identifying Design Debt in Embedded Systems: A Case Study

Shahariar Kabir Bhuiyan

Master of Science in Computer Science
Submission Date: June 2016
Supervisor: Carl-Fredrik Sørensen

Department of Computer and Information Science
Norwegian University of Science and Technology

Abstract

Coming.

Sammendrag

Kommer.

Preface

Here is the preface

Acknowledgements

CONTENTS

Abstract i

Sammendrag iii

Preface v

Acknowledgements vii

Table of Contents x

List of Tables xi

List of Figures xiii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Context . . . 2
- 1.3 Research Design and Questions . . . 3
- 1.4 Contrubution . . . 3
- 1.5 Thesis Structure . . . 3

2 State-of-the-Art 5
- 2.1 Technical Debt . . . 5
  - 2.1.1 Definitions of Technical Debt . . . 6
  - 2.1.2 Classification of Technical Debt . . . 7
  - 2.1.3 Causes and Effects of Technical Debt . . . 7
  - 2.1.4 Identification of Technical Debt . . . 9
  - 2.1.5 Strategies and Practices for Managing Technical Debt . . . 14
- 2.2 Design Debt . . . 14

2.3 Software Quality . . . 14
2.4 Object-Oriented Metrics . . . 14
2.5 Software Evolution and Maintenance . . . 16
2.6 Software Reuse . . . 17
2.7 Refactoring . . . 17
2.8 Embedded Systems . . . 17

3 Research Methodology **19**
3.1 Research Methods in Software Engineering . . . 19
3.2 Choice of Research Method . . . 20
3.2.1 Case Study Method . . . 20
3.3 Case Context . . . 21
3.4 Research Process . . . 22
3.4.1 Determine and Define the Research Questions . . . 22
3.4.2 Select the Cases and Determine Data Gathering and Analysis Techniques . . . 23
3.4.3 Prepare To Collect Data . . . 25
3.4.4 Data Collection . . . 26
3.4.5 Evaluate and Analyze the Data . . . 27
3.4.6 Prepare the Report . . . 28
3.5 Summary of the Research Design . . . 28

4 Results **29**
4.1 Measuring the Software Quality using Object-Oriented Metrics . . . 29
4.1.1 Object-Oriented Metrics in Firmus . . . 29
4.1.2 Object-Oriented Metrics for the whole Project . . . 30
4.1.3 Object-Oriented Metrics for the Components . . . 35
4.2 Identifying Code Smells using Automatic Approaches . . . 53

5 Discussion **57**
5.1 Evaluation of Object-Oriented Metrics by Applying Threshold Values 57
5.2 Research Evaluation . . . 62
5.3 Threats To Validity . . . 67
5.3.1 Internal Validity . . . 67
5.3.2 External Validity . . . 68
5.3.3 Construct Validity . . . 68
5.3.4 Conclusion Validity . . . 68

6 Conclusion **69**
6.1 Future Work . . . 69

Referances **70**

LIST OF TABLES

2.1 Types of Technical Debt 8
2.2 Code Smell Taxonomy 12
2.3 ISO9126 Quality Attributes 15

3.1 System Metrics 22
3.2 Research Questions 23

4.1 OO-metrics for Project Firmus 30
4.2 OO-metrics for component A 36
4.3 OO-metrics for Component B 38
4.4 OO-metrics for Component C 40
4.5 OO-metrics for Component Ex 43
4.6 Component G 45
4.7 OO-metrics for Component L 47
4.8 OO-metrics for Component N 49
4.9 OO-metrics for Component P 51
4.10 OO-metrics for Component S 53
4.11 Number of Code Smells detected 54
4.12 Duplication in Project Firmus 55
4.13 Speculative Generality Results 55
4.14 Dead Code Results 56

5.1 Thresholds for object-oriented software metrics 58

LIST OF FIGURES

2.1 Fowler's Technical Debt Quadrant 6
2.2 Technical Debt Landscape 7

4.1 Frequency chart for the CBO and LCOM metric 31
4.2 Frequency chart of the DIT and NOC metric 32
4.3 Frequency chart of the WMC and RFC metric 33
4.4 Frequency chart of the NIM and NIV metric 34
4.7 Frequency distribution of OO-metrics in Component A 37
4.8 Frequency distribution of OO-metrics in Component B 39
4.9 Coming 41
4.10 Coming 44
4.11 Coming 46
4.12 Coming 48
4.13 Coming 50
4.14 Coming 52

CHAPTER 1

INTRODUCTION

This chapter provides an introduction to this masters thesis. We begin with outlining the motivation and context for the research. Then a brief description of the research questions is presented. Thesis outline is presented in the last section.

1.1 Motivation

Successful embedded systems continuously evolve in response to external demands for new functionality and big fixes [1]. 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. When technical debt keeps accumulating, systems can become unmanageable and eventually unusable. More resources during software maintenance have to be spent on paying off the interest (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. This is known as Internet of Things. This has led to embedded systems threatened by security issues. Matthew Garret recently revealed that he had access to the electronic equipment connected to a network in every hotel room in a hotel located in London.

Several studies has classified the metaphor of technical debt into different types of debt that are associated with the different phases of software development("siter noen artikler her"). Design debt is an example of a type of technical debt, which accumulates when compromises are made in software architecture level. Software design plays a significant role in the development of large systems [4], and unlike code-level debt, design debt usually has more significant consequences [5].

1.2 Research Context

This master thesis builds upon our previous study "Managing Technical Debt in Embedded Systems" [6], a prestudy that was carried out in the fall of 2015. The written assignment for the specialization project had the following definition:

> **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 prestudy 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 are 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 identify design flaws.

1.3 Research Design and Questions

The goal of the analysis is to identify design flaws in their software before it gets worse. The relevant research methods in software engineering can be survey, design and creation, case study, experimentation, action research, and ethnography [7]. 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 [8], which fits our desire to study an industrial system. Case studies can be both qualitative and quantitative [7, 9]. The research process we have chosen to adopt in this study follows the principles of the six steps defined by Soy [10]: *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

TODO: Contribution: New knowledge about design debt in embedded software. How it differs from existing research. Will be written at the end of this study.

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

This chapter presents the state-of-the-art topics which are relevant to 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 [11]. 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 [11]:

> *"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 [12]. 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 [13–15]. Unmanaged technical debt can cause projects to face

significant technical and financial problems, which ultimately leads to increased maintenance and evolution costs [16].

2.1.1 Definitions of Technical Debt

Several researchers have attempted to give us a clear picture of what technical debt is [4, 17, 18]. Fowler [17] presents a technical debt quadrant which consists of two dimensions: *reckless/prudent* and *deliberate/inadvertent* [17]. 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 meat 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 much as learning as it is coding.

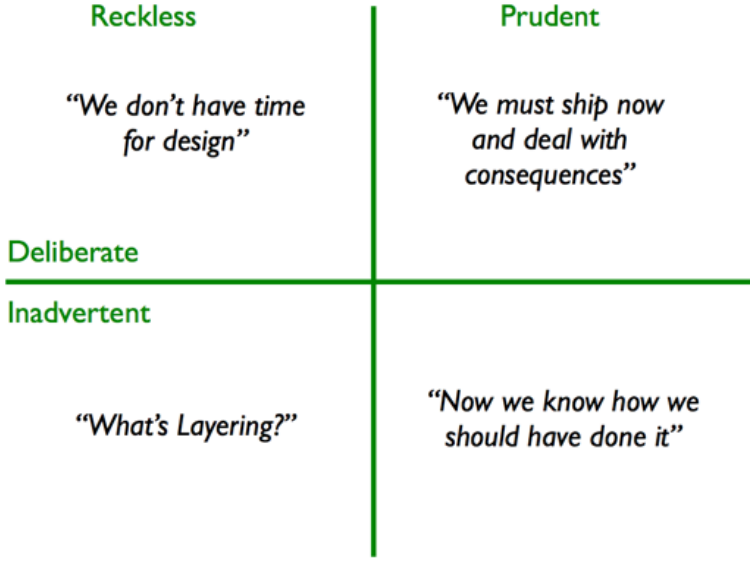


Figure 2.1: Fowler's Technical Debt Quadrant

McConnell [18] classified technical debt as intentional and unintentional debt.

Intentional debt is described as debt that is incurred deliberately. For example, an organization 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 [19, 20]. Short-term debt is usually incurred reactively, for tactical reasons. Long-term debt is usually incurred proactively, 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. [4] 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.

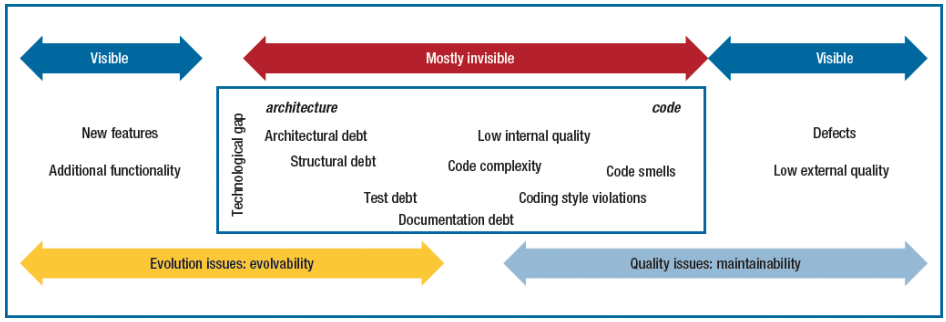


Figure 2.2: Technical Debt Landscape

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 [15, 19, 21–24] 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 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. [14] 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

Table 2.1: Types of Technical Debt

Subcategory	Definition
Architectural debt [15, 19, 21]	Architectural decisions that make compromises in some of the quality attributes, such as modifiability.
Code debt [15, 21, 22]	Poorly written code that violates best coding practices and guidelines, such as code duplication.
Defect debt [15, 22]	Defect, failures, or bugs in the software.
Design debt [15, 21, 23]	Technical shortcuts that are taken in design.
Documentation debt [15, 21, 24]	Refers to insufficient, incomplete, or outdated documentation in any aspect of software development.
Infrastructure debt [15, 19, 22]	Refers to sub-optimal configuration of development-related processes, technologies, and supporting tools. An example is lack of continuous integration.
Requirements debt [15, 24]	Refers to the requirements that are not fully implemented, or the distance between actual requirements and implemented requirements.
Test debt [15, 21, 24]	Refers to shortcuts taken in testing. An example is lack of unit tests, and integration tests.

leverage [14]. 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 stakeholders. When the project team felt pressure from the different stakeholders, technical debt decisions were made without quantifications of possible impacts.

Lim et al. [25] 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. [26] 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. [27] 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 special-

ist 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. [28] studied the effects of god classes and technical debt on software quality. God classes are examples on bad coding, and therefore includes a possibility for refactoring [23]. The results indicated that god classes require 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 [29] 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 in to 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. [19] 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 [26].

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. [30], 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 [31], 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 [30, 32]. This causes a violation in the software designed modular structure, which is called a modularity violation. Wong et al. [32] 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 [32]. Among these violations, 161 were confirmed. Zazworka et al. [30] 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 and architecture design of software systems. 23 design patterns are widely used in software development and is classified into three types: creational, behavioural, and structural. What each include.

nevn noen fordeler med bruk av design patterns.

However, software continuously evolve in response to external demands for new functionality. One consequence of such evolution is software design decay. Izurieta et al. [33] 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. That is, as a pattern realization evolves, its structure and behavior tend to deviate from its original intent. Design pattern grime is a specific type of design pattern decay [33].

However, changes in the code base could lead to code ending up outside the pattern. This is known as design grime.

Code Smell

Some forms of technical debt accumulate over time in the form of source code [30]. Fowler et al. [34] describes the concept of code smells as choices in object-oriented systems that does 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 [34]. 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 [34].

Mäntylä et al. [35] proposes a taxonomy based on the criteria on code smells defined by Fowler et al. [34]. 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 is 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 refers to classes with high coupling. The last group of problem is Other, which refers to code smells that does 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. [34].

Several studies has 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. [36] 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. [37] 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 [38]. Travassos et al. [38] present a set of reading techniques that gives specific and practical guidance for identifying defects in Object-Oriented design. However, Marinescu [39] 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 [40]. Moreover, a study by Mäntylä revealed more issues regarding manual 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.

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

Automatic approaches for identifying code smells reduce the effort of browsing through large amounts of code during code inspection process. Ciupke [40] 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 [39] 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 [41]. 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. [42] 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

Software tools play a critical role in the process of identifying technical debt. The identification of software design and code issues has been done with automatic static analysis code tools, by looking for violations of recommended programming practices that might cause faults or degrade some parts of software quality.

Mention the tools here, and its case study context.

Static analysis tools are able to alert software developers of potential problems in the source code.

ASA issues identify problems on source code line level.

Person person = aMap.get(”bob”); if(person != null) // do something with person String name = person.getName(); ¡- potential nullpointerexception.

Tools can point to the problem, and suggest solutions.

Inexpensive.

Several tools for detecting code and design issues have been proposed in the literature. Tools like PMD, Checkstyle, and FindBugs detect problems related to coding standars, bugs patterns, or unused code. There are tools that allow developers to write detection rules.

2.1.5 Strategies and Practices for Managing Technical Debt

Increasing awareness of technical debt

Detecting and repaying technical debt

Prevent accumulation of technical debt

2.2 Design Debt

Design debt is defined as su

Code smells are design flaws in object-oriented design that may lead to maintainability issues in future evolution of the software system ("Siter: The Evolution and Impact of Code Smells: A Case study of two open source systems")

2.3 Software Quality

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 [43]. These metrics are: Cyclomatic Complexity, Size, and Comment Percentage. Cyclomatic complexity evaluates the complexity of an algorithm in a method. Cyclomatic complexity for a method should be below 10 [43]. Size of a method is used evaluate the understandability of the code. Size are 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.

In addition to the traditional metrics, Chidamber and Kemerer [44] 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 [43]. 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

Table 2.3: ISO9126 Quality Attributes

Quality Attributes	Criteria	Description
Functionality		
	Suitability	
	Accuracy	
	Interoperability	
	Security	
Reliability		
	Maturity	
	Fault Tolerance	
	Recoverability	
Usability		
	Understandability	
	Learnability	
	Operability	
	Attractiveness	
Efficiency		
	Time behaviour	
	Resource utilization	
Maintainability		
	Analyzeability	
	Changeability	
	Stability	
	Testability	
Portability		
	Adaptability	
	Installability	
	Co-existence	
	Replaceability	
All		
	Compliance	

complex to predict its behavior [43]. Moreover, this metric are related to efficiency, reusability, understandability, and testability [43]. 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 [43]. 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 [43]. 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

[43]. Larger number of coupled objects indicates higher sensitivity to changes in other parts of the design. CBO evaluates efficiency and reusability [43]. 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 [43].

TODO: Nevne artikler som har tatt i bruk disse metricene.

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 [45]. Lehman [46] 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 [47]. 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 [45]. 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* [48]. Maintenance can be classified into four types [47, 48]:

- Adaptive: Modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.
- Perfective: Modification of a software product after delivery to improve performance or maintainability.
- Corrective: Reactive modification of a software product performed after delivery to correct discovered faults.
- Preventive: Maintenance performed for the purpose of preventing problems before they occur.

Van Vliet [49] 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 [49]. This has not changed since the 1980s when Lientz and and Swanson conducted a study on software maintenance [50]. 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.

Adaptive maintenance is the modification of a software product performed after delivery to keep a computer program usable in a changed or changing envionment. Corrective maintenance is the reactive modification of a software product performed after delivery to correct discovered faults. Perfective maintanence is the modification of a software product after delivery to improve performance and maintainability.

2.6 Software Reuse

2.7 Refactoring

Refactoring increases the code readability and maintainability. Refactoring depneds on the type og design defect found in the system has has a direct influence on the software maintenance cost("Siter: A quantitative investigaton of software metrics threshold values at acceptable risk level")

2.8 Embedded Systems

CHAPTER 3

RESEARCH METHODOLOGY

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

Research is believed to be the most effective way of coming to know what is happening in the world [9]. 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 [51]. Empirical studies can be explanatory, descriptive, or exploratory [8].

There are two types of research paradigms that have different approaches to empirical studies [8]; the qualitative, and the quantitative paradigm. Qualitative research is concerned with studying objects in their natural setting [8]. 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 [8]. 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 [7] 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

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 provides both quantitative and qualitative information about the system [7], 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 [8]. 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 [8, 10]. Runeson et al. [52] 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 [10, 52]. According to Yin [53], 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 [53]. Soy [10] 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

To study the consequences of design debt, we have chosen to study a commercial system by conducting a case study. The conducted case study took place at Autronica Fire and Security AS, an international company with their headquarter based on Trondheim, one of the largest cities in Norway. Autronica is a leading innovator, manufacturer, and supplier of fire safety equipment and marine safety

Table 3.1: System Metrics

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

monitoring and surveillance equipment. 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. Table 3.1 summarizes the system metrics, which includes the test files.

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. To ensure reuse of code and libraries, the company has developed a library that is used by this system and other systems as well.

3.4 Research Process

A research process provides a systematic approach on how to fulfill the goal of a research. In this study, we have chosen to follow the principles of the six steps defined by Soy [10]: *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.*

3.4.1 Determine and Define the Research Questions

First, we need to set up the goals of this research by defining the research questions. In prior to our previous research [6], we stated that we are interested in getting deeper insight into the field of technical debt. As mentioned in Section

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?

2.1.2, many subcategories of technical debt exists. With regards to that, we have chosen to investigate design debt in embedded systems.

In order to determine and define the research questions, we begin with an analysis of the state-of-the-art to determine what prior studies have determined about the topic of software design flaws. Google Scholar, ACM Digital Library, Scopus, and IEEE Xplore Digital Library were used tremendously in order to gather research papers for the analysis of the state-of-the-art topics that are relevant for our thesis. After getting familiar with the topics of area of this study, we defined our research questions for this study. These research questions will be our primarily driving force thorough this research. We have defined four research questions RQ1-4, which we have summarized in Table 3.2.

3.4.2 Select the Cases and Determine Data Gathering and Analysis Techniques

To investigate the research questions, a representative context has to be chosen. With regards to that, 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 AS for approximately six weeks. A brief description of the company can be found in Section 3.3. Autronica provided us a workspace and multiple data sources, including access to their source code, issue lists in Stash, system requirements, and design and code documentation. Our data were mainly extracted from these sources.

The first step is to find the structural code and design attributes of the software system. A part of the literature review was to get familiar with existing tools that has been used to address similar problems. Below, we have listed each tool that has been used to extract relevant data in this case study. Doxygen is used by the company to generate and keep the documentation up-to-date. In order to understand the system and how the different components interact, we spent some time analyzing the system documentation. Moreover, Doxygen can generate various diagrams such as inheritance diagrams, and dependency graphs. However, a downside with Doxygen is that it lacks a feature for interacting with the diagrams and graphs. Doxygen allows us to specify depth of the graphs that are generated, but they can become very large, which makes the graphs difficult to understand. Moreover, Doxygen does not provide full diagrams for internal

dependencies in each component, it can generate dependencies for a chosen file. 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.

Some static analysis tools has been used to extract design problems at code level. These includes Understand, SonarQube, CppClean, CppDepend, CppCheck, and Sonargraph.

Selection of Tools

Various tools have been used to mine for data and understand the structure of the system studied. The following sections will describe the tools that has been used in this research.

Doxygen is a free software for generating documentation from annotated C++ sources. Doxygen has the ability to generate documentation in HTML or in Latex. Since the documentation is extracted from the source code, it is easier to keep the documentation up to date. In addition, Doxygen can be configured to extract the code structure from undocumented sources files, which makes it possible to visualize the relations between various elements in the software. Doxygen is used by the company to keep the documentation up-to-date, and was therefore used in this project to learn about the system and to visualize the system.

ArgoUML is an open source UML modeling tool that includes support for all standard UML 1.4 diagrams [54]. ArgoUML features reverse engineering of C++ projects by reading C++ source files and generate and UML model and diagrams.

Enterprise Architect is a commercial UML modeling tool. It has the ability to produce UML diagrams from code. This tool was used to create class diagrams for each component, allowing us to identify possible code smells, such as Large Class code smell, and God Classes.

SonarQube is an open source platform for quality management of code. It has the ability to monitor different types of technical debt. It supports multiple languages through plugins, including Java, C/C++, JavaScript, and PHP. A downside with SonarQube is that some of the plugins requires a commercial license, and that some features included are not applicable for C++.

Understand is a commercial code static analysis tool. It supports dozen of languages, including Java, C/C++, Fortran and Python. Understand can help developers analyze, measure, visualize, and maintain source code. It includes many features, including dependency graph visualization of code, and various metrics about the code (e.g. coupling between object classes),.

CppDepend is a commercial static analysis tool for C/C++ code.

CppClean is an open source static analysis tool for C/C++ code. It attempts to detect problems in C++ that slow development in large code bases. Among

many features, CppClean supports finding unnecessary *'#includes'* in header files, global/static data that are potential problems when using threads, and unnecessary function declarations.

CppCheck is another open source static analysis tool for C/C++ code. Unlike CppClean, CppCheck detects various kinds of bugs that the compilers normally do not detect, such as memory leaks, out of bounds, and uninitialized variables.

CCCC is a free software tool for measurement of source code related metrics. This tool are able to measure some of the metrics that are defined by Chidamber and Kemerer [44].

SourceMonitor is a program which inspects the source code to find out how big the system is and to identify the relative complexity of the modules. It collects metrics through source files.

Data is collected from issue lists, and has been mapped to their corresponding components and source files in order to find which component we should look more at. This data will be compared to the data we have extracted using static analysis tools mentioned above.

3.4.3 Prepare To Collect Data

The third step in this case study is about preparing for data collection. Firstly, we need to review the system documentation in order to get familiar with the system and its components. Due to the time limit, we have chosen to collect data from three critical components using the tools listed in Section XX. We prioritize the components using the issue lists, and break down the components to identify bad design. Bad design includes god classes, dependency cycles, complex interfaces, and unused code in the components. A Word document is created to keep track of the extracted data so we can review it later for analysis. If any interviews would be needed, they would be set up and planned on this stage.

Metrics Selection

Using Understand, we measured the software metrics by analyzing the source code. In total, we extracted 9 metrics for each class on the system, excluding the tests.

- 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.
- NOM (Number of Methods): The number of all local methods in a class. More member of functions is considered to be more complex and therefore more error prone. Desirable value is low.
- WMC (Weighted Methods per Class): This metrics sums the cyclomatic complexity of each class by counting the cyclomatic complexity for each method.

For each metrics, we have computed descriptive statistics on all the classes of the system. 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. These statistics are:

- Minimum: The minimum value of a metric.
- Maximum: The maximum value of a metric.
- 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.
- Median: Value in the middle of a given data set.
- Standard Deviation: A measure of how spread out the numbers are. Higher values indicates greater spread.

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. [34]. 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 God Classes. In addition, a dependency graph was extracted, which show internal dependencies in each component, and dependencies between components. We noticed two form of dependencies, direct dependencies and circular dependencies.

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 [55]. Using Understand, CCCC, and SourceMonitor, we were able to extract multiple metrics. More precisely, we have extracted the following metrics: LCOM, DIT, CBO, NIV, NIM, NOM, NOC, and WMC. In our measurements, we decided to exclude the test classes as they may affect the metrics in both positive and negative way.

3.4.5 Evaluate and Analyze the Data

In the fifth step of the case study, we will examine the data collected in fourth step. For the data analysis of the collected object-oriented metrics, we perform some statistical analysis by measuring the descriptive statistics for the different metrics. Descriptive statistics are used to summarize data in a meaningful way. By examining the data, we may be able to identify some patterns in the data. The descriptive statistics we have measured are the following; minimum, maximum, median, sample mean, and standard deviation. The minimum and maximum value helps us to identify the smallest and largest data value in the data set. The median value is the midpoint of the data set. It helps us identifying which half of the observations are above and below the midpoint. Sample mean is the average of the data, which is found by summing all of the observations divided by the number of observations. Both median and sample mean measure the central tendency. At last, the standard deviation measures how spread out the data are from the mean. Higher values indicates greater spread in the data. Furthermore, we calculate the frequency distribution for the different metrics. This was done to make the findings more accessible by visualizing the data, and to interpret

the data afterwards. The results are then compared with some of the defined thresholds in the literature, and our findings from the literature review.

Code smells were identified using tools, and analyzed using manual approaches. We have manually inspected some of the detected smells in order to determine whether the hits are false negative or false positive. For example, to verify a Long Method code smell, we manually checked the method in its class.

3.4.6 Prepare the Report

At last, the methods conducted will be reported through this thesis. This involves all the steps that we have gone through this thesis, 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. At last, 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 Section XX were answered using the case study methodology and a literature review. The literature review helped us getting familiar with the topic and to identify the tools needed to mine data. Furthermore, the case study was conducted in collaboration with a company located in Trondheim.

CHAPTER 4

RESULTS

In this chapter, we present the results of this study. In this study, we evaluated the design and software quality of the system. This study also addresses all of the research questions.

4.1 Measuring the Software Quality using Object-Oriented Metrics

One way to identify design debt is to measure the software quality using metrics.

The first step to identify design debt is to find the structural code and the design attributes of a system. The design attributes in this case is the object-oriented metrics.

4.1.1 Object-Oriented Metrics in Firmus

In addition to the traditional metrics, we have also gathered data using object-oriented metrics. Traditional software metrics are important for identifying large and complex files, but they alone may not tell us why some classes are large and complex. Object-oriented metrics we have used to measure the quality of the code is mostly based on the work of Chidamber and Kemerer. [44]. 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

Table 4.1: OO-metrics for Project Firmus

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	100	55	42.711	32.961
DIT	0	4	1	1.066	1.069
CBO	0	30	5	6.178	5.162
NOC	0	20	0	0.467	1.866
RFC	0	115	10	15.991	18.769
NIM	0	48	7	8.458	7.006
NIV	0	18	1	2.195	2.828
WMC	0	325	10	20.293	31.934

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 "X". We present their descriptive statistics which includes the minimum, maximum, median, sample mean, and standard deviation values for the whole system. In addition, we present descriptive statistics for each component in the system.

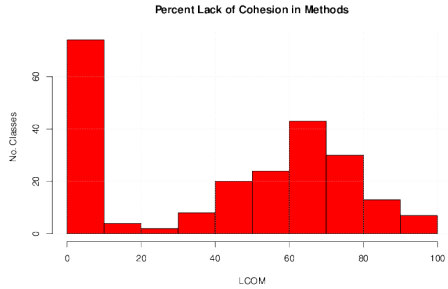
The metric calculation are from several tools i.e., Understand, and CCCC. They all provide extensive numbers of software quality metrics.

A description of object-oriented metrics can be found in Chapter 3, Section X.

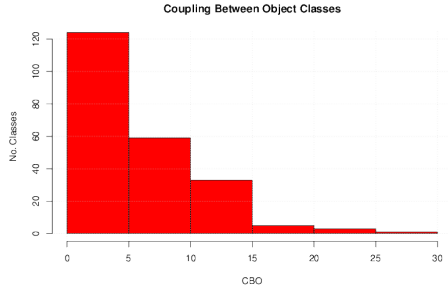
4.1.2 Object-Oriented Metrics for the whole Project

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. Descriptive statistics such as minimum, maximum, median, sample mean, and standard deviation are presented in this section. 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 "X" show the frequency distribution of LCOM values. There are 77 classes with LCOM value of 0%, indicating that these classes has high cohesion. However, 119 classes has 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%. Classes with low cohesion increases 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 two or more classes to make them more cohesive.



(a) Frequency chart of the LCOM metric

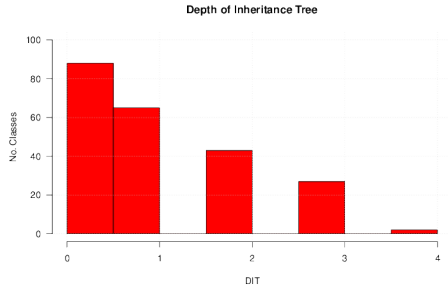


(b) Frequency chart of the CBO metric

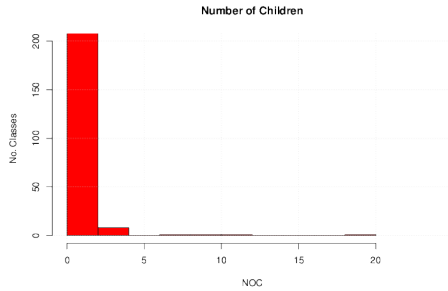
Figure 4.1: Frequency chart for the CBO and LCOM metric

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 "X" 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 [44], 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 for improvements for classes with DIT value of 0.



(a) Frequency chart of the DIT metric



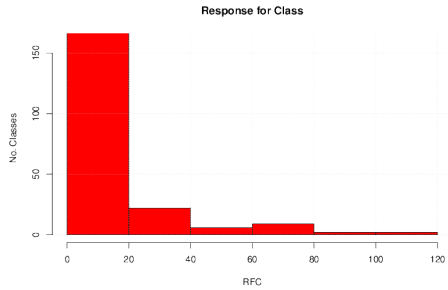
(b) Frequency chart of the NOC metric

Figure 4.2: Frequency chart of the DIT and NOC metric

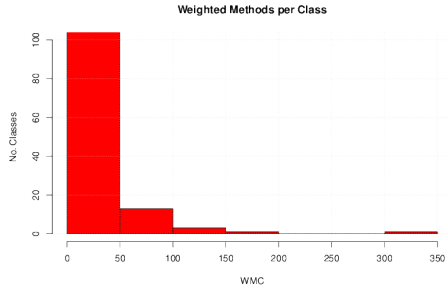
Furthermore, the NOC metric measures the number of subclasses of a class. The median value of NOC implies that most half of the classes has no immediate children, which indicates that inheritance may not be used enough. According to the Figure "X", approximately 86% of the classes seem to have no subclasses. 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 eventually refactoring.

WMC By the WMC2 metric, we can observe the complexity of a class by summing complexity of all methods. In general, low values of WMC2 indicates greater

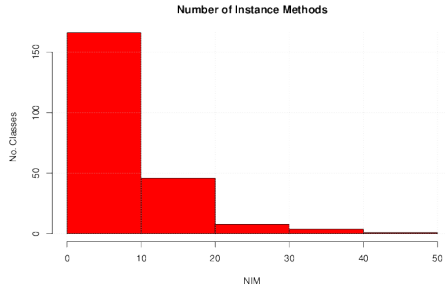


(a) Frequency chart of the RFC metric

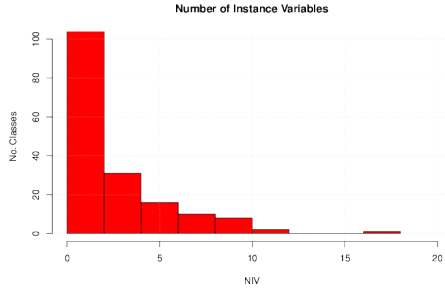


(b) Frequency chart of the WMC metric

Figure 4.3: Frequency chart of the WMC and RFC metric



(a) Frequency chart of the NIM metric

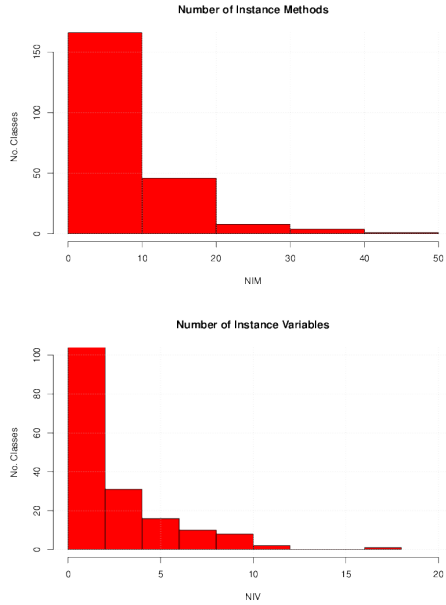


(b) Frequency chart of the NIV metric

Figure 4.4: Frequency chart of the NIM and NIV metric

polymorphism in a class, while higher higher values of WMC2 indicates more complexity. By examining at Figure "X", we observe that majority of the classes have a value of WMC2 less than 10. More precisely, 123 classes has a value of WMC2 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. NIM and NIV metric can help us identify Large Class code smell.



4.1.3 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 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 visualizes the frequency distribution of the analyzed 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 eight classes with flat intheritance 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 tradeoff, it may increase complexity of the class. Moreover, the results indicate that most classes only have a few subclasses. Thirty-two classes has no subclasses. However, one class has NOC value of eight.

Table 4.2: OO-metrics for component A

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	94	62	46.405	34.321
DIT	0	4	2	1.567	1.167
CBO	0	30	6	6.758	6.639
NOC	0	8	0	0.758	1.706
RFC	6	115	38	43.649	31.515
NIM	4	40	10	13.135	8.891
NIV	0	12	1	2.270	2.969
WMC	3	194	19	31.081	36.554

The results show that 37.5% (i.e. 15 classes) of all classes are strongly cohesive, which implies that more than half of the classes show lack of cohesion. By examining Figure 4.7, we see that two classes has LCOM values larger than 90%, indicating loose class structures. Furthermore, 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 class have CBO value of twenty-nine, indicating a possible fault-prone class which affects its reusability and maintainability. This particular class has LCOM value of 62, WMC2 value of 95, and RFC value of 25.

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 more high RFC. The maximum RFC is 115, and classes with high RFC are usually difficult to maintain and test. However, the class with a RFC value of 115 has

The values of WMC2 ranges from 2 to 194. The median value indicate that half 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 WMC2, which is evident by inspecting the standard deviation.

Classes with large number of instance variables are few. More than half of the classes has one instance variable or less. The largest number of instance variables is 12, revealing that software system does not apply information hiding principle appropriately for this class. Furthermore, each class is revealed to have two instance methods. Approximately 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 provide several services to other classes. The maximum value of NIM captured is 40. There are two classes with NIM value of 40, one having a WMC2 value of 194 and the other having a WMC2 value of 72.

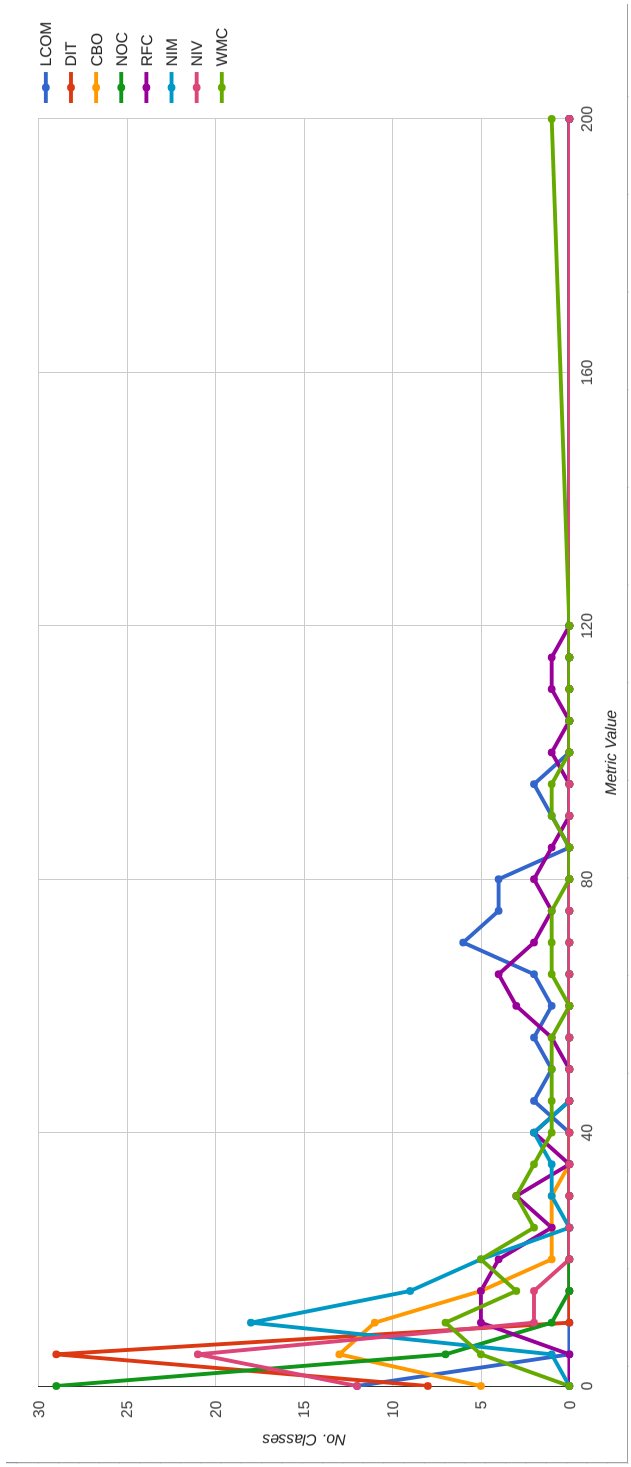


Figure 4.7: Frequency distribution of OO-metrics in Component A

Table 4.3: OO-metrics for Component B

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	22	91	66	63.348	15.916
DIT	0	1	1	0.609	0.499
CBO	1	21	5	6.13	4.799
NOC	0	1	0	0.087	0.288
RFC	6	48	9	14.217	11.977
NIM	6	48	9	13.434	10.974
NIV	1	10	2	3	2.504
WMC	3	325	19	35.826	66.796

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 90 percent, indicating none of the classes 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. Moreover, the WMC2 values ranges from 3 to 325. We decided to examine the class with a WMC2 value of 325. This class has a LCOM value of 74, and a CBO value of 10. Its RFC value is 44, while NIM and WMC is both 36. This class has no subclasses, but it does inherits methods and variables from one superclass. In terms of cyclomatic complexity, we believe that this class is the most complex class in this 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, WMC, and WMC2 values. Manual inspection of the classes did not reveal any duplicated code. However, in terms of refactoring, we do think that all these classes need the same effort.

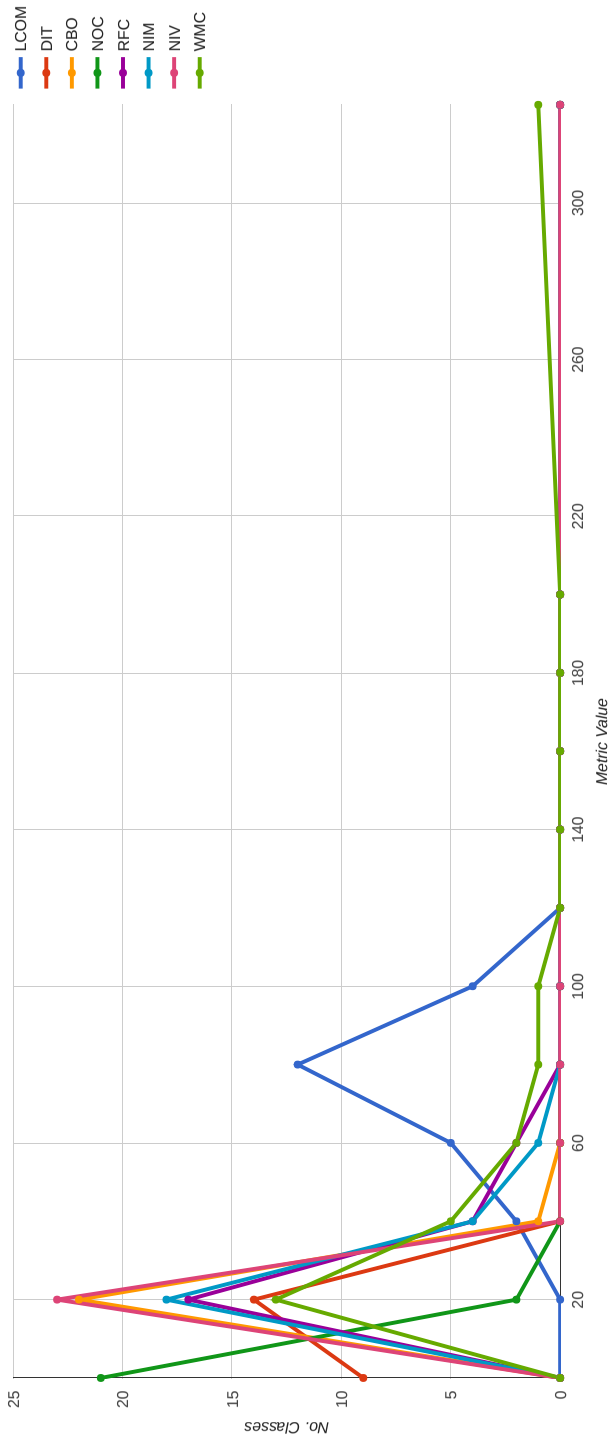


Figure 4.8: Frequency distribution of OO-metrics in Component B

Table 4.4: OO-metrics for Component C

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	89	61	54.8	22.139
DIT	0	1	0	0.1	0.308
CBO	1	19	4.5	6.5	5.587
NOC	0	1	0	0.05	0.223
RFC	3	26	8.5	10.3	5.741
NIM	3	26	8.5	9.85	5.153
NIV	0	9	2	3.15	3.013
WMC2	3	106	12.5	27.65	30.975

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 99. The median shows 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 not be used. Moreover, CBO values ranges from 1 to 18. Each class has a CBO value of 5 in average. The results show that there is one class with 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. WMC2 metric values ranges from 2 to 106. The median value implies that half of the classes has a complexity value of 9, implying that complexity is well managed for those classes. However, the maximum value tells us that there is one complex class in this component. This particular class has a LCOM value of 63, and is coupled to 18 other objects, hence being the class with CBO value of 18. RFC, WMC, and NIM values of this class is 26, all maximum values that we analyzed. All these values are an indication of a possible fault-prone and complex class. In addition, this class may be affected by the Large Class code smell.

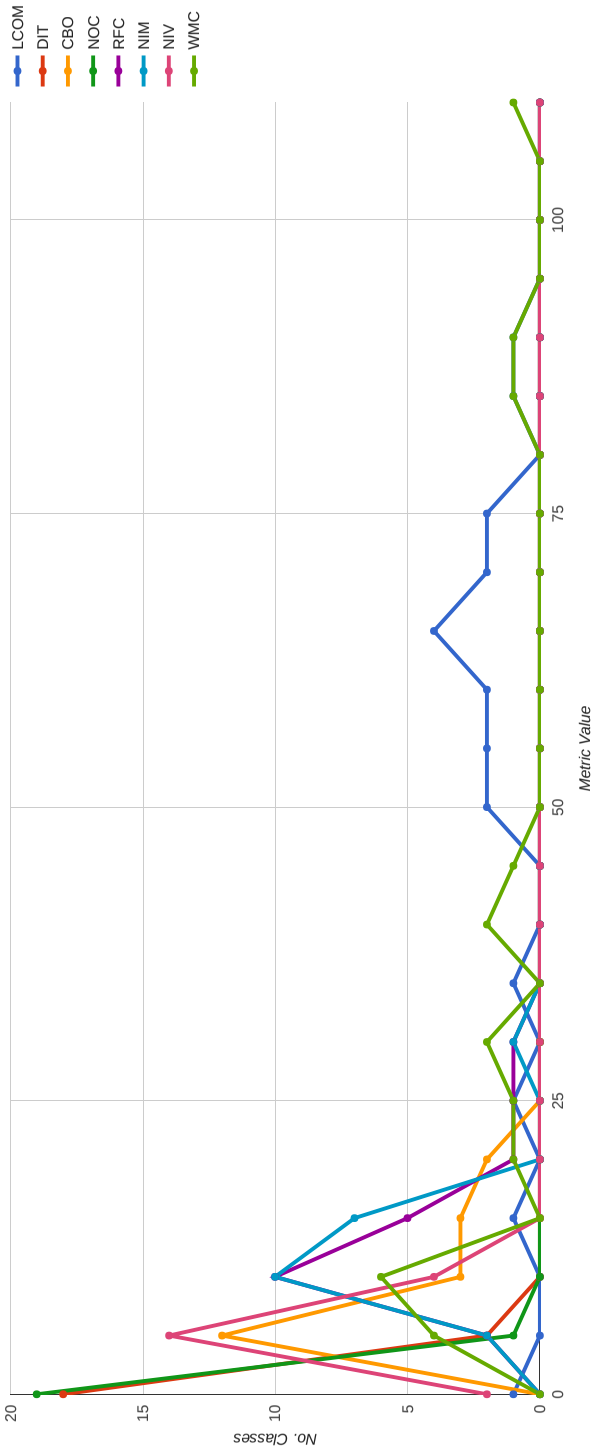


Figure 4.9: Coming

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, which tells that the class is not very cohesive. Furthermore, our result show a DIT value of 1, an NOC value of 0, indicating that this class only inherits from a superclass. The CBO value is set to 10, implying that this class is coupled to 10 objects. The values of RFC, WMC, and NIM have a value of 8. Moreover, the class however has only 1 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 in we identified in this 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.

The values of LCOM ranges from 0 to 100, indicating that there are classes with low and high cohesion. The median value of LCOM is 0, implying that half of the classes in this component has high cohesion. Even though half of the classes has high cohesion, there are still many classes with low cohesion. Our results show that there are 22 classes with LCOM value larger than 50, where 6 classes has LCOM value of 80 or more. There are only 2 classes with LCOM value of 100. Moreover, the average cyclomatic complexity of the classes has a value of 8.7, while the median has a value of 7. These values implies 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 WMC2 larger than 20, where the maximum value is 41. These values indicates that the complexity is well managed to this point. By examining the CBO values, we observe that only 7 classes are self-contained. The 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 be difficult to understand and maintain. The result show a RFC range from 0 to 28, with more than 50% of the classes having RFC value of 10 or less.

Table 4.5: OO-metrics for Component Ex

Metric	**Min**	**Max**	**Median**	**Sample Mean**	**Standard Deviation**
LCOM	0	100	0	26.302	33.082
DIT	0	3	2	1.581	1.121
CBO	0	16	4	5.314	4.459
NOC	0	20	0	0.744	2.736
RFC	0	28	8	10.279	6.030
NIM	0	22	3	4.907	3.846
NIV	0	10	0	1.209	2.098
WMC	0	41	7	8.744	8.117

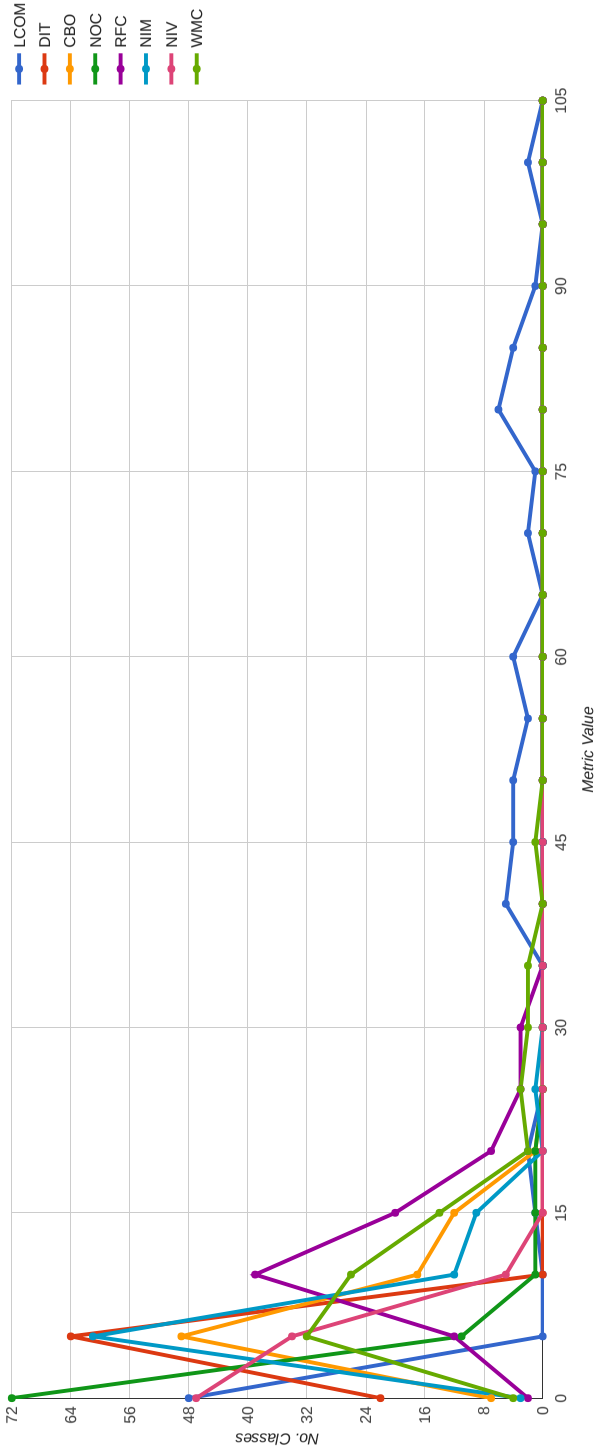


Figure 4.10: Coming

Table 4.6: Component G

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	94	60	50.25	31.236
DIT	0	2	1	0.625	0.609
CBO	0	23	5.5	6.312	4.987
NOC	0	2	0	0.25	0.622
RFC	2	30	9	10.187	6.382
NIM	0	29	7	8.437	5.459
NIV	0	18	2	3.062	3.926
WMC	1	123	12	19.437	24.794

Component G

Component G consists of 59 files. These files includes 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 can be seen in Figure 4.11.

Overall, the statistics are indicating 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 18 classes have a DIT value of larger than 0, indicating a higher degree of reuse. There are only 5 classes with subclasses in this component. WMC2 values in this component ranges from 1 to 123, where only class has a value of WMC2 larger than 100. We decided to examine the class with WMC2 value of 123. The results show that this class has a LCOM value of 92, CBO value of 22, RFC value of 30, NIM value of 9, NIV value of 18, and WMC value of 30. These values say that this class is probably influenced by the Large Class code smell, and is a candidate for inspection and refactoring.

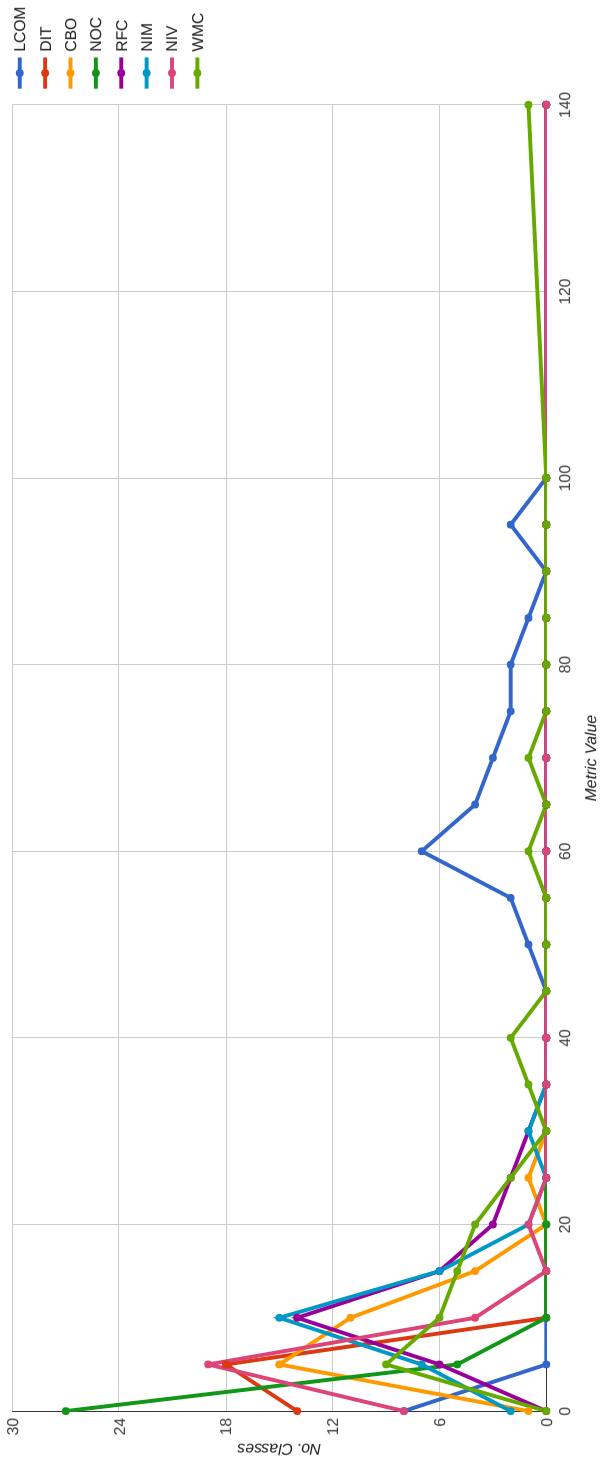


Figure 4.11: Coming

Table 4.7: OO-metrics for Component L

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	80	68	50.857	35.130
DIT	0	1	1	0.571	0.534
CBO	1	14	6	6.714	4.609
NOC	0	0	0	0	0
RFC	5	12	9	8.571	2.936
NIM	3	12	9	8.286	3.402
NIV	0	5	1	1.571	2.070
WMC	4	42	11	19.571	14.524

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 should eventually be split up into multiple classes. Moreover, 4 classes have a DIT value of 1. WMC2 values ranges from 4 to 42. There are only 2 classes with WMC2 values larger than 30.

Figure 4.12: Coming

Table 4.8: OO-metrics for Component N

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	79	70.5	54.5	33.899
DIT	0	1	0	0.25	0.463
CBO	3	19	12.5	11.5	4.536
NOC	0	1	0	0.125	0.353
RFC	6	32	9	11.625	8.568
NIM	6	21	8.5	9.75	5.036
NIV	0	8	5.5	4.375	3.068
WMC	4	125	32	40.375	36.707

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 show 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 WMC2 value of 125, RFC value of 32, CBO value of 17, WMC value of 23 and NIM value of 21. These values tells us that this class may be affected by Large Class and God Class code smell.

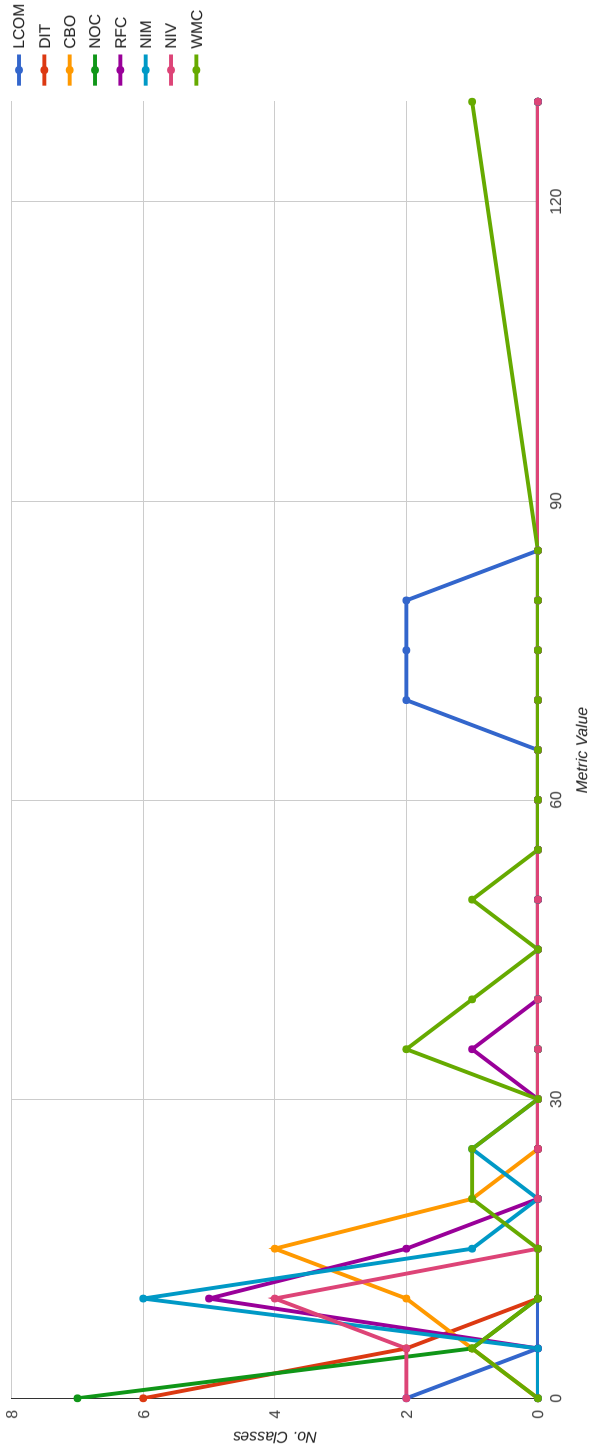


Figure 4.13: Coming

Table 4.9: OO-metrics for Component P

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	81	63	57.25	25.409
DIT	0	1	0	0.25	0.463
CBO	0	13	6	6	4.472
NOC	0	1	0	0.125	0.353
RFC	2	14	6.5	7.375	3.502
NIM	2	13	6.5	6.875	3.044
NIV	0	6	3.5	3.125	2.031
WMC	1	47	8	15.75	15.809

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 lies between a range from zero percent to eight-one percent. A median value shows the level of cohesiveness in the system. In this context, median value show 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 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 lies between a range from zero to twelve, with a mean and median of 5.5. In addition, WMC2 values lies between a range from 1 to 47, where two classes has WMC2 values larger than 30. In addition, we were able to identify a complex class, possible a God Class code smell. This class has a LCOM value of 77, WMC2 value of 47, and DIT value of 1. Moreover, the class has 13 methods, and RFC value of 14.

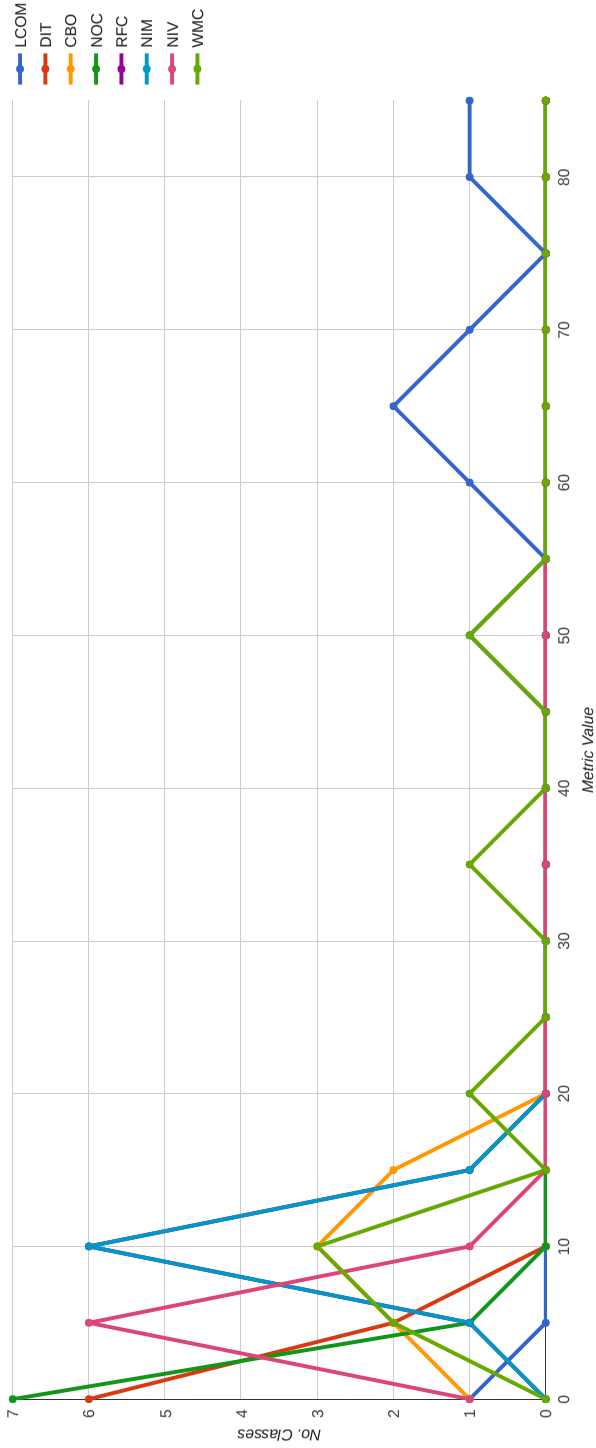


Figure 4.14: Coming

Table 4.10: OO-metrics for Component S

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

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 points 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, WMC2 metric values indicate that 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. 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, which indicates that this class is not complex.

4.2 Identifying Code Smells using Automatic Approaches

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 smells in the source code. Code smells are an indicator of software design flaws in object-oriented systems that can decrease software maintainability which may lead to

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

issues in further evolution of the system [36]. 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 considers a Long Method as code smell if lines of code in method exceeds 200 lines. Using Understand, we identified 10 long methods, spread across six different files. 7 of 10 long methods are located in test files.

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

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

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.

Dead Code

Fowler and Beck [56] 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 [35]. We examined three types of "Dead Code" code smell in Project Firmus: "Commented Out" Code, Unreachable Code, and Unnecessary Includes in Header

Table 4.14: Dead Code Results

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

Files. In total, we found 151 hits of ”Dead Code” code smell, which we have summarized in Table 4.14.

God Classes

CBO and LCOM can be useful to detect God Classes with support of LOC and WMC. (article: on the effectiveness of concern metrics to detect code smells: an empirical study)

A class with large values of WMC and RFC indicates many possible reponses since the class may have a large number of methods that can be executed. These values along with LCOM can be used to measure God Class code smell. By identiying

5 files

CHAPTER 5

DISCUSSION

This thesis aims to investigate the possibilities to identify design debt in embedded systems. Our case study evaluated a software system written in C/C++. This chapter will discuss the results we gathered during our case study.

5.1 Evaluation of Object-Oriented Metrics by Applying Threshold Values

Measuring software metrics in object-oriented software is important in terms of quality management [57, 58], as software metrics can be used as predictors of fault-prone classes in object-oriented systems [59]. A study by Basili et. al [59] assessed Chidamber and Kemeerers [44] 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 software life-cycle. Object-oriented measurements alone are not necessary sufficient to identify parts of the system with major design violations. According to Tarcisio, software metrics are not effectively used in software industry due to the fact that for the majority of metrics, thresholds are not defined [57]. Threshold is defined as values used to set ranges of desirable and undesirable metric values for measured software [58]. 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. [60] 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).

Threshold values were derived for the individual metrics using their descriptive statistics from Table 4.1 in order to identify the classes with major design violations. Threshold values have been identified using statistical information [60]. For each metric, we used the the sample mean and standard deviation from Table 4.1 to derive its corresponding threshold value. The first threshold value is corresponds with the mean and represents the most typical value in the data set [61]. According to Lanza et al. [60], the first threshold value can be calculated by subtracting the standard deviation from the sample mean. However, the standard deviation in our data set may be larger than the sample mean, which leads to negative threshold values. The second threshold value the sum of the sample mean and the standard deviation. It represents high, but still acceptable values. The third threshold value is simply the second threshold value multiplied with 1.5 [60]. It is considered as extreme, and should not be included in the data set. In addition, we have gathered thresholds that has been proposed by researchers for the metrics we have measured in this thesis. Table 5.1 presents the metrics and their threshold values.

Table 5.1: Thresholds for object-oriented software metrics

Metric	Observed Value	Low	High	Extreme value	Recommended Max Value
LCOM	100	42	75	100+	72.5 [57]
DIT	4	1	2	3+	4 [57], 5 [62–64]
CBO	30	6	11	17+	5 [62,64], 14 [65,66]
NOC	20	0	2	4+	3 [57], 5 [64], 10 [63]
RFC	115	15	34	51+	50 [62,63]
NIM	48	8	15	23+	
NIV	18	2	5	7+	
WMC	325	19	51	76+	34 [57], 40 [62], 50 [66]

Depth in Inheritance Tree

DIT indicates how deep a class is in the inheritance tree. It is evident that a deep inheritance makes software maintenance more difficult("SITER DALY et al. 1996"), as classes with higher value of DIT are associated with higher defects [67]. Moreover, higher degree of DIT indicates a trade-off between increased complexity and increase reuseability By following Chidamber and Kemerer's [44] guide to interpreting their 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 there is a majority of DIT values below 2, it may represent poor exploitation of the advantages of object-oriented design and inheritance, because a DIT value of 2 and 3 indicates higher degree of reuse. In Table 4.1, we observe that DIT median value for Project "Firmus" is 1. More precisely, approximately 39% of the classes have a DIT value of 0, while 26.7% of the classes have a DIT

value of 1. The 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 metrics.

Classes with high values of DIT have shown to be very significant in identifying fault-prone classes [59]. We derived the following threshold for the DIT metric to identify classes with high values of DIT: a low/good value of 1, a high/typical value of 2, and 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 Chidamber and Kemerer [44] considers as a good value. Therefore, we have chosen to apply the recommended max value of 5 seems as it has been recommended by other researchers. We were not able to identify any classes with a DIT value of 5 or more, but we identified 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 suggests that appropriate design preferences are being followed by the company [67].

Number Of Children

NOC measures the number of children a class has. In general, the greater number of children, the greater potential of reuse, since inheritance is a form of reuse. However, if a class have a large number of children, it may be a case of misuse of subclassing [59]. Higher degrees of NOC indicates increase in reuse, but in trade-off, the classes may require more testing. In Table 4.1, we observe that NOC median value is 0. The distribution of NOC metric shows that approximately 86.5% of all classes have no children, and that a small number of classes have many immediate subclasses. Both Chidamber and Kemerer [44], and Basili et al. [59] 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 leads developers not to reuse.

Following threshold value 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 extreme/bad value 4. These values are comparable to the thresholds that Filo et al. [57] have identified in their study. The difference is that they classify extreme/bad value as 4. By applying these thresholds, we observe that at 91.2% of the classes are classified as low/good, 5.2% of all classes are classified as high/typical, and 3.6% of all classes are classified as extreme/bad. Similar to DIT, these values indicate limitation in reuse opportunities in favor of comprehensibility of the overall architecture, which again suggest that appropriate design preferences for the system are being followed. These classes are an indication of something that may be hard to understand an maintain. These classes should be reviewed.

Lack of Cohesion in Methods

LCOM is related to the counting of methods using common attributes in a class. In general, smaller values of LCOM present cohesive and independent class, which is desirable since it promotes encapsulation. Larger values of LCOM does increase the complexity of the class, hence increasing the likelihood of errors during the development process. Our derived threshold values for LCOM metric reveals a low/good value 42, a high/typical value of 75, and an extreme/bad value of 100. When applying this threshold, we only identified two classes with an extreme/bad value. However, by analyzing the frequency distribution of LCOM values, we clearly see that the normal distribution of the data is not normal. This may be the reason for the derived threshold values. By comparing the recommended threshold value with our derived values, we notice that the recommended threshold value of 72.5 is very close that what we identified as high/typical value. By applying the recommended max threshold, we identified 41 classes with a LCOm value that is larger than 72.5. We do observe 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. In addition, Chidamber and Kemererer [44] and Basili et al. [59] state that classes with large values of LCOM could be more error prone, 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 indicates the extent of lack of reuse potential of a class, and that more effort is required to maintain and test the class. An analysis by revealed that classes higher values of CBO are associated with higher defects [67]. According to Chidamber and Kemererer [44], 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. Basili et al. [59] states that CBO appears to be useful to predict class fault-proneness. Our results revealed a CBO median value of 5, indicating that at least 50% of the classes refers to 5 or less object classes. The CBO value is generally less for most classes, hence most of these classes are easy to understand, reuse, and maintain. We did notice that some of these classes 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. Comparing our thresholds, we do think that the extreme/bad value may be a little bit high, especially when we compare the values with the

recommended max values. Moreover, C&K did classify a median value of 9 as high. Therefore, we chose to use high/typical value as our default threshold. Bay applying the threshold, we observe that 11.3% of all classes have a coupling of 12 or more. To promote encapsulation and improve modularity of a class, coupling between classes should be kept to a minimum. As the identified classes have high coupling values, they may be more sensitive to changes in other parts of the design, leading to maintenance problems. Moreover, it is likely that these classes are difficult to reuse in other parts of the system because the class is sensitive to change, and more complex to test. CBO 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 count includes all methods available in the class hierarchy. Larger values of RFC makes testing and debugging more complicated since it requires a greater level of understanding on the part o the tester [44], hence increasing 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 show that most classes tend to be able to invoke a small number of methods. 74.2% of the classes have a RFC value of less than 15.

We were able to derive the following threshold values for 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 [63]. However, it is recommended a default threshold from 0 to 50 for a class. By applying the thresholds on the captured metric for Project Firmus, we were able to identify 17 classes with RFC value larger than 50, with one class having a RFC value of 115. According to Chidamber et al. [68], RFC has been found to be highly correlated with WMC and CBO. However, our data reveal that RFC is not correlated with WMC and CBO. The class with a value of 115 had a CBO value of 2 and WMC value of 22. However, its DIT value is 4 may explain that large value of RFC comes from inherited methods. The same thing applies to the class with a RFC value of 109. The class have a CBO value of 3 and a WMC value of 10. However, the DIT value of this class is 4. These observation indicate that RFC is associated with DIT. Since there is a number of classes that have no parents, the RFC values also tend to be low.

Weighted Methods per Class

WMC refers to the sum of complexity in each method. A greater value of WMC indicates a complex class. A class may contain many methods, hence affecting

WMC value. If a class has many methods, some of the methods may have low complexity while other methods have high complexity. Our results reveal a median value of 10 and sample mean value of 19.707. The results are quite similar to what Chidamber and Kemerer [44] and Basili et al. [59] measured in their respective studies. Another aspect of our captured data is the similarity of the distribution of the metric values in their and our analyzed system. By looking at Figure X in Chapter 4, we notice that approximately 60 percent (i.e., 119 classes) have a WMC value of less than 10. This seem to suggest that most classes have a small number of methods. In addition, 7.4 percent (i.e., 17 classes) have value greater than 51. One of the classes in Component B revealed to have a WMC value of 325, which indicates that time and effort to develop and maintain this class may be high. This particular class has 44 methods and a CBO value of 10. A class with many methods are most likely application specific, hence reducing its reuse potential. Moreover, an increase in number of methods is associated with an increase in defects [67]. By examining the other classes, we noticed a trend where classes with more methods tend to have higher complexity. For instance, a class with a WMC value of 194 revealed to have 40 methods, suggesting an increase in defects as well.

We derived the following threshold values for WMC: low/good value of 19, high/typical value of 51, and extreme/bad value of 76. By comparing these values with the recommended max value, the practitioners seem to recommend a max value close to the value we regard as high/typical. Therefore, it seems like this value is more appropriate to use to identify the classes with design flaws. By applying the threshold, we identified 16 classes with a WMC higher than 51. As we mentioned earlier, high complexity are associated with an increase in defects, hence indicating poor design and in worst case, unmanageable. In such cases, maintenance effort increase drastically. The 16 classes we revealed with high complexity 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 [69–71]. 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 questions is related to the techniques we have used to identify design debt in this research. Both automatic static code analysis, and by measuring object-oriented metrics 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 thorough this research are Doxygen, Understand, SonarQube, SourceMonitor, 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 were 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. CppDependLong Parameter List code smell were detected using CppDepend.

Another approach to assessing the software quality is based on object-oriented metrics [72]. Object-oriented metrics can be used to identify design flaws, and defect-prone, change-prone, and fault-prone classes. In addition, object-oriented metrics affect the quality attributes of a system. For example, large values of WMC will affect a systems maintainability and reusability [43]. Object-oriented metrics are calculated over data that are extracted from the systems source code. The most well-known suite of software metrics for object-oriented systems is desribed by Chidamber and Kemerer. This suite of metrics have been applied in empirical investigations of object-oriented systems by multiple researchers, including Basili et al. [59], Chidamber and Kemerer [44], Okike [73], and Bakar et al. [74]. Moreover, we have applied this suite of object-oriented metrics to discover potential fault-prone classes.

Understand is an integrated development environment that enables static code analysis through visuals, documentation, and metric tools. The software is capable of analyzing projects with multiple lines. An academical license tool was provided, and the tool was used in our case study to compute software metrics. Each file in the system is 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. [75] calculated threshold values of object-oriented metrics by using statistical models. Understand was used to extract relevant metric data from one of the systems. Codabux et al. [72] extracted class-level metrics for defect- and change-prone classes using Understand.

By applying thresholds for object-oriented metrics, we were able to identify the classes with potential design flaws in which inspection is needed. 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. Based on the results, an alternative design structure can be applied, and refactoring can be applied to classes with larger metric value than its threshold. In other words, threshold values can be used to predict possible fault-prone classes that need to be inspected. For instance, a class with its WMC value larger than its threshold value will indicate high complexity for that particular class. Such 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. Basili et al. [59] states that WMC, DIT, NOC, CBO, and RFC are useful metrics to predict fault-prone classes.

RQ2: What are the effects of design debt?

The second research question is related to the consequences of having design flaws in embedded 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 [76,77]. Moreover, classes with large values of RFC may be harder to understand and test, hence affecting the software's understandability, testability, maintainability [76]. In addition, RFC may affect system's functionality and reusability as objects communicates by message passing [77]. LCOM measures the cohesion of a system. Lack of cohesion in a class increases is complexity, which ultimately leads to errors during development. This metric affects the systems efficiency, reusability, and understandability [76,77]. 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 [76,77]. Furthermore, the DIT and NOC metric are related to inheritance which enables reuse. Large values of DIT indicates deep hierarchy which constitute greater design complexity. Deep hierarchy enhances the potential reuse of inherited methods but in trade-off, complexity will increase which affects other quality attributes. In total, this metric evaluates efficiency, reusability, understandability, and testability [76]. In addition, DIT metric may also be related to flexibility, extendability, effectiveness, and functionality [77]. NOC primarily evaluates efficiency, testability, and

resuability of a system [76], but it may also influence flexibility, understandability, extendability, and effectiveness of a system [77].

Furthermore, code smells are manifestations of design flaws that can have negative influence on software 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 has discovered. Sjoberg et al. [78] investigated the relationship between 12 different code smells and maintenance effort. Their result revealed that none of these code smells were associated with more maintenance effort. Similarly, Hall et al. [79] 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. [80] 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. [81] and Dhillon et al. [82] state that bad smells are positively associated with increased error rate in software projects. Furthermore, Olbrich et al. [83] proved that God Class and Brain 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. [37] provided evidence that classes with specific code smells are more subject to change than others.

To summarize, 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.

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. During our 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. We were able to find Duplicated Code, Dead Code, Speculative Generality, and Long Method as the most common code smells in general. Furthermore, object-oriented metrics are useful for investigating each individual class and the software in general. For example, by examining the metrics, we identified possible large and god class code smells, indicating 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 refactoring and re-engineering. Codabux et al. [19] does 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 [84]. Without refactoring, the design of the program will decay over time.

Our results show that 5% of the source code, including the test files, contains duplicated code. In general, removing duplicated code will reduce the number of lines of code. Some duplicated code can found in the same class, while other are spread across multiple classes. Thus, they must be handled differently. In most of the cases, we identified the same block of code occurring 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 [34]. 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 will create 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 it should be invoked by the other class.

Moreover, we identified 10 long methods. To shorten a method, "Extract Method" can be applied by finding parts of the method that seem to go nicely together and make a new method [34]. 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 [34]. "Replace Temp with Query" extract 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. Long parameter list can be slimmed down with "Introduce Parameter Object" and "Preserve Whole Object" [34].

In addition, we identified 15 methods with Long Parameter List code smell, whereas 3 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 [34].

Our results found 1153 hits of Speculative Generality code smell, including unused methods, local variables, and static globals. Fowler et al. [34] 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, which reported 151 hits, including "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 applying refactoring. We believe that by applying 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. In some cases, refactoring is unlikely to reduce fault-proneness in classes, and may increase fault-proneness in a class instead [79], hence affecting 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. To reduce the maintenance effort, we do believe that improving the software metrics and other work practices may be better more beneficial than refactoring code smells.

5.3 Threats To Validity

Validity is related to how much the results can be trusted [8]. We consider threats to the external, internal, and construct validity of this study.

To validate this research, more experiments need to be done with different OO-languages on different systems. This study was carried out in an industrial context. A suggestion will be to carry the study out in multiple open-source systems and other industrial systems written in different systems. It would be interesting as well to compare different industrial systems within the same OO-language. 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.

5.3.2 External Validity

External validity refers to the degree to which the results from the study can be generalized to the population. The system investigated in this study consists of one simple size and an application domain, hence increasing the threat to external validity.

5.3.3 Construct Validity

5.3.4 Conclusion Validity

Conclusion validity refers to the degree in which correct conclusion can be drawn from the relationship between treatment and the outcome. Our case study consisted of studying one system, so in general, the statistical power is very low. Deeper studies needs to be performed to confirm is our results have more applicability.

CHAPTER 6

CONCLUSION

This research presents the result of a case study that has investigated design debt in embedded systems. The purpose of this study was to identify design debt in embedded systems by measuring object-oriented metrics and detecting code smells. The results show that **.

6.1 Future Work

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

- Measuring other object-oriented metrics
- Implementation of tools which can suggest refactoring options
- A deeper study on multiple release version of a system to compare the evolution of metrics and patterns over time.
- Study design pattern

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

[5] 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.

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

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

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] S. McConnell, "Managing technical debt," 2007. [Online; accessed 03-May-2016].

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

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

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

[47] 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.

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

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

[50] 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.

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

[52] 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.

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

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

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

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

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

[58] 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.

[59] 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.

[60] 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.

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

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

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

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

[65] 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.

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

[67] 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.

[68] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial use of metrics for object-oriented software: An exploratory analysis," *Software Engineering, IEEE Transactions on*, vol. 24, no. 8, pp. 629–639, 1998.

[69] 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.

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

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

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

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.