Abstract

Who needs an abstract?

Sammendrag

Hvem trenger et sammendrag?

Preface

Here is the preface

Acknowledgements

I would like to thank Ingar Kulbranstad from Autronica for giving me this geat oppoturnity.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 and Background

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(”SITER COMPOSE: A composite embedded software approach”). Software code often ends up not contributing to the mission of the original intended software architecture. 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 [2], 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"). Architectural technical debt is an example of a type of TD, which accumulates when compromises are made in software architecture level. Architecture plays a significant role in the development of large systems [3], and unlike code-level debt, architectural debt usually has more significant consequences [4].

This thesis builds upon our previous study "Managing Technical Debt in Embedded Systems" [5]. In our previous research, we wish to look deeper into a more narrow field by conducting a deeper case study with companies.

1.2 Research Design

The relevant research methods in software engineering can be survey, design and creation, case study, experimentation, action research, and ethnography [6]. In this study, literature review and case study have been used. Literature review was a part of the pre-study and has been used to get familiar with the term architectural technical 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 [7]. It can be both qualitative and quantitative [6, 8].

The main research questions investigated in this thesis are:

1. **RQ1**: How can architectural technical debt be identified?
2. **RQ2**: Why does architectural technical debt accumulate?
3. **RQ3**: What are the effects of architectural technical debt?

1.3 Contrubution

Contribution: New knowledge about architectural technical debt in embedded software. How it differs from existing research.

1.4 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 [9]. 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 [9]:

> *"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 [10]. 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 [11–13]. Unmanaged technical debt can cause projects to face significant technical and financial problems, which ultimately leads to increased maintenance and evolution costs [14].

2.1.1 Definitions of Technical Debt

Several researchers have attempted to give us a clear picture of what technical debt is [3, 15, 16]. Fowler [15] presents a technical debt quadrant which consists of two dimensions: *reckless/prudent* and *deliberate/inadvertent* [15]. 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.

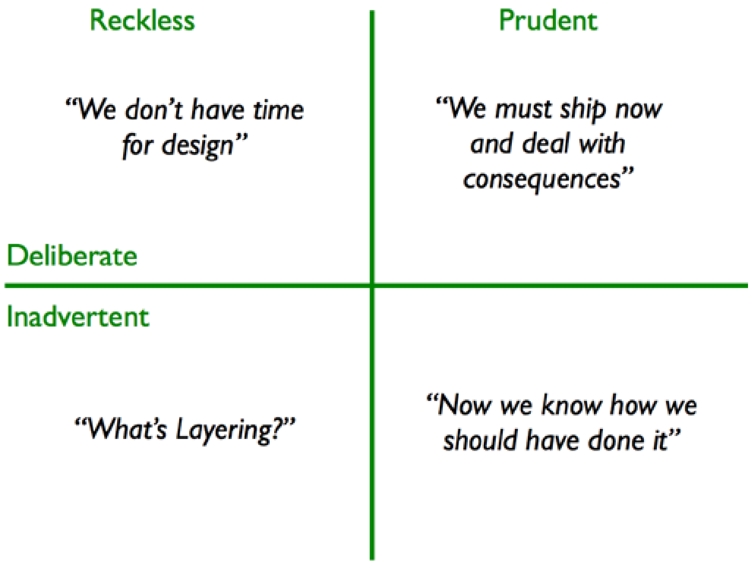
Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 2.1: Fowler's Technical Debt Quadrant

McConnell [16] 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 [17, 18]. 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. [3] 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.

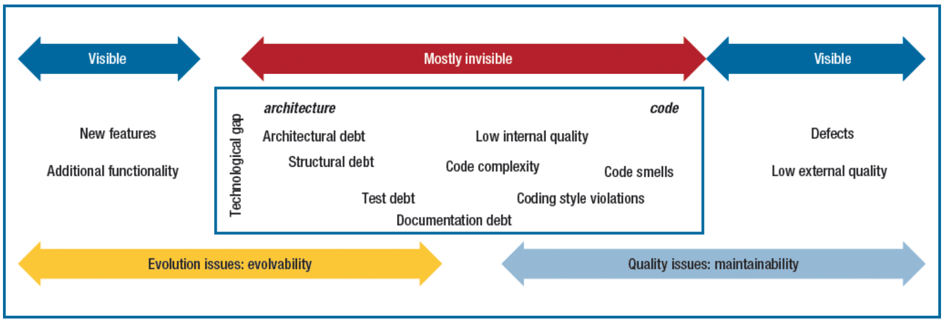

Figure 2.2: Technical Debt Landscape

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 [13, 17, 19–22] 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. [12] 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 [13, 17, 19]	Architectural decisions that make compromises in some of the quality attributes, such as modifiability.
Code debt [13, 19, 20]	Poorly written code that violates best coding practices and guidelines, such as code duplication.
Defect debt [13, 20]	Defect, failures, or bugs in the software.
Design debt [13, 19, 21]	Technical shortcuts that are taken in design.
Documentation debt [13, 19, 22]	Refers to insufficient, incomplete, or outdated documentation in any aspect of software development.
Infrastructure debt [13, 17, 20]	Refers to sub-optimal configuration of development-related processes, technologies, and supporting tools. An example is lack of continuous integration.
Requirements debt [13, 22]	Refers to the requirements that are not fully implemented, or the distance between actual requirements and implemented requirements.
Test debt [13, 19, 22]	Refers to shortcuts taken in testing. An example is lack of unit tests, and integration tests.

leverage [12]. 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. [23] 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. [24] 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. [25] carried out an industrial case study where they analyzed documents, emails, and code files. Additionally, they interviewed multiple developers and project managers. The case study revealed that technical debt were mainly taken by strategic decisions. Furthermore, they commented out that using a unique specialist could lead the development team to solutions that the specialist wanted and believe were correct, leading the team to incur debt. The study also identified that technical debt can both increase and decrease the amount of working hours.

Zazworka et al. [26] 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 [21]. 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 [27] 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. [17] 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 [24].

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. [28], 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 [29], 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 [28, 30]. This causes a violation in the software designed modular structure, which is called a modularity violation. Wong et al. [30] 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 [30]. Among these violations, 161 were confirmed. Zazworka et al. [28] 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. [31] 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 [31].

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 [28]. Fowler et al. [32] 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 [32]. 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 [32].

Mäntylä et al. [33] proposes a taxonomy based on the criteria on code smells defined by Fowler et al. [32]. 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. [32].

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. [34] 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. [35] 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 [36]. Travassos et al. [36] present a set of reading techniques that gives specific and practical guidance for identifying defects in Object-Oriented design. However, Marinescu [37] 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 [38]. 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 [38] 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 [37] 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 [39]. 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. [40] 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.

SOFTWARE METRICS Proposed as quality indicator for software systems. Rien defined 61 object-oriented design heuristics characterizing

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 Architectural Technical Debt

Bass et al. [41] defines software architecture as following:

> *The software architecture of a system is the set of structures needed to reason about the system, which compromise software elements, relations among them, and properties of both.*

The architecture of a software is one of the most important artifacts within the systems life cycle [41,42]. Architectural design decisions that are made during the design phase, affect the systems ability to accept changes and to adapt to changing market requirements in the future. As the design decisions are made early, it will directly affect the evolution and maintenance phase [43], activities that consumes a big part of the systems lifespan [44]. Design changes are specifically related to the adaptive and corrective categories. The issues of software architecture has long been a concern for those building and evolving large software systems [45].

Architectural technical debt is regarded as implemented solutions that are sub-optimal with respect to the quality attributes defined in the desired architecture.

Architecture plays a significant role in the development of large systems [3].

Examples

2.2.1 Dependencies

2.3 Software Quality

2.4 Patterns

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 [46]. Lehman [47] introduced the study of software evolution. Software evolution is a process that usually takes place when the initial development of a software

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	

project is done and was successful [48]. 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 [46]. 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* [49]. Maintenance can be classified into four types [48, 49]:

- 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 [44] 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 [44]. 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.

2.6 Software Reuse

2.7 Refactoring

2.8 Embedded Systems

2.8.1 Component Software

CHAPTER 3 RESEARCH METHODOLOGY

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 term research design. Section 3.3 presents the research questions that are to be answered, as well as the research design that are used to acquire the research data.

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 [8]. 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 [7].

There are two types of research paradigms that have different approaches to empirical studies [7]; the qualitative, and the quantitative paradigm. Qualitative research is concerned with studying objects in their natural setting [7]. 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 [7]. 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 [6] 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, and our previous research, we wanted to conduct an exploratory case study to get deeper insight about the problem.

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 [7]. 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 [7, 52]. Moreover, Runeson et al. [53] 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 [52, 53]. According to Yin [54], 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 [54]. Soy [52] proposes six steps that can be used when carrying out a case study:

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

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

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

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

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

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

3.3 Case Context

We had the opportunity to work with Autronica.

AutroSafe is a high-end distributed fire alarm system. It was first released around year 2000. The software that is used in AutroSafe is written in C/C++. 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 replace the existing software by using newer technologies. It is developed by Autronica Fire and Security. Their main office is located in Trondheim, Norway.

In this research, we have studied one system. This is a pilot project by the company with the purpose on replacing an existing system. The goal is to identify the technical debt before it gets worse. System description, language, commercial, usage, context. How it was selected.

To study the consequences of technical debt, we have chosen to study a commercial system. One of the disadvantages using commercial is the constraints associated with obtaining permission to mine and publish findings. This system is developed using C/C++, which makes data mining a challenge due to most tools available for data mining in this context operate on Java systems.

Summary of system: Written in C/C++ and has been in development since late 1990s. It is used widely on their products. We have studied a pilot project which goal is to use better design, technologies etc. This is a pilot project so it has not entered the evolution. Our goal is to identify technical debt that has accumulated recently, and we'd like to discover them before they get messy.

Tools that we have used: Various tools are used to mine data and understand the structure of the system studied. Eclipse plugins, Doxygen, etc. AltovaUML? A commercial suite of tools. UML design tool with the added functionality to produce UML diagrams from code.

How big the system is etc.

Lines: 88546 Lines of Code: 49287 Files: 461 Number of Packages Number of Classes Number of Interfaces Number of Methods Average Lines of Code per Class Developers on this project

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 [52], combined with the research process suggested by Oates [6].

Figure X illustrates the research process that has been used through this thesis.

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 [5], we are interested in getting deeper insight into the field of technical debt. As mentioned in Section 2.1.2, many subcategories of technical debt exists. With regards to that, we have chosen to investigate design debt in embedded systems.

Table 3.1: Research Questions

RQ1	How can design debt be identified?
RQ2	Why does design debt accumulate?
RQ3	What are the effects of design debt?
RQ4	What kind of design debt can be found in embedded systems?
RQ5	How to pay design debt?

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 this topic, along with our experiences from our previous research. These research questions will be out primarily driving force through this research. We have defined three research questions RQ1-3 which are summarized in Table 3.1.

Literature Review

To define the research questions, and to design the case study, it was necessary to conduct a literature review.

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. We had the opportunity to wok with Autronica Fire and Security in this study. "Firmus" is the project name for the next generation of AutroSafe. The main goal of this project is to replace the existing software that is used by AutroSafe today by adopting todays standards.

A company located in Trondheim were interested in this case study and collaborated with us by offering us a system to study. The case study was conducted on a system developed in C/C++., which we have briefly described in Section XX. Furthermore, the company provided us a workspace and multiple data sources. We had access to their source code, issue lists in Stash, system requirements, design and code documentation. Our data were mainly extracted from these sources using various tools. The system consists of multiple components. Due to our time limit, we focused on three of the biggest components in the system.

A part of the literature review was to get familiar with existing tools that has been used to address similar problems. In Section XX, 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.

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.

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 [55]. 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 class hotspots and design pattern realizations.

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

SourceMonitor

Metrics Selection

The metrics selected

Class OO Metrics:

LCOM (Lack of Cohesion in Percent): 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.

DIT (Max Depth Inheritance Tree): Maximum depth of the class in the inheritance tree.

CBO (Count of Coupled Classes): Number of other classes coupled to this class. Desiriable value is lower.

NOC (Number of Children): Number of subclasses this class has.

RFC (Count of All Methods): Number of methods this class has, including inherited methods.

NIM (Count of Instance Methods): Number of instance methods this class has.

NIV (Count of Instance Variables): Number of instnace variables in this class.

WMC (Count of Methods): Number of local methods in this class. More member of functions is considered to be more complex and therefore more error prone. Desirable value is low

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 multiple sources by using different tools to improve the reliability of the study. The first source of design flaws it to look for code smells in the source. Table 2.2 in Section 2.1.4 lists the code smells that are presented by Fowler et al. [32]. We have identified the different code smells by using various static analysis tools and manual confirmation by looking at class and dependency diagrams. For instance, code duplication data were extracted using SonarQube, and long methods were identified using CppDepend and Understand. Furthermore, UML class diagrams were generated by reverse engineering the source code to verify some of the results. Doxygen was not able to provide full class diagrams or dependency graphs for the system, so we had to look for other tools. 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 classes and their corresponding relations. This led us to look for commercial software. Using Understand, we were able to extract internal dependencies in each component, and dependencies between components. We noticed two form of dependencies, direct dependencies and circular dependencies. In addition, Understand was able to generate UML class diagrams, which were used to confirm complex classes and interfaces.

The second source is to analyze the various software and object-oriented metrics. Metrics is used to manage, predict, and improve the quality of a software product [56]. Using tools, we have extracted multiple metrics. Project Summary metrics summarizes the system by counting lines, classes etc. File metrics looks at each file. Class metrics looks at object-oriented design metrics.

The third source of evidence are interviews to validate the results from first and second source, done at the end of the case study. The developers are interviewed using a semi-structured interview. This type of interview allow us to improvise and explore the course of the interview. The interview is recorded and written notes are taken.

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. Validations and confirmations of the data needs to be done in a systematic way, and confirmations is needed to draw some valuable and conclusions about them.

We analyze the data by looking at the extracted data from the tools manually. Furthermore, we plot the findings into a word document. This was done to make the findings more accessible, and to interpret the data afterwards. The results are then compared with our findings from the literature review.

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 Metrics

4.1.1 Traditional Metrics

Table 4.1 summarizes the measurements over the project as a whole from the different tools. The results from the different tools reveals some differences in the measurements. For example, CCCC did not identify number of files in the project, but instead it identified number of non-trivial modules in the project. Non-trivial modules include all classes, and any other module for which member functions are identified. Furthermore, SonarQube identified 851 classes in the project. We can see that the difference between the results of amount of classes from SourceMonitor/Understand and SonarQube is big. In addition, SonarQube identified a total of 45153 statements in the source code, while Understand identified 35254 statements, which is pretty close to what SourceMonitor identified.

Using CCCC, we were able to measure information flow between the different modules. This is done by identifying and counting inter-module couplings in the module interfaces.

Moreover, we gathered two types of metrics: Class metrics and file metrics. File

Table 4.1: Project Summary

	CCCC	**SonarQube**	**Understand**
Lines		88404	88546
Lines of Code (LOC)	46220	48693	49287
Number of Files (NOF)		461	461
Number of Modules (NOM)	459		
Classes		851	830
Functions		3111	3589
Statements		45153	30109 (executabl
Comments (COM)	19705	15962	23017
Information Flow Measure (inclusive) (IF4)	171493		
Information Flow Measure (visible) (IF4v)	166589		
Information Flow Measure (concrete) (IF4c)	3789		

metrics returns software quality metrics for the specific file, while class metrics contains metrics for each class. A file can contain multiple classes. We wish to look at the different metrics on each file to indicate if there is any weaknesses in that particular file and what classes the file contains. The metrics we measured are lines of code, complexity of file, average complexity per function, amount of functions, and max depth.

SonarQube class metric includes nested classes, enums, interfaces, and annotations. Understand and SourceMonitor class metric includes classes and struct keyword.

SonarQube has more accurate data on classes and functions, while SourceMonitor has more accurate data about statements.

4.1.2 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. [58]. They have proposed a set of static metrics that are designed to measure the quality of object-oriented software. These metrics are widely known, and their metrics suite is the deepest research in object-oriented metrics investigation and the measurements we have are the following: Weighted Method per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Lack of Cohesion in Methods (LCOM), Response For a Class (RFC), and Coupling between Object Classes (CBO). In addition to these metrics, we have chosen to count the number of instance variables and instance methods in each class. These values can help us identifying Large Class code smell.

Table 4.2: OO-metrics for Project Firmus

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	100	55	42.205	33.042
DIT	0	4	1	1.061	1.062
CBO	0	30	5	6.079	5.179
NOC	0	20	0	0.454	1.850
RFC	0	115	10	15.777	18.677
WMC	0	48	7	8.616	7.167
NIM	0	48	7	8.376	6.983
NIV	0	18	1	2.223	2.811

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

4.1.3 Object-Oriented Metrics for the whole Project

We have decided to exclude the tests from object-oriented metrics analysis. A total of 321 files were analyzed. These files contains 229 classes, and 32068 lines of code. Descriptive statistics such as minimum, maximum, median, sample mean, and standard deviation are presented in this section. Table 4.2 presents descriptive statistics for class level metrics for the whole project.

TODO: Find X% of the total classes with high coupling, low cohesion, deep hierarchy.

LCOM: A class is cohesive if LCOM is low. The median shows that more than 50% of the classes have low cohesion. Classes with low cohesion increases the complexity of the software, and may therefore increase the likelihood of errors during development.

DIT and NOC: DIT value is generally low in the captured statistics. A class with DIT = 0 is the root of a class hierarchy. With an average value of 1, more than half of the classes inherits from a superclass. However, the the max value of DIT indicates some classes with deep hierarchy. Both samle mean of NOC and DIT are pretty low. They show that inheritance is used in most of the classes to an optimal level. This depth level is well managed at this point, and it probably comes from inheritance. However, the median value of NOC points out that that approximately half of the classes have a flat class structure, indicating inheritance may not be used enough. Moreover, the max value of NOC is very large. 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.

CBO: In general, higher values of CBO indicates fault prone classes. Both sample mean and median shows low CBO values for over half of the classes in this system. However, the maximum value that has been captured is very large. This class is an example of a class that is hard to understand, harder to reuse, and more difficult to maintain.

RFC and WMC: The RFC statistics reveals that most classes have a RFC of less than 10. However, the maximum value is revealed to be 115. Classes with large RFC tends to be complex and have decreased understandability. Testing classes with large RFC is more complicated. In addition, most of the classes have a WMC of less than 7, but there are a few classes with more extreme values. Those classes with highest WMC are candidates for inspection and refactoring.

NIM and NIV: NIM and NIV can help us identify Large Class code smell. The results show that most classes are small. The sample mean of NIV tells us that each class has an average of 2 instance variables. However, the max value reported 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. The maximum value is 48, which indicates that at least one class contains 48 methods. These values can help us identifying Large Class code smell.

4.1.4 Object-Oriented Metrics for the Components

Descriptive statistics in Table 4.2 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 56 files. Among these files, we identified 40 classes and 6286 lines of code. Figure X shows the distribution of the metrics.

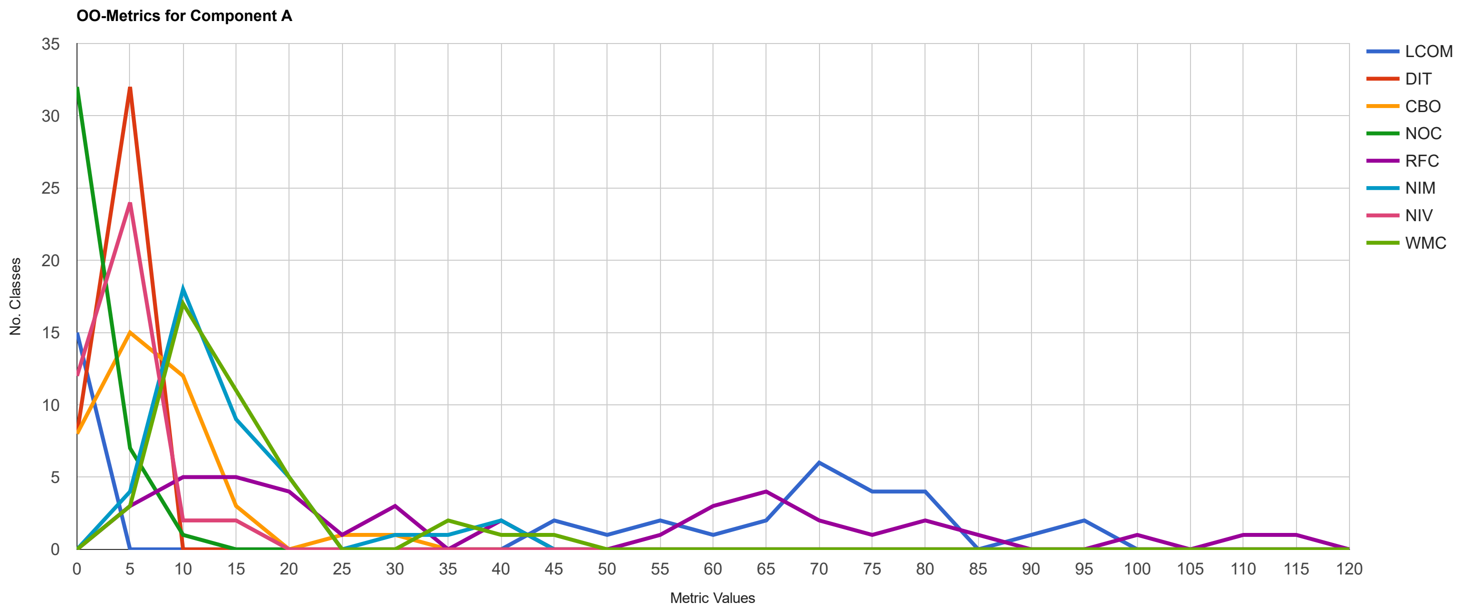

Figure 4.1: Frequency distribution of OO-metrics in Component A

Table 4.3: OO-metrics for component A

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	94	57	42.925	35.222
DIT	0	4	1	1.525	1.132
CBO	0	29	5	5.875	6.252
NOC	0	8	0	0.7	1.652
RFC	2	115	28.5	40.525	32.252
WMC	2	44	10.5	12.675	9.339
NIM	2	40	10	12.3	8.979
NIV	0	12	1	2.4	2.889

In Table 4.3, we present descriptive statistics for Component A.

Among the 40 classes, over half of the classes has low cohesion.

DIT and NOC is very low. CBO has average values, but one of the classes has a maximum value of 8. RFC median shows, but the average value is higher than the median. This indicates that some classes have high RFC values, hence increasing the sample mean. WMC is set to be low, but some classes has high values. NIM and NIV is moderate.

Component B

Excluding the tests, we counted 42 files containing 23 classes and 3905 lines of code. Descriptive statistics for Component B are presented in Table 4.3.

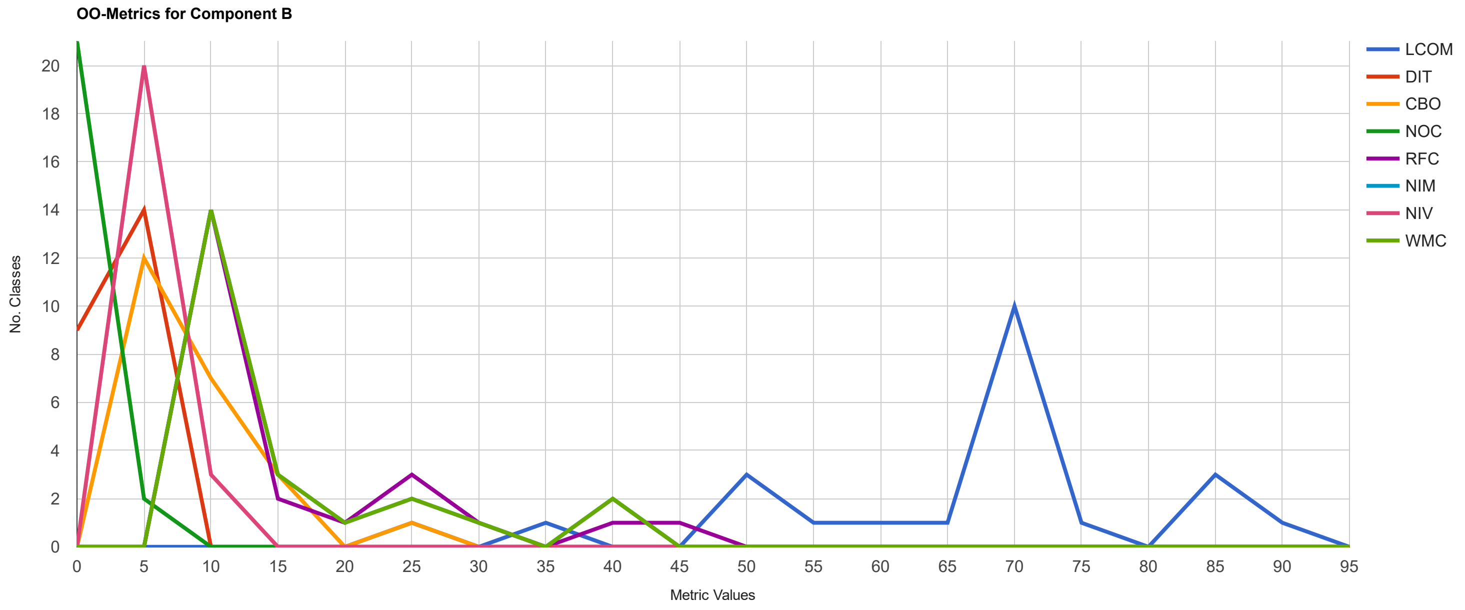
OO-Metrics for Component B
No. Classes
Metric Values
0
2
4
6
8
10
12
14
16
18
20
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
LCOM
DIT
CBO
NOC
RFC
NIM
NIV
WMC

Table 4.4: OO-metrics for Component B

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	100	55	42.205	33.042
DIT	0	4	1	1.061	1.061
CBO	0	30	5	6.079	5.179
NOC	0	20	0	0.454	1.850
RFC	0	115	10	15.777	18.677
WMC	0	48	7	8.616	7.156
NIM	0	48	7	8.375	6.983
NIV	0	86	1	2.222	2.811

LCOM is very high. Similar to Component A, half of the classes have low cohesion.

DIT and NOC has a very low sample mean and median. However, max NOC value is 20.

RFC has average values, but max is very high.

CBO is average, but max value is set to 30.

WMC is average, but max is high.

NIM and NIV.

Component C

Component C contains 30 files, 20 classes, and 4763 lines of code.

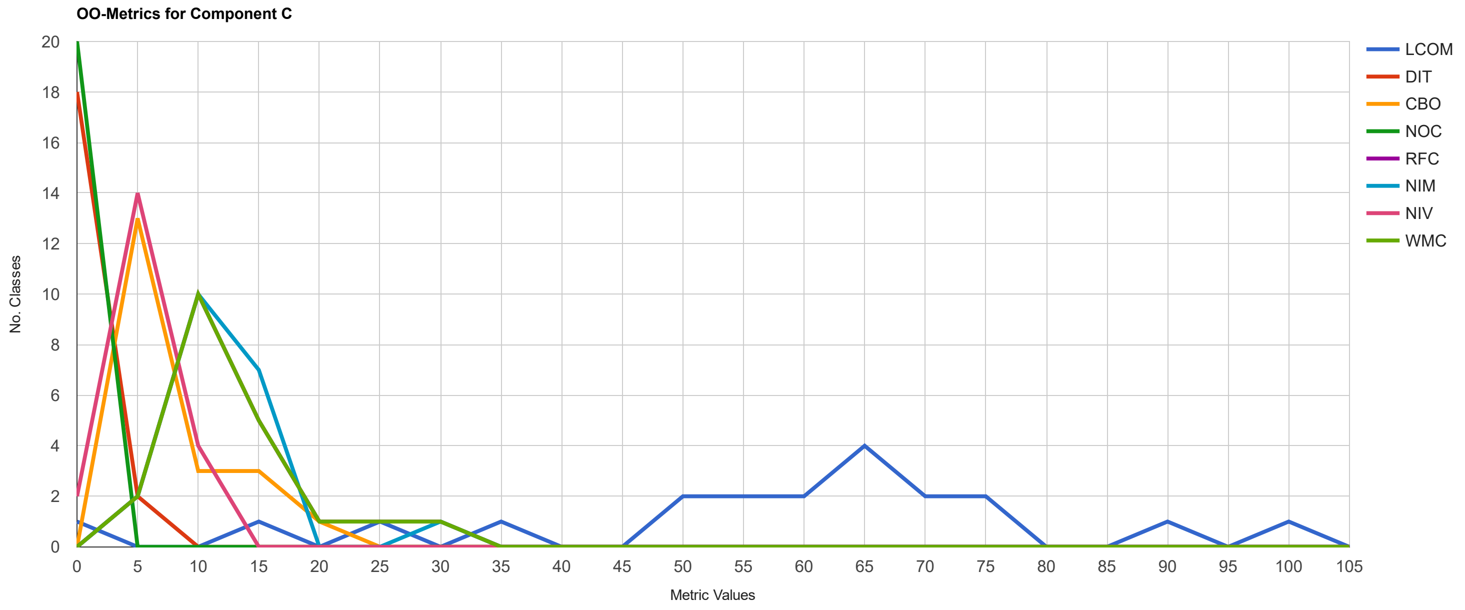
OO-Metrics for Component C
No. Classes
20
18
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
Metric Values
LCOM
DIT
CBO
NOC
RFC
NIM
NIV
WMC

Table 4.5: OO-metrics for Component C

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	99	61	55.7	23.58
DIT	0	1	0	0.1	0.308
CBO	1	18	4	5.55	4.662
NOC	0	0	0	0	0
RFC	3	26	8.5	10.3	5.741
WMC	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

LCOM median shows that half of the classes has more than 60% LCOM.

DIT and NOC is very low. Inheritance is not used that much.

CBO has moderate values, but maximum value is relatively high.

RFC moderate values.

WMC: Max is a bit far away from sample mean and average.

NIM and NIV

Component D

Component D has 13 files with 1647 lines of code. Among these, we found one class.

Component En

3 files, 367 lines of code. Only 1 class.

Component Ex

48 files, 4089 lines of code. 86 classes.

Table 4.6: OO-metrics for Component D

Metric	**Min**	**Max**	**Median**	**Sample Mean**	**Standard Deviation**
LCOM	68	68	-	68	-
DIT	1	1	-	1	-
CBO	7	7	-	7	-
NOC	0	0	-	0	-
RFC	8	8	-	8	-
WMC	8	8	-	8	-
NIM	8	8	-	8	-
NIV	2	2	-	2	-

Table 4.7: OO-metrics for Component En

Metric	**Min**	**Max**	**Median**	**Sample Mean**	**Standard Deviation**
LCOM	62	62	-	62	-
DIT	0	0	-	0	-
CBO	1	1	-	1	-
NOC	0	0	-	0	-
RFC	8	8	-	8	-
WMC	8	8	-	8	-
NIM	8	8	-	8	-
NIV	2	2	-	2	-

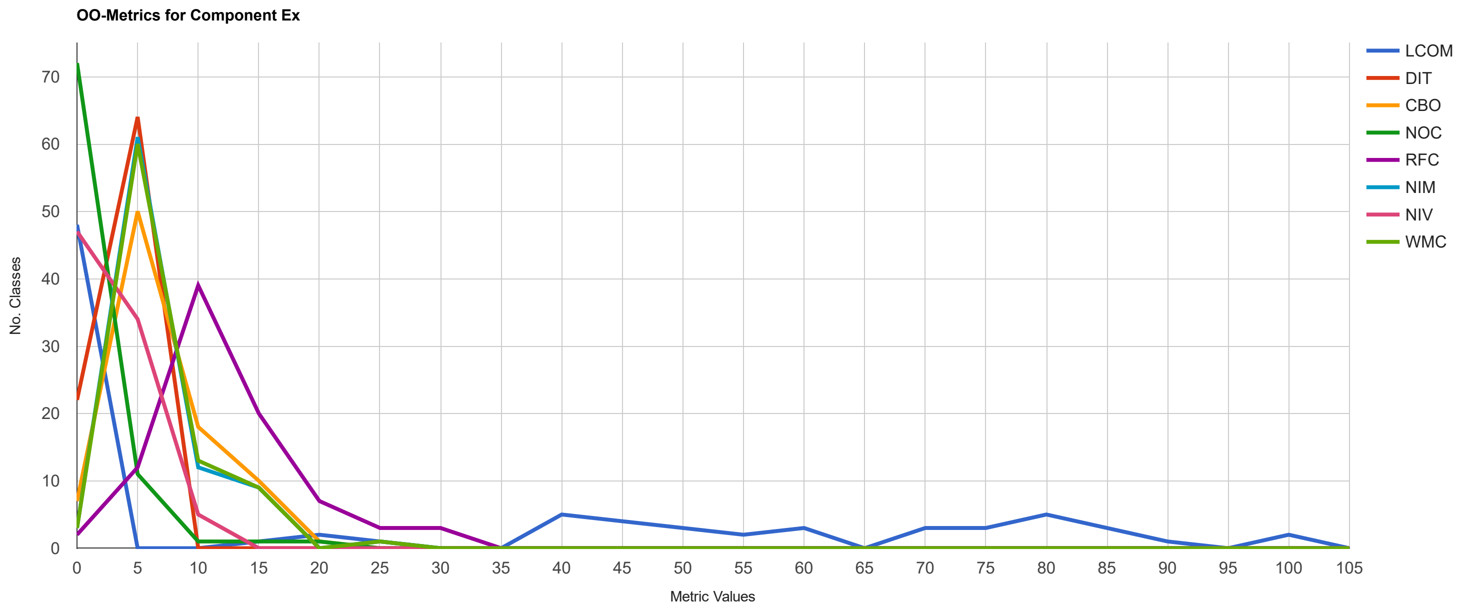
OO-Metrics for Component Ex
No. Classes
Metric Values
0
10
20
30
40
50
60
70
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
LCOM
DIT
CBO
NOC
RFC
NIM
NIV
WMC

Table 4.8: OO-metrics for Component Ex

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	100	0	25.988	32.905
DIT	0	3	2	1.581	1.121
CBO	0	16	4	4.919	4.018
NOC	0	20	0	0.744	2.736
RFC	0	28	8	10.279	6.030
WMC	0	22	3.5	5.07	3.928
NIM	0	22	3	4.907	3.846
NIV	0	10	0	1.209	2.098

Component G

59 files, 3701 lines of code, 32 classes.

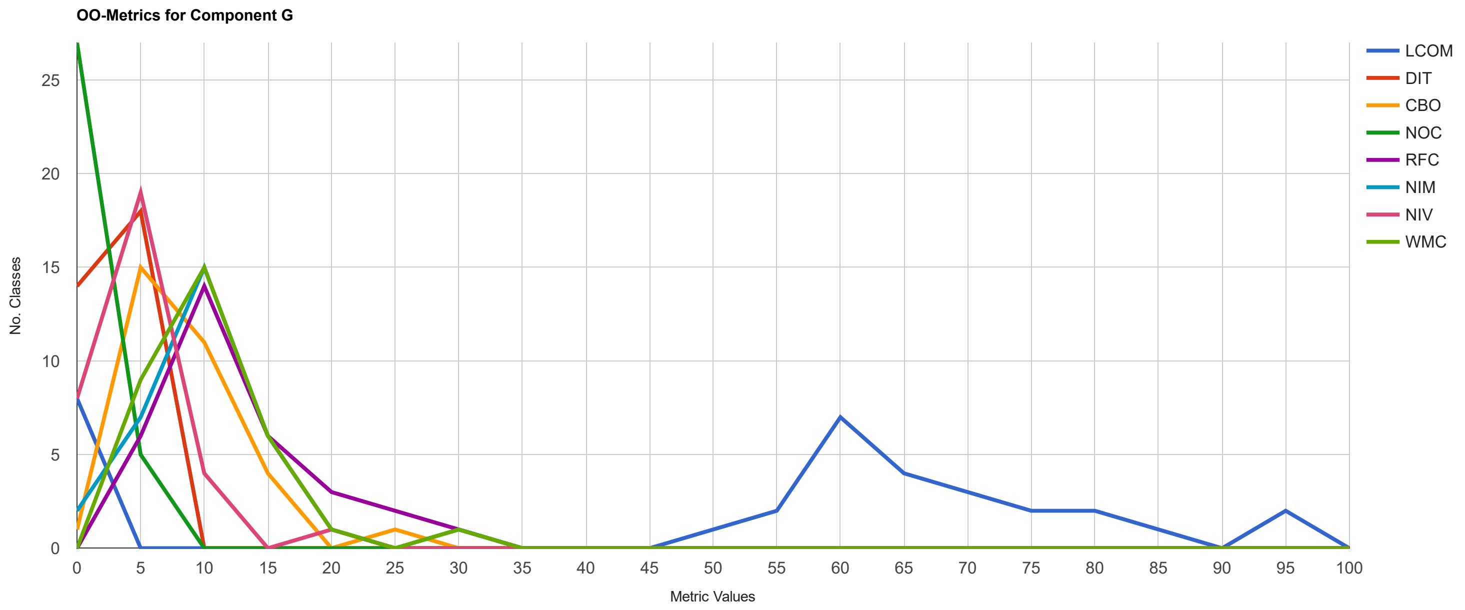
OO-Metrics for Component G
No. Classes
Metric Values
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
LCOM
DIT
CBO
NOC
RFC
NIM
NIV
WMC

Table 4.9: 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	22	5.5	6.187	4.987
NOC	0	2	0	0.25	0.622
RFC	2	30	9	10.187	6.382
WMC	2	30	7.5	8.594	5.405
NIM	0	29	7	8.437	5.459
NIV	0	18	2	3.062	3.926

Component L

16 files, 849 lines of code, 7 classes.

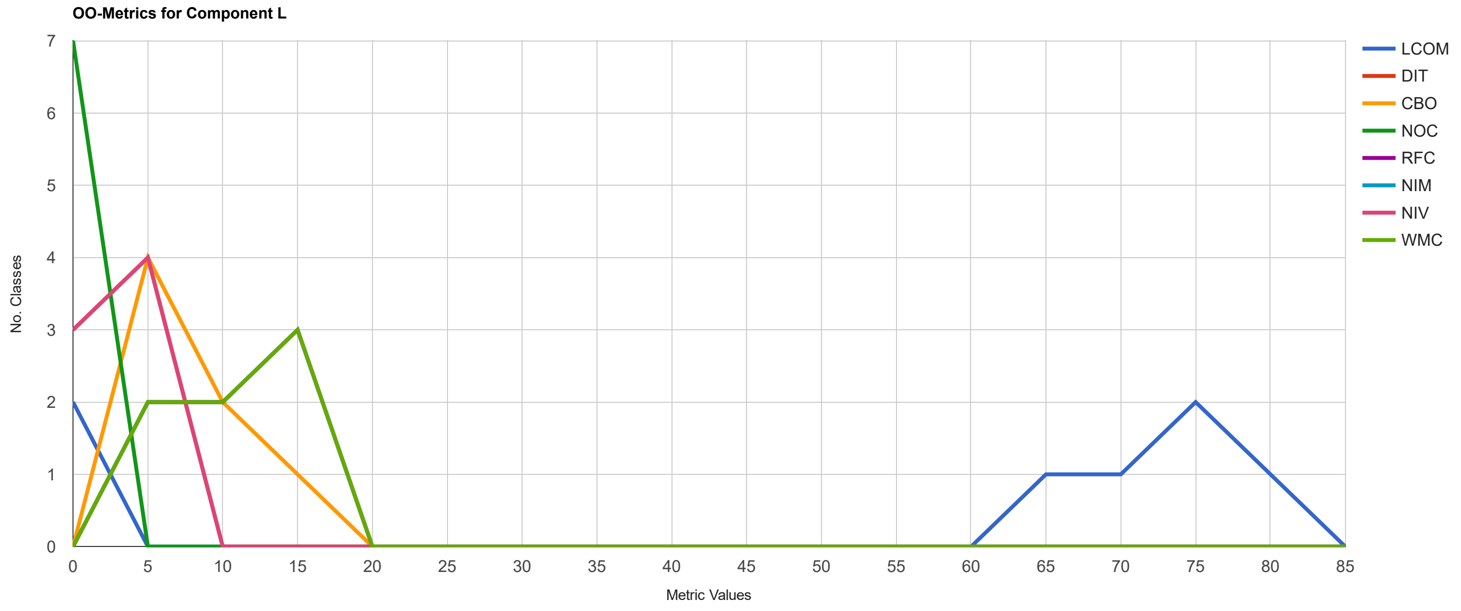
OO-Metrics for Component L
No. Classes
Metric Values
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
LCOM
DIT
CBO
NOC
RFC
NIM
NIV
WMC

Table 4.10: OO-metrics for Component L

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	80	58	50.857	35.130
DIT	0	1	1	0.571	0.534
CBO	1	13	4	5.571	4.197
NOC	0	0	0	0	0
RFC	5	12	9	8.571	2.936
WMC	5	12	9	8.571	2.936
NIM	3	12	7	8.286	3.402
NIV	0	5	1	1.571	2.070

Component N

17 files, 1839 lines of code, 8 classes.

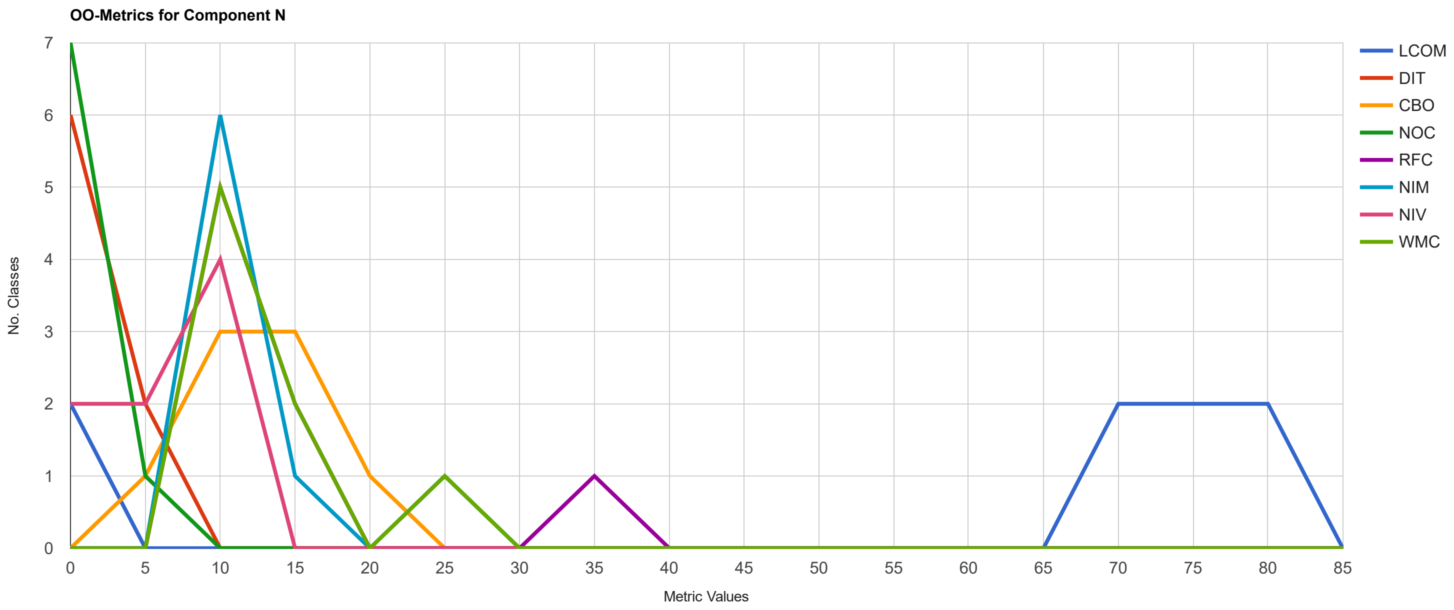
OO-Metrics for Component N
No. Classes
Metric Values
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
LCOM
DIT
CBO
NOC
RFC
NIM
NIV
WMC

Table 4.11: 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	17	10.5	10	4.140
NOC	0	1	0	0.125	0.353
RFC	6	32	9	11.625	8.568
WMC	6	23	9	10.5	5.580
NIM	6	21	8.5	9.75	5.036
NIV	0	8	5.5	4.375	3.068

Component P

12 files, 722 lines of code, 8 classes.

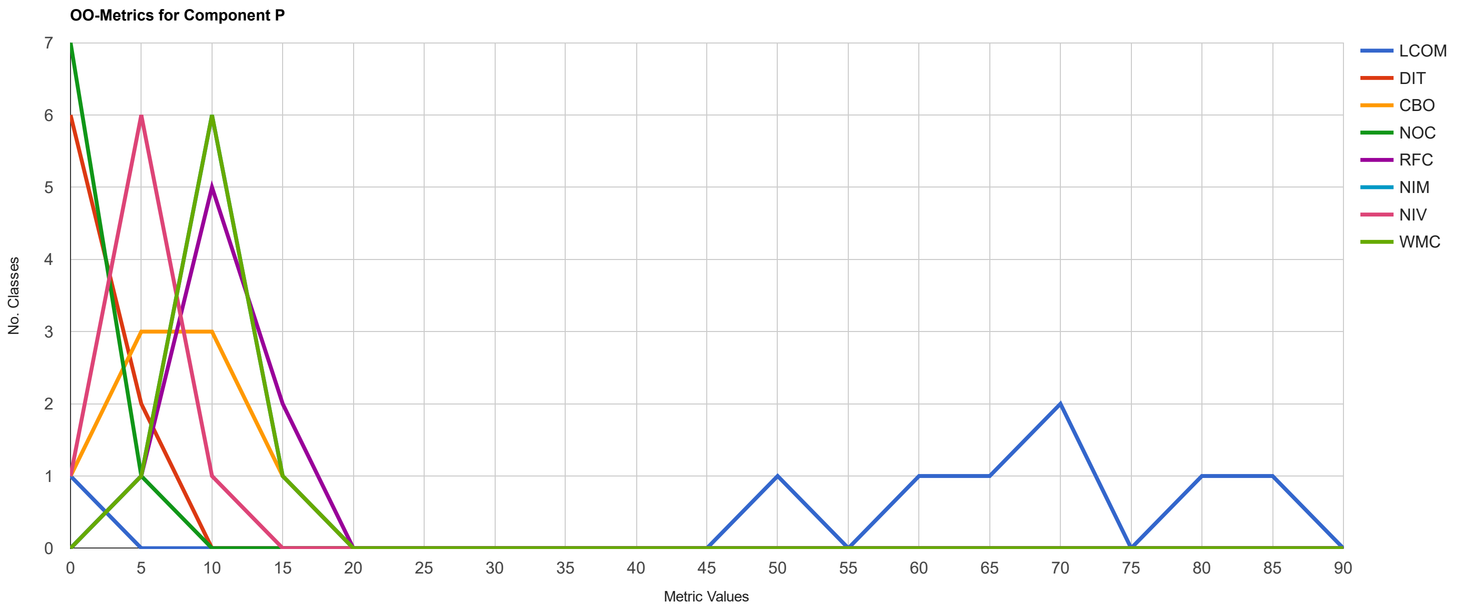
OO-Metrics for Component P
No. Classes
Metric Values
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
LCOM
DIT
CBO
NOC
RFC
NIM
NIV
WMC

Table 4.12: OO-metrics for Component P

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	0	81	65	58.25	26.611
DIT	0	1	0	0.25	0.463
CBO	0	12	5.5	5.5	3.964
NOC	0	1	0	0.125	0.353
RFC	2	14	6.5	7.5	3.625
WMC	2	14	6.5	7.25	3.412
NIM	2	13	6.5	7	3.117
NIV	0	6	3.5	3.125	2.031

Component S

4 files, 223 lines of code, 2 classes.

Component W

1 file, 1 class, 69 lines of code.

Our results show that (whats good and wrong with these metrics, what can be done).

4.2 Identifying Code Smells using Automatic Approaches

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. Table 4.15 describes the number of code smells that were identified using automatic.

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 4.15 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 affecting 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

Table 4.13: 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	6	4.5	4.5	2.121
NOC	0	0	0	0	0
RFC	6	9	7.5	7.5	2.121
WMC	6	9	7.5	7.5	2.121
NIM	6	9	7.5	7.5	2.121
NIV	1	1	1	1	0

Table 4.14: OO-metrics for Component W

Metric	Min	Max	Median	Sample Mean	Standard Deviation
LCOM	58	58	-	58	-
DIT	0	0	-	0	-
CBO	4	4	-	4	-
NOC	0	0	-	0	-
RFC	6	6	-	6	-
WMC	6	6	-	6	-
NIM	6	6	-	6	-
NIV	2	2	-	2	-

Table 4.15: Number of Code Smells detected

Code Smell	Detected
Long Method	10
Large Class	8
Long Parameter List	15
Duplicated Code	Approximately 5% of the source code. 39 files affected.
Speculative Generality	1153
Dead Code	151

Table 4.16: 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

spread across Component B, N, P, C, D, L, Ex, S, and G. Table **??**ummarizes duplication in the various components.

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 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 4.17 summarizes Speculative Generality code smell that were identified through a code analysis using Understand. The results are divided into the categories unused functions, unused local variables, and unused static globals.

Table 4.17: Speculative Generality Results

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

Table 4.18: Dead Code Results

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

Shotgun Surgery

The results shows that X classes are infected with the Shotgun Surgery code smell.

Dead Code

Fowler and Beck [57] 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 [33]. We examined three types of ”Dead Code” code smell in Project Firmus: ”Commented Out” Code, Unreachable Code, and Unnecessary Includes in Header Files. In total, we found 151 hits of ”Dead Code” code smell, which we have summarized in Table 4.18.

Large Class

We were not able to identify any large classes using automatic tool. However, by counting the number of instances, variables, and methods using object-oriented metrics, we were able to identify some large classes in the system. Table X summarizes Large Class code smells.

How did we study the different code smells, the apporach and the results. The results from Table XX

As we see, there are many code smells detected. We take a closer look at some of the classes; presented in UML diagrams here:

CHAPTER 5

DISCUSSION

5.1 Threats To Validity

5.1.1 Internal Validity

5.1.2 External Validity

5.1.3 Construct Validity

CHAPTER 6

CONCLUSION

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] 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.

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

[4] 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.

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

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

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] J. Knodel, M. Lindvall, D. Muthig, and M. Naab, “Static evaluation of software architectures,” in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pp. 10–pp, IEEE, 2006.

[43] R. Pressman, *Software Engineering: A Practitioner’s Approach.* New York, NY, USA: McGraw-Hill, Inc., 7 ed., 2010.

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

[45] D. E. Perry, “State of the art: Software architecture,” in *International Conference on Software Engineering*, vol. 19, pp. 590–591, IEEE COMPUTER SOCIETY, 1997.

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

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

[48] 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.

[49] “IEEE Standard for Software Maintenance,” *IEEE Std 1219-1998*, 1998.

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

[53] 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.

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

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

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

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

[58] 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.