



Managing Technical Debt in Embedded Systems

Shahariar Kabir Bhuiyan

Autumn 2015

Specialization project 2015

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

Supervisor 1: Carl-Fredrik Sørensen

Abstract

Context: Today, software is contributing a substantial part of new functionalities and innovations of the embedded industry. With size and complexity of the embedded software growing as time the time goes, additional challenges arises, including implicit assumptions of technical debt. Technical debt refers to situations where shortcuts are taken in technical decisions. Technical debt has been identified as one of the key reasons to software system projects failures. Accumulation of technical debt may reduce the dependability and maintainability of the embedded systems.

Objective: The goal of this study is to compare existing research on technical debt and embedded systems, with the insights and experiences of traditional software and embedded system practitioners from the industry. We will explore and understand the causes of technical debt, as well as approaches for managing technical debt in embedded systems.

Research Method: This research starts by conducting a literature review to get an overview within the field of technical debt and embedded systems. Furthermore, four people from different companies were interviewed using semi-structured interviews. Two of them works with traditional software development, while the last two works with embedded system development.

Results: The results consists of the knowledge similar to the reseach within technical debt, and embedded software quality. There are many similarities between the causes and management practices of technical debt. The results shows that technical debt is mostly formed as a result of intentional decisions to reach deadlines. However, the results revealed that the embedded system industry has more control over technical debt than traditional software industry. The results also revealed that neither of the participants had any specific management plan for reducing technical debt, but several practices were identified. The results may help companies to understand the existing practices of technical debt management, and use it to improve their own processes.

Preface

This report has been written during autumn 2015, as a part of the course TDT4501, Specialization Project for Software Engineering, at Norwegian University of Science and Technology. This study can be seen as a pre study for the upcoming Master's Thesis in the spring 2016.

This work has been supervised by Carl-Fredrik Sørensen. I would especially like to thank him for providing insightful input and valuable feedback during the period. He also helped us with providing names of potential candidates in Trondheim along with their contact details.

At last, I would like to thank all the four participants who participated in the interviews.

Trondheim, December 16, 2015

Abstract	i
Preface	iii
1 Introduction	1
1.1 Motivation and Background	1
1.2 Research Questions	2
1.3 Research Method	3
1.4 Project Structure	3
2 State-of-the-Art	5
2.1 Technical Debt	5
2.1.1 Definitions of Technical Debt	6
2.1.2 Comparison with Financial Debt	8
2.1.3 Causes and Effects of Technical Debt	8
2.1.4 Current Strategies and Practices for Managing Technical Debt	10
2.2 Software Quality	12
2.3 Software Architecture	12
2.4 Software Life-cycle	14
2.4.1 Software Development Life Cycle and Methodologies	14
2.5 Software Evolution	17
2.5.1 Evolution Processes	18
2.5.2 Software Maintenance	20
2.6 Software Reuse	21
2.7 Refactoring	22
2.8 Configuration Management	22
2.9 Embedded Systems	24
2.9.1 Embedded Software	24
3 Research Method	27

3.1	Empirical Strategies	27
3.1.1	Choice of Methods	28
3.1.2	Data collection	28
4	Results	30
4.1	Introduction of participants	30
4.2	Definition of Technical Debt	31
4.3	Causes of Technical Debt	32
4.3.1	Time Pressure	32
4.4	Incurring Technical Debt	34
4.5	Management of Technical Debt	36
5	Discussion	38
5.1	Definitions of Technical Debt	38
5.2	Causes of Technical Debt	39
5.3	Incurring Technical Debt	40
5.4	Management of Technical Debt	40
5.5	Research Questions	42
5.6	Threats to Validity	42
5.6.1	Internal validity	43
5.6.2	External Validity	43
5.6.3	Construct Validity	44
5.6.4	Conclusion Validity	44
6	Conclusion	45
6.1	Future Work	46

LIST OF TABLES

1.1	Internet of Things Units Installed Base by Category [1]	2
2.1	Types of Technical Debt	10
2.2	The Sub-Characteristics Adopted by ISO/IEC 9126-2001	13
2.3	Reusable assets in software projects	21
2.4	Types of Software Artifacts that can be refactored.	22
4.1	The participants and their background	30
4.2	The participants and their project	31
5.1	Research Questions and Their Findings	43

LIST OF FIGURES

1.1	Model of research process [2]	3
2.1	The Technical Debt Curve [3]	6
2.2	Fowler's Technical Debt Quadrant	7
2.3	Technical Debt Landscape	7
2.4	The Colors Reconcile Four Types of Possible Improvements.	11
2.5	Agile implementation Success Rate by The Standish Group	17
2.6	Software evolution process	18
2.7	Software lifespan	19
2.8	Software evolution process	20
2.9	Distribution of maintenance activities [4]	21

1.1 Motivation and Background

The field of embedded systems is growing rapidly based on the evolution in electronics and widespread use of sensors and actuators. From consumer electronics, automobiles, to satellites, embedded systems represent one of the largest segments of the software industry. The automotive domain is going through a major transition where all car manufacturers are working towards the development of self-driving cars. Internet of Things describes the concept of interconnecting the virtual world of computers with the real world of physical artifacts [5]. This leads to a distributed network of devices communicating with other devices as well as humans. Gartner [1] have estimated that in 2020, 25 billion connected "things" will be in use. Our society has come to depend on such systems for its day-to-day operation. Embedded systems is forecasted to grow exponentially in the next 10 years [6].

Software plays an important role in the development of embedded systems [7]. Embedded software is the primary driving force for implementing different functionalities of today's embedded systems. The software is specialized for one particular type of hardware, and may therefore have hardware specific run-time constraints. As the complexity of embedded system grows, the ability to maintain the required quality of such systems becomes more difficult [7]. Traditional software development technologies do not take into account the specific needs of embedded systems development [6].

Moreover, many companies are forced to think about their time-to-market strategy to keep up with the increased competition. Companies need to decide what kind of shortcuts they will have to take in the development process. Such compromises causes the creation of a financial overhead in the future maintenance activities, usually termed as Technical Debt [8]. As the amount of technical debt accumulated in the developed embedded systems grows continuously, companies need to manage the overall debt while keeping the system flexible and extensible. Defects can cause life-threatening situations, delays can create hugs

costs, and insufficient productivity can impact an entire company [7]. Pacemakers are a good example of how embedded software helps millions of person live a better live. Yet between 1990 and 2000, software errors accounted for about 40% of half million devices recalled [7]. If the companies could catch software defects earlier in the system design process, their income would be saved. It is necessary for companies to find out how to make decisions so future maintenance and evolution has as low cost as possible.

According to Gartner [9], 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. IT management teams must measure the level of technical debt in their products, and develop a strategy to deal with technical debt.

Table 1.1: Internet of Things Units Installed Base by Category [1]

Category	2013	2014	2015	2020
Automotive	96.0	189.6	372.3	3,511.1
Consumer	1,842.1	2,244.5	2,874.9	13,172.5
Generic Business	395.2	479.4	623.9	5,158.6
Vertical Business	698.7	836.5	1,009.4	3,164.4
Grand Total	3,032.0	3,750.0	4,880.6	25,006.6

1.2 Research Questions

The main objective of this research is to increase the knowledge of technical debt in embedded systems by looking at significant sources for technical debt, and practices for managing the debt. The reason for this is that embedded systems usually has long lifetime, and it is important to find out how such systems are managed because the architecture and design decisions are usually made long time ago and the decision makers might not be available anymore.

The research questions in this research will be:

- **RQ1:** What practices and tools exists for managing technical debt? How are they used?
- **RQ2:** What are the most significant sources of technical debt?
- **RQ3:** When should a technical debt be paid?
- **RQ4:** Who is responsible for deciding whether to incur, or pay off technical debt?

1.3 Research Method

The most relevant research methodologies in software engineering are illustrated in Figure 1.1. Throughout this research, the research process illustrated in the model will be used as a basis for the elaboration of how to conduct the research in the upcoming master thesis.

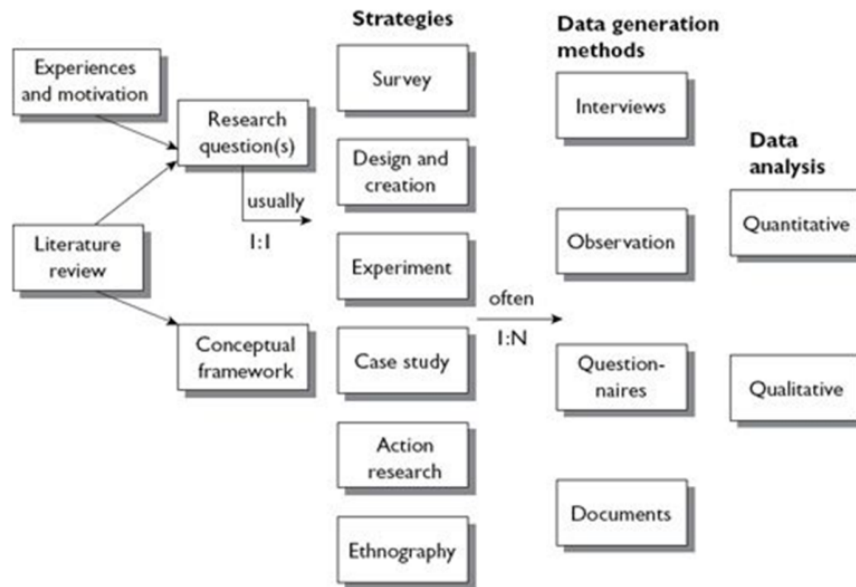


Figure 1.1: Model of research process [2]

To define the research questions, it is necessary to get an overview of the research field by conducting a review of published research within the selected area of study, providing a conceptual framework. Moreover, researchers can use their experiences and motivations. A research strategy is needed to answer the research questions. Oates [2] presents six different research strategies: survey, design and creation, experiment, case study, action research, and ethnography. A data collection method is needed to produce empirical data or evidence. Furthermore, Oates [2] presents four different data collection methods: interviews, observations, questionnaire, and documents. Research data can either be quantitative or qualitative.

1.4 Project Structure

The report is structured into several chapters:

- **Chapter 1** introduces the problem and motivation behind this project along with the search questions.
- **Chapter 2** provides a state-of-art within the field of technical debt, embedded systems, and software engineering.

-
- **Chapter 3** presents the research method, and the procedures that was used behind the method.
 - **Chapter 4** provides an overview of the results acquired from the research method.
 - **Chapter 5** presents a discussion of the relevant topics related to the research questions.
 - **Chapter 6** concludes the report and provides some points to future work.

This chapter presents state-of-the-art topics which are relevant to this project. Section 2.1 presents the technical debt with definitions, causes, and management tools. Section 2.2 looks into the topic of software quality and the relevant attributes. Section 2.4 presents the software life cycle along with the different development methodologies. Section 2.3 presents software architecture. Section 2.5 presents software evolution and maintenance. Section 2.6 takes a look into software reuse. Section 2.7 presents refactoring. Section 2.8 presents configuration management. Lastly, Section 2.9 will take a closer look at embedded systems and software.

2.1 Technical Debt

The concept of technical debt was first introduced by Ward Cunningham in 1992 to communicate technical problems with non-technical stakeholders [8]. The concept was used to describe the system design trade-offs that are made everyday. To deliver business functionality as quickly as possible, '*quick and dirty*' decisions had to be made, which affected the future development activities. Furthermore, Cunningham describes technical debt as "*shipping first time code is like going into debt. A little debt speeds development as long as it is paid back promptly with a rewrite*". As the time goes, technical debt accumulates *interest*, leading to increased costs of a software system [10, 11]. Li et al. [12] defines interest as *the extra effort needed to modify the part of the software system that contains technical debt*. However, not all debts are necessarily bad. A small portion of debt may help developers speed up the development process, resulting in short-term benefits [10].

Figure 2.1 illustrates the effects of technical debt growth in a system.

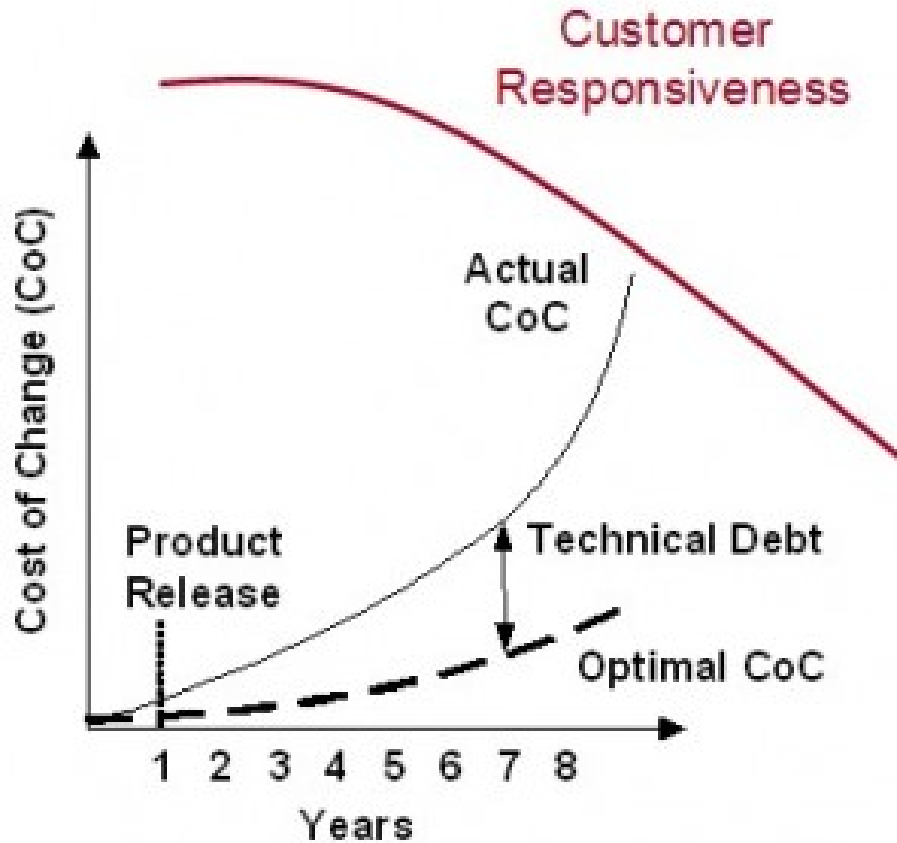


Figure 2.1: The Technical Debt Curve [3]

2.1.1 Definitions of Technical Debt

McConnell [13] describes technical debt as *a design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including the increased cost over time)*. He splits the term into two categories based on how they are incurred; technical debt that is incurred *intentionally*, and technical debt that is incurred *unintentionally*. For instance, unintentional debt accumulates when a junior software developer writes low quality code due to lack of knowledge and experience. Intentional debt commonly occurs when an organization makes a decision to optimize for the present rather than the future. These type of decisions results in shortcuts being taken to solve a problem.

Fowler [14] presents a more formal explanation of how technical debt can occur. He categories technical debt into a quadrant with two dimensions, which he calls the *"Technical Debt Quadrant"*. As seen in the Figure 2.2, the debt is grouped into four categories:

- **Reckless/Deliberate debt:** The team feels time pressure, and takes shortcuts intentionally without any thoughts on how to address the consequences in the future.
- **Reckless/Inadvertent debt:** Best practices for code and design is ignored, and a big mess in the code base is made.

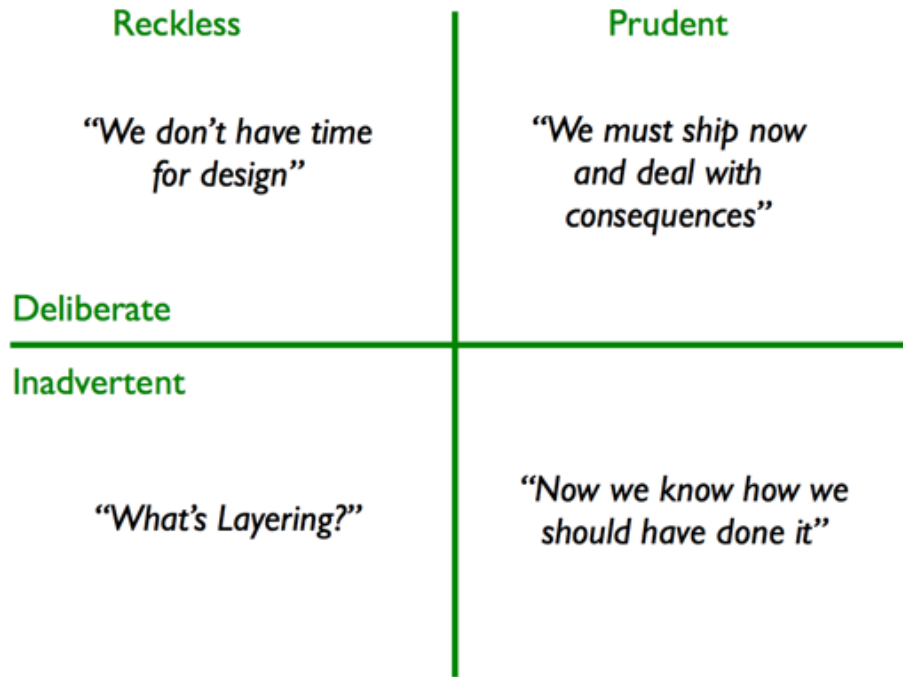


Figure 2.2: Fowler's Technical Debt Quadrant

- **Prudent/Deliberate debt:** : The value of taking shortcuts is worth the cost of incurring debt in order to meet a deadline. The team is aware of the consequences, and has a plan in place to address them in the future.
- **Prudent/Inadvertent debt:** Software development process is as much learning as it is coding. The team can deliver a valuable software with clean code, but in the end they may realize that the design could have been better.

Krutchén et al. [15] divides technical debt into two categories; *Visible debt*, that is visible for everyone. It contains elements such as new functionality to add, and defects to fix. *Invisible debt* is the other category, debt that is only visible to software developers. Figure 2.3 illustrates a map of the "Technical Debt Landscape", in which distinguish visible and invisible elements. On the left side of Figure 2.3, technical debt mostly affects evolvability of the software system, while on the right side, technical debt mainly affects maintainability.

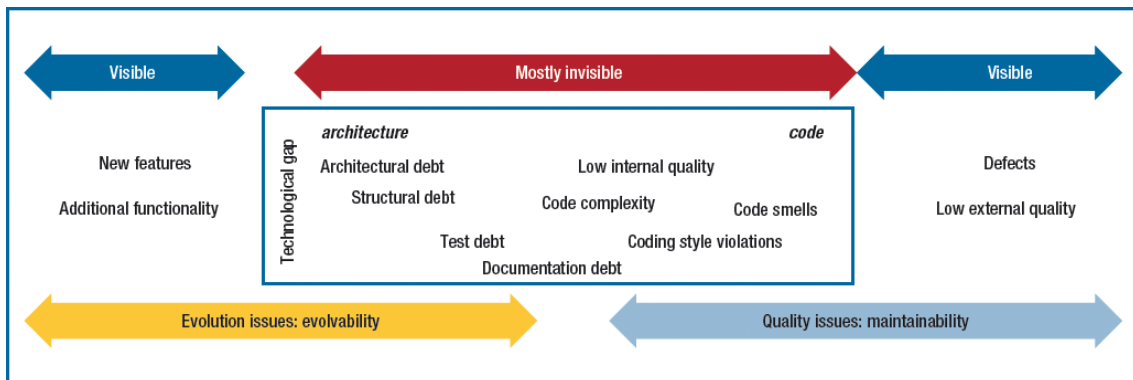


Figure 2.3: Technical Debt Landscape

2.1.2 Comparison with Financial Debt

Technical debt has many similar characteristics to financial debt [16, 17]:

- You take a loan that has to be repaid later
- You usually repay the loan with interest
- If you can not pay back, a very high cost will follow. For example, you can lose your house or car.

Like financial debt, technical debt accrues interest over time which comes in the form of extra effort that has to be dedicated in future development [10, 11]. Stakeholders can choose to continue paying interest, or to reduce future interest payments by refactoring the problem [14].

However, if the debt is not repaid, the development may slow down, resulting in software project failure or bankruptcy [16]. Moreover, there are some differences between financial debt and technical debt. The debt has to be repaid eventually, but not on any fixed schedule [16]. This means that some debts may never have to be paid back, depending on the interest and the cost of paying back the debt [18].

2.1.3 Causes and Effects of Technical Debt

Multiple studies have tried to analyze the reason for companies to incur technical debt.

Klinger et al. [11] conducted an industrial case study at IBM where four technical architects with different backgrounds were interviewed. The goal was to examine how decisions to incur debt were taken, and the extent to which the debt provided leverage [11]. 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. [19] 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. [20] 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. [21] 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. [22] 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 [17]. 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 [23] 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. [24] 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.

The 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 [20].

Table 2.1: Types of Technical Debt

Subcategory	Definition
Architectural debt [12, 18, 24]	Architectural decisions that make compromises in some of the quality attributes, such as modifiability.
Code debt [12, 18, 25]	Poorly written code that violates best coding practices and guidelines, such as code duplication.
Defect debt [12, 25]	Defect, failures, or bugs in the software.
Design debt [12, 17, 18]	Technical shortcuts that are taken in design.
Documentation debt [12, 18, 26]	Refers to insufficient, incomplete, or outdated documentation in any aspect of software development.
Infrastructure debt [12, 24, 25]	Refers to sub-optimal configuration of development-related processes, technologies, and supporting tools. An example is lack of continuous integration.
Requirements debt [12, 26]	Refers to the requirements that are not fully implemented, or the distance between actual requirements and implemented requirements.
Test debt [12, 18, 26]	Refers to shortcuts taken in testing. An example is lack of unit tests, and integration tests.

Furthermore, the studies points out that several types of technical debt are related to software life-cycle phases. The effects of taking shortcuts can happen in several stages of software life-cycle. Table 2.1 lists the types of technical debt that has been identified in the literature.

2.1.4 Current Strategies and Practices for Managing Technical Debt

Managing technical debt compromises the actions of identifying the debt and making decisions about which debt should be repaid [13, 15, 18]. This section examines some methods that has been proposed by several authors.

Brown et al. [18] proposed open research questions to understand the need to manage technical debt. The questions includes refactoring opportunities, architectural issues, identifying dominant sources of technical debt, and identifying issues that arise when measuring technical debt.

Lim et al. [19] suggested four strategies for managing technical debt. The first strategy is to do nothing because the technical debt may never be visible to the customer. The second strategy is to use a risk management approach to evaluate and prioritize technical debts cost and value by allocating 5-10% of each release cycle to address technical debt. The third strategy is to include the customers and non-technical stakeholders to technical debt decisions. Finally, the last strategy suggests to track technical debt items using tools like a Wiki, or a backlog.

Codabux et al. [24] suggested best practices such as refactoring, repackaging, re-engineering, and developing unit tests to manage technical debt. Moreover, they also propose that having dedicated teams with the purpose of reducing technical debt, while the product development team devote 20% of their effort toward technical debt reduction.

Guo et al. [10] suggested using of portfolio management for technical debt management. This approach collects technical debt items to a *Technical Debt List* (Technical DebtL). TDL is used to pay technical debt back based on its cost and value. Three activities support the TDL. The first activity is Technical Debt Identification. Technical Debt Identification uses several tools to identify technical debt items, in which are automatically placed in the TDL. The second activity is Technical Debt Estimation. Each item in the list is assigned the estimates for the debt principal, and the interest. The third activity, Decision Making, is used to determine which debts should be addressed first, and when they should be addressed.

Nugraho et al. [27] proposed an approach to quantify technical debt and its interest by using a software quality assessment method. This method rates the technical quality of a system in terms of the quality characteristics of ISO/IEC 9126.

Krutchén et al. [15] suggested listing debt-related tasks in a common backlog during release and iteration planning. Figure 2.4 illustrates how these elements can be organized in a backlog. Moreover, Krutchén mentioned that project backlogs often contain the green elements. The rest are seen rarely, especially the black elements, they are nowhere to be found.

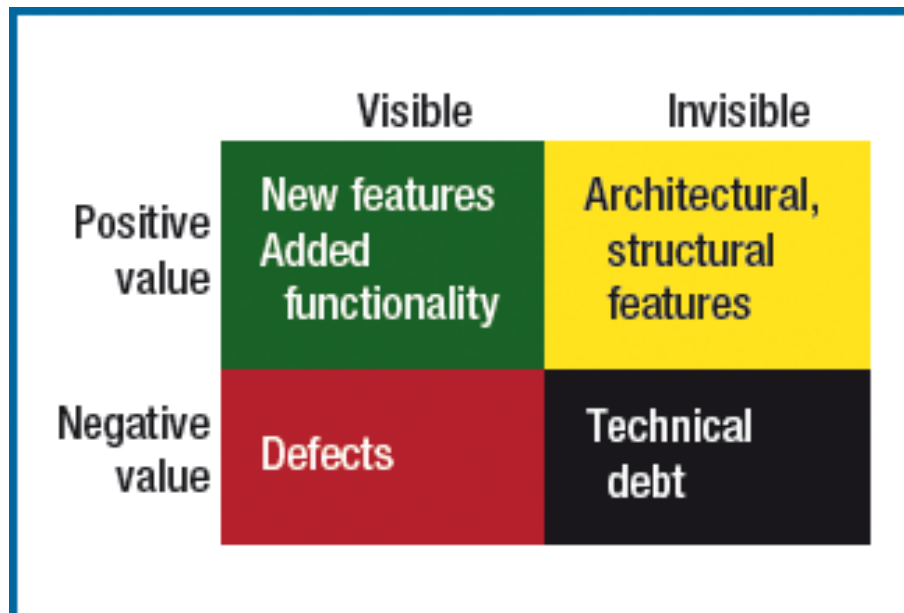


Figure 2.4: The Colors Reconcile Four Types of Possible Improvements.

SonarQube is an open source application for quality management [28]. It manages the results of various code analysis tools, and is used to analyze and measure a projects technical quality. Technical debt is computed based on the SQALE (Software Quality

Assessment based on Life-cycle Expectations) methodology [29]. SQA is a method for assessing technical debt in a project. It is based on tools that analyze the source code of the project, looking at different types of errors such as mismatched indentation, and different naming conventions. Each error is assigned a score based on how much work it would take to fix that error. The analysis gives a total sum of technical debt for the entire project.

2.2 Software Quality

Software Quality (SQ) is defined as *an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it* [30]. ISO/IEC 9126-2001 is an international standard for evaluating software [31]. It is currently one of the most widespread quality standards [32]. The standard defines quality as *the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs*. Additionally, ISO/IEC 9126-2001 offers a valuable conceptual framework for SQ, where it makes a distinction between *"quality in use"*, *"external quality"*, and *"internal quality"* [31, 32]. *Quality in use* is the users view of the quality of a system. *External quality* reflects the dynamic aspect of a software application, and is subdivided into six quality characteristics. *Internal quality* is reflected by a subdivision of the six external quality characteristics into internal quality attributes.

Table 2.2 describes the quality attributes, and their sub-characteristics (criteria).

2.3 Software Architecture

Bass et al. [33] 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 [33, 34]. 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 [30], activities that consumes a big part of the systems lifespan [4]. The issues of software architecture has long been a concern for those building and evolving large software systems [35].

Software architecture can be seen from two standpoints [36]; *prescriptive and descriptive architecture*. The *prescriptive architecture* of a system captures the design decisions

Table 2.2: The Sub-Characteristics Adopted by ISO/IEC 9126-2001

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

made prior to the construction. This is normally called as-conceived software architecture. *Descriptive architecture* describes how the system has actually been build, called for as-implemented software architecture.

As the system evolves, it is ideal that the prescriptive architecture is modified first. In practice, the system - the descriptive architecture - is often directly modified [33]. This may be due to developers sloppiness, short deadlines, or lack of documented prescriptive architecture. These principles introduces two new concepts; *architectural drift* and *architectural erosion* [33]. Architectural drift occurs when the documents are updated according to the implementation. The software architecture ends up as an architecture without vision and direction. Architectural erosion occurs when the implementation drifts away from the planned architecture.

System requirements can be categorized as *functional requirements*, *quality attribute requirements* (QA requirements), and *constraints* [33]. *Functional requirements* states what a system must do. *QAs* are the non-functional requirements of a system. *Constraints* is a design decision with zero degrees of freedom. Moreover, long-term responsiveness of a system can be achieved by providing a solution of a system that would enable and achieve systems driving QA.

As a final step in the architecture design phase, more and more organizations are evaluating their design decisions using Architecture Trade-off Analysis Model [33, 37]. The goal of the ATAM is to understand the consequences of architectural decisions with respect to the quality attribute requirements of the system.

2.4 Software Life-cycle

A Software Life-Cycle (SLC) is the phases a software product goes through between its convenient, to when its no longer available for use [38]. According to *IEEE Standard for Developing SLC Processes* [38], there are five general groups of related activities in the SLC:

1. The first group is project management. Every software life-cycle starts with the project initiation. Project planning, and project monitoring and control are two other, necessary activities withing this group for each project iteration.
2. The second group is pre-development. This group consists of activities that needs to be performed before the software development phase. Concept exploration is a good example of such activity.
3. The third group is the development itself. It includes the activities that must be performed during the development.
4. The fourth group is post-development. It includes activities to be performed after development to enhance the software project. The retirement activity involves removal of the existing system from its active support by ceasing its operation or support, or replacing it with a new system or an upgraded version of the existing system.
5. The final group is called integral. This group consists of activities that are necessary to ensure successful completion of a project. These activities are seen as support activities rather than activities that are directly oriented to the development effort.

2.4.1 Software Development Life Cycle and Methodologies

A software development process or a software development life-cycle (SDLC) is defined as the process by which user needs are translated into a software product [39]. The process involves translating user needs into requirements, transforming requirements into design, implementing design into code, testing the code, and sometimes, installing and checking out the software for operational use.

A software development methodology is defined as a framework to structure, plan, and control the software development process. Many software development methodologies exists, and the basic life-cycle activities are included in all life-cycle models, often in

different orders. The difference is in terms of time to release, risk management, and quality. The models can be of different types, but they are usually defined as traditional and agile software development methodologies.

Traditional Software Development

Traditional software development methodologies are based on a sequential series of steps. They usually starts with elicitation and documentation of a complete set of requirements, followed by architecture and high level design, development, testing, and deployment. The most well-known of these traditional software development methodologies is the Waterfall method, the oldest software development process model [40]. The Waterfall Model divides the software development life-cycle into five distinct and linear stages [4]; requirements engineering, design, implementation, testing, and maintenance. There are many risks associated with the use of Waterfall model [41]:

- **Continuous requirements change:** Requirements are specified at the beginning of the software development process, and the remaining software development activities have to follow the initial requirements. This kind of model is not appropriate to use for software where technology and business requirements always change.
- **No overlapping between stages:** Each stage in the Waterfall model needs to be completed entirely before proceeding into the next phase.
- **Poor quality assurance:** Lack of quality assurance during the different phases is another source of risk. Testing the system is the last stage in the development process. Thus, all problems, bugs, and risk are discovered too late.
- **Relatively long stages:** Long stages in the development process makes it difficult to estimate time and cost. Additionally, there is no working product until late in the development process.

Using sequential design processes such as Waterfall model in software development processes to build complex, intensive systems is often a failure [16]. According to the Standish Group, CHAOS Report of 2015, 29% of all projects using Waterfall method failed in 2015, with no useful software deployed¹.

Agile Methods

To address the challenges posed by traditional methods, agile methods were developed as a set of lightweight methods [41]. Agile methods try to deal with collaboration in a way that promotes adaptive planning, early delivery, and continuous improvement, making

¹Standish Group 2015 Chaos Report - Q&A with Jennifer Lynch, <http://www.infoq.com/articles/standish-chaos-2015>

the development phase faster and more flexible regarding changes [42]. Agile Manifesto describes four values that defines agile software development [43]:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Scrum is one of the most popular agile software development methodologies [4, 30, 44]. It is an iterative and incremental software development model. An advantage with iterative procedures is that parts of a system is developed early on, and can be tested before implementation of other parts. This reduces the risk of having long stages [45]. The idea with Scrum is to divide the development into short periods, called sprints. Unlike the approach in the Waterfall model, the team can estimate how long it will take to implement tasks which can be accomplished during each sprint. To implement the requirements step by step, a product backlog is kept containing the features that have yet to be implemented. The product backlog is not static as it changes to the needs of the project, adding new features, and removing obsolete items. Sprint backlog contains items from the backlog that a team works on during a sprint.

Lean Development adapts the concepts and principles of Toyota Product Development System, to the practice of developing software [46]. It is seen as a key component in building a change tolerant business [47]. The seven lean principles that is applied to software development are summarized as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole* [30, 46].

Despite the fact that agile methods addressed the challenges posed by traditional methods, there are many risks associated with agile methodologies [41]:

- **Very large software system:** Developing large, complex software systems results in large increments. This increase the time span between increments, and thus require a higher cost to deal with changes and bugs if discovered.
- **Large development team:** Having large teams results in difficulties in managing communication between team members.
- **High reliance on human factor:** Agile methodologies relies on the development team, and their abilities to communicate with the customers.
- **Inappropriate customer representative:** This factor can influence the development process and the team members.
- **Distributed development environment:** Agile methodologies requires close interaction between the development team. Having a distributed development environment might challenge the communication between team members due to different

CHAOS RESOLUTION BY AGILE VERSUS WATERFALL				
SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	56%	38%	4%
	Waterfall	44%	45%	11%

The resolution of all software projects from FY2011–2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000.

Figure 2.5: Agile implementation Success Rate by The Standish Group

time zones.

- **Scope creep:** With minimal planning conducted, developers can be easily distracted from the project main objectives.

2.5 Software Evolution

Increasingly, more and more software developers are employed to maintain and evolve existing systems instead of developing new systems from scratch [44]. Software evolution is a process that usually takes place when the initial development of a software 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 [44]. Software evolution is also important because technology tend to change rapidly.

Rajlich et al. [48] proposed a view of the software lifespan, as shown in Figure 2.6. This view divides the software lifespan into five stages with initial development as the first stage. The key contribution is to separate the maintenance phase into an evolution stage, followed by a service stage, and at last the phase-out stage.

Initial development produces the first version of the software from scratch.

Evolution is the phase where significant changes to the software may be made. This could be addition of new features, correct previous mistakes, or adjust the software to new business requirements or technologies. Each change introduces a new feature or some other new property into the software.

Servicing is the stage where relatively small, essential changes are allowed. The company considers how the software can be replaced. Legacy software is a term to describe software in this stage.

Phase-out is the phase where software may still be used, but no further changes are being implemented. Users must work around any problems that they discover, or replace the software with something else.

Close-down is when the managers or customers completely withdraw the system from production.

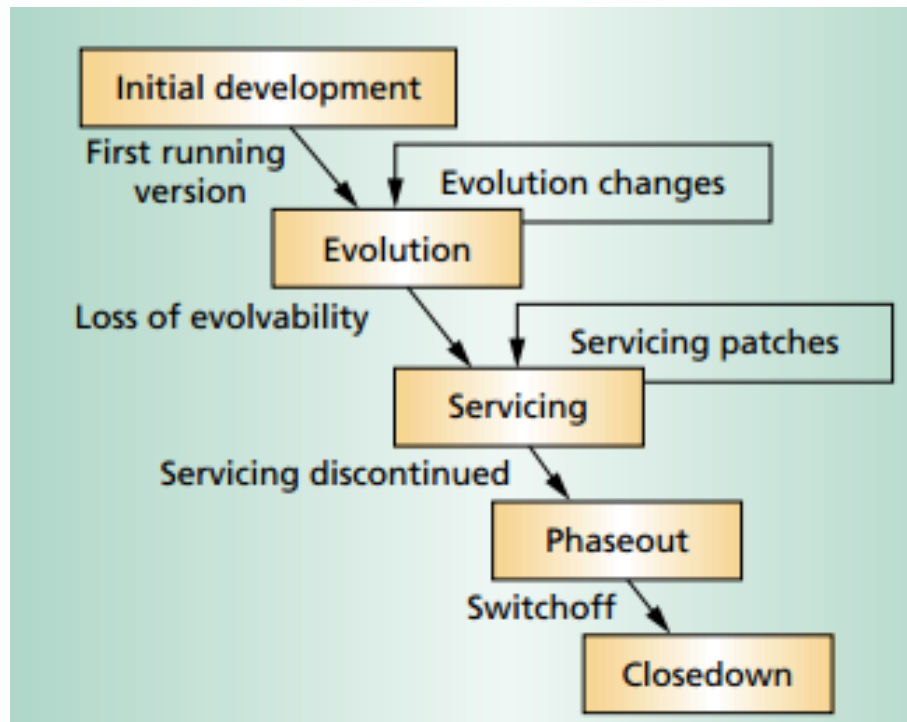


Figure 2.6: Software evolution process

A variation of this process is the versioned stage model, as shown in Figure 2.7. When a software version is completed and released to the customer, the evolution continues with the company eventually releasing another version and only servicing the previous version.

2.5.1 Evolution Processes

Software evolution usually starts with change proposals, which may be new requirements, existing requirements that have not been implemented, or bug reports from stakeholders. The process of implementing a change goes through these stages [44] as shown in Figure 2.8.

The process starts with a set of proposed change requests. The cost and impact of the change is analyzed to decide whether to accept or deny the proposed changes. If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes such as fault repair, adaptation, and new functionality,

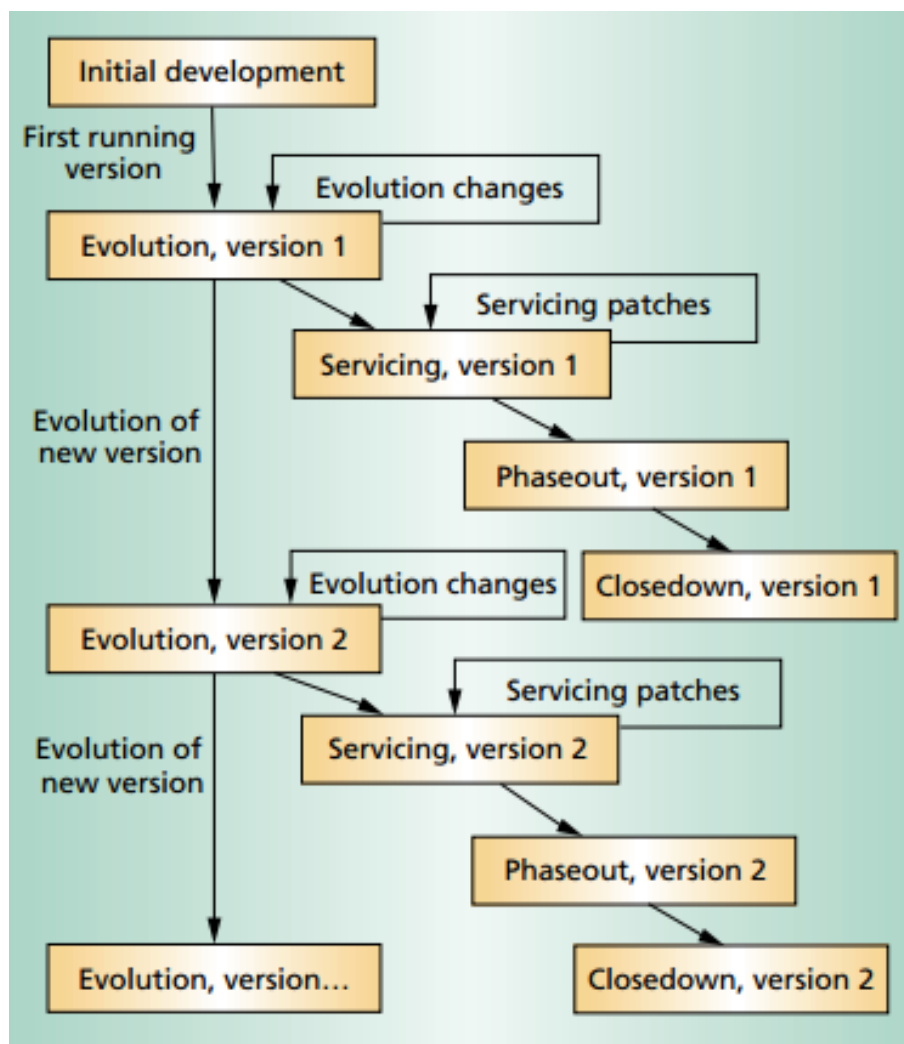


Figure 2.7: Software lifespan

are considered, to decide which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process ends with a new iteration with a set of proposed change requests for the next release.

Sometimes, the need of urgent changes may appear, such as a serious system fault that must be repaired to allow normal operation. In these cases, the usual process will not be beneficial as it takes time. An emergency fix is usually made to solve the problem. A developer choose a quick and workable solution rather than the best solution. The trade-off is that the the requirements, the software design, and the code become inconsistent. As a system changes over time, it will have impact on the systems internal structure and complexity. Software evolution might cause poor SQ and erosion of software architecture over time [33].

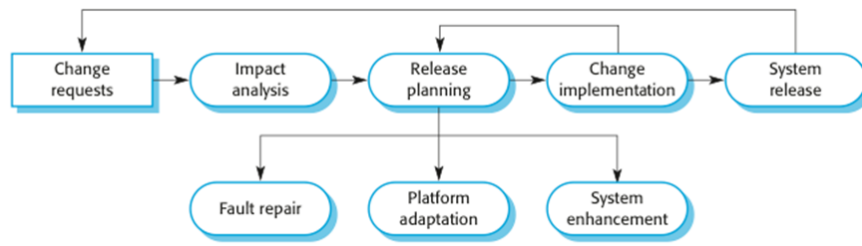


Figure 2.8: Software evolution process

2.5.2 Software Maintenance

IEEE 1219 defines software maintenance as follows [49]:

Modification of a software after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

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.

According to van Vliet, the real maintenance activity is corrective maintenance [4]. 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 [4]. This has not changed since the 1980s when Lientz and Swanson conducted a study on software maintenance [50]. Their study found out that most severe maintenance problems were caused by poor documentation, demand from users for changes, poor meeting scheduled, and problems training new hires. Some other problem areas were lack of user understanding, user training, and that customers did not understand how the system worked.

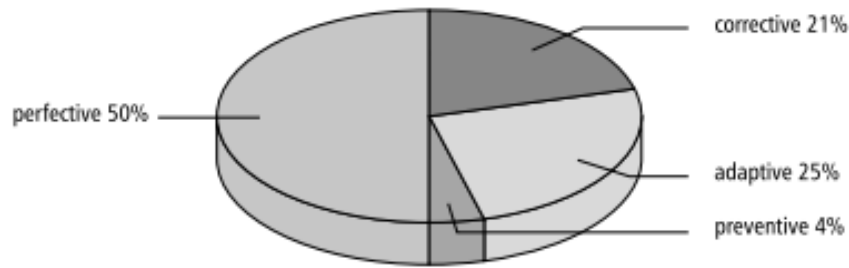


Figure 2.9: Distribution of maintenance activities [4]

2.6 Software Reuse

Software reuse is the process of using existing software artifacts, or knowledge, to create new software, rather than building it from scratch. Software reuse is a key method for improving software quality [51]. Software reuse can be specified in two directions, development *for* reuse, and development *with* reuse [52]. Development *for* reuse is related to components for reuse or system generalization. Development *with* reuse is related to how existing components can be reused in new system.

Table 2.3 lists several assets from a software project that can be reused [51].

Table 2.3: Reusable assets in software projects

1. architectures	6. estimates
2. source code	7. human interfaces
3. data	8. plans
4. designs	9. requirements
5. documentation	10. test cases

Slyngstad et al. [52] conducted an empirical study in Statoil ASA where they investigated developers views on software reuse. The results indicated that reuse include lower costs, shorter development time, higher quality of the reusable components, and a standardized architecture. These findings are very similar to the key benefits of reuse that Lim has described [53]. The quality of software artifacts increases every time the item is reused, because errors are discovered more frequently, making it easier to keep the artifact more stable [54].

Additionally, there can be problems associated with software reuse. A case study on a selected feature from self-driving miniature car development revealed that reuse of legacy, third party, or open source code, was one of the root causes for the accumulation of technical debt [55]. Morisio et al. [56] identified three main causes of software reuse failure; not introducing reuse-specific processes, not modifying non-reuse processes, and not considering human factors, combined with lack of commitment by top management.

2.7 Refactoring

technical debt accumulates as developers write code [17], and can be reduced by refactoring. Fowler defines refactoring as means of adjusting the design and architecture towards new requirements without changing the external behavior of a program in order to improve the quality of the system [57]. It is an act of improving the design and quality of an existing system [4]. Most of the time spent on reducing technical debt is on refactoring activities itself. These activities includes planning the design and architecture, rewriting the code, and adjusting documentation [30]. It is believed that refactoring improves software quality and developer productivity, by making it easier to understand and maintain software codes [58], thus a way to manage the technical debt of a system.

Table 2.4 lists the software artifacts that refactoring applies to [59].

Table 2.4: Types of Software Artifacts that can be refactored.

Programs
Refactoring at the source code or program level. For example, extracting methods, and encapsulating fields.
Designs
Refactoring at design level, for example in the form of UML models. Design patterns, software architecture, and database schemas, are some examples on artifacts that can be refactored at this level.
Software Requirements
Refactoring at the level of requirements specification. For example, decomposing requirements into a structure of viewpoints.

2.8 Configuration Management

Systems always change to cope with bugs and introduce new features. A new version of a system is created when changes are made [60]. Dart [61] defines configuration management (CM) as a discipline for controlling the evolution of software systems. CM identifies every component in a project and has an overview of every suggestions and changes from day one to the end of the product. CM involves four related activities [44]:

Change management is intended to ensure that the evolution of a system is a managed process, and to prioritize changes. Costs and benefits has to be analyzed to approve changes and track what components have been changed. The process starts with an actor submitting a change request. The request is checked for validity. If it is valid, the costs to this change are analyzed. The change request is passed to the change control board if it is not minor. The impact of the change from a strategic

and organizational standpoint is considered, and if it is accepted, it is passed on. There are some important factors in the decision making process [44]:

a) The consequences of not making the change *b)* The benefits of the change *c)* The number of users affected by the change *d)* The costs of making the change *e)* The product release cycle

Version management is the process of keeping track of different and multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other. This is often done with version management tools, which provide features like

a) version and release identification; *b)* storage management; *c)* change history recording; *d)* independent development; or *e)* project support

System building creates an executable system by compiling and linking the program components, data, and libraries. The build process involves checking out component versions from the repository managed by the version management system, so it is necessary for system build tools and version management tools to communicate. There are many system build tools available, which provides features like *a)* build script generation; *b)* version management system integration; *c)* executable system creation; *d)* test automation; or *e)* document generation.

Release management prepares the software for external release and keeps track of the system versions that have been released for customer use. Managing releases is a complex process as a release needs documentation such as configuration files, data files, and an installation program. Some factors that influences release planning are *a)* the technical quality of the system; *b)* platform changes; *c)* Lehman's fifth law; *d)* competitions; *e)* market requirements; or *f)* customer change proposals.

There are some challenges related to development of embedded systems [62]:

- **Complex file sets:** Embedded systems consists of multiple diverse components, both hardware and software. This makes the system complex. Embedded system may also have different adjustable components for a specific platform, making it easier to sell a product by tweaking some parameters. Dealing with these variates is a major challenge. Another challenge is that a product requires correct version of a component. Ensuring the consistency between components and their dependents files is a challenge as well.
- **Distributed teams:** Components may be developed in different places in our world. Two team might for example work on the same components, especially when development are being outsourced. Such collaboration needs every developer to access each others work. The challenge is keep the team synchronized.
- **Management and versioning of intellectual property:** Embedded systems, or software generally might use third-party technologies. It is important that those

technologies are up-to-date, and maintained. These updates needs to be traceable such that each components has the right, compatible and stable version of its software. If something is not outsourced, it might be a challenge for developers to contribute and trace their changes.

Software Configuration Management (SCM) is the task of tracking and controlling changes in the software through reliable version selection and version control. SCM is a part of CM. Some examples on SCM that is widely used is Git, SVN, and Adele. Choosing a robust SCM system makes it possible to deal with big and complex files. It also supports distributed development. The right combination of SCM system and best practices makes it possible for embedded development projects to progress fast and efficiently.

2.9 Embedded Systems

Embedded systems are special-purpose computer systems designed to perform certain dedicated functions under certain constraints. For instance, an embedded system in an automobile provides specific functions as a subsystem for the car itself [63]. According to Crnkovic [64], the various types of embedded systems share common requirements such as: *real-time requirements, resource consumption, dependability, and life-cycle properties*. Moreover, embedded systems are known as safety-critical and real-time systems due to their operational environment characteristics and common requirements [63, 65]. Properties such as response time and worst case execution time, are important design concerns [66]: *"When the break pedal is pressed, the computer should initiate the breaking action within one millisecond"*. Embedded systems are expected to be failure-free [67], but these complex requirements may hinder embedded systems to deliver reliable service given a disturbance to its services [68].

2.9.1 Embedded Software

In modern embedded systems, there is always a software element. The *IEEE Standard Glossary of Software Engineering Terminologies* defines embedded systems software as *a part of a larger system which performs some of the requirements of that system* [39]. By 2010, premium class vehicles are expected to contain one gigabyte on-board software [7, 69]. Current BMW 7 series implements about 270+ functions that a user interacts with, deployed over up to 67 embedded platforms [69]. Software constitutes only one part in embedded systems. Integration of these subsystems is always a challenging task for complex systems [69]. Additionally, other issues that needs to be addressed includes unstable requirements, technology changes, location of software errors, and inadequate documentation [70]. According to Ebert, in 2008, there were about 30 embedded microprocessors per person in developed countries with at least 2.5 million function points of embedded software. The volume of embedded software is increasing at 10 to 20% per year depending

on the domain [7]. Embedded software developers face many challenges in their work like conflicts in the requirements placed on them. For instance, low memory usage while ensuring high availability [71]. Consider that the role of embedded systems is often critical, the importance of managing software quality is therefore necessary to deliver software in a useful, safe, and reliable way. Table 2.2 in Section 2.2 describes the ISO/IEC 9126:2001 quality attributes.

Embedded Software Quality

Embedded software has many common characteristics with traditional computer-software. However, compared to traditional computer-software development, embedded software development has been proven to be more difficult [7, 37]. Sherman [37] points out that is important that embedded systems not only meet the functional requirements of 'what' they should do, but also the non-functional or 'quality' requirements expected of them. Non-functional requirements have come to be referred to as software quality attributes. Ebert et al. [7] states that quality is difficult to measure in embedded systems. Concerning the design of embedded systems, often discussed behaviors are related to run-time qualities depending on the system domain. For instance, reliability and safety qualities are important in automotive domain [69], while performance and reliability qualities are important in mission critical systems [32]. However, other important design-time qualities directly related to business goals, such as maintainability and usability, are often compromised compared to run-time qualities.

Graaf et al. [6] studied seven European firms to determine the state of the practice in embedded software engineering. One of the key findings in the study was that system engineering was mostly driven by hardware constraints. Software development did not start until the project hit a phase where hardware development changes would be expensive. This led to suboptimal solutions in terms of software architecture, because the system architecture was more or less fixed. Existing software development technologies do not consider the specific needs of embedded systems development.

Risks and malfunctions in embedded software are much higher than in traditional application software. Security rapidly grows in relevance as embedded software communicates autonomously with other computing devices [7]. In the automotive industry, more and more vehicles are getting connected to the cyber-infrastructure [69]. This has resulted in attacks carried out via this cyber-infrastructure. For instance, earlier in 2015, two researchers discovered the possibility to start Tesla Model S using a laptop².

Despite the growth of embedded systems, research has revealed that many embedded system projects do not start from scratch [6, 69]. Several authors have pointed out that software reuse is applied a lot in the embedded system industry. According to Graaf et al.

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

[6], software reuse is rather ad hoc. Most of the products that is developed today is based on previous products. Reuse artifacts, such as requirements and code, are therefore reused by copying them. Furthermore, Pretschner et al. [69] states that the difference between functionality changes from one vehicle generation to the next is relatively small. Most of the old functionality remains and can be found in a new car generation. Functionality differs mostly not more than 10%. However, more than 10% of the software is re-written.

Given the lack of focus on design-time aspects of SQ, the amount of technical debt accumulated in embedded systems grows continuously. This happens because of several factors, and especially because the current level of technical debt is not visible to the developers and managers. According to Graaf et al. [6] and Ebert et al. [7], insufficient requirements and testing are two major cost drivers in embedded software development. 40% of all software defects in embedded systems results from insufficient requirements, especially non-functional requirements [6, 7, 45]. Furthermore, testing after code completion consumes 30 to 40% of embedded development resources, and depending on the project life cycle, testing requires a lead time of 15-50% of total project duration [7]. Moreover, time pressure from the management has been identified as a negative impact on embedded projects, which usually leads to suboptimal solutions.

CHAPTER 3

RESEARCH METHOD

This chapters presents the main empirical strategies used in this research.

3.1 Empirical Strategies

Empirical studies follows two types of research paradigms; the qualitative, and the quantitative paradigm [72]. Qualitative research is concerned with studying objects in their natural setting [72]. Its data include non-numeric data found in sources as interview tapes, documents, or developers' models [2]. Quantitative research is concerned with quantifying a relationship or to compare two or more groups [72]. It is based on collecting numerical data [2].

Oates [2] 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, explorative 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.

3.1.1 Choice of Methods

Out of the six research strategies presented by Oates [2], survey was chosen as a strategy for this research. As mentioned before, questionnaires and interviews are two common means for data collection [72]. The method chosen is usually based on type of results researches are looking for. There are three types of interviews [2]:

- Structured interviews: The use of pre-determined, standardized, identical questions for every interviewee. Structured interviews are very similar to questionnaires.
- Semi-structured interviews: A list of themes and questions is prepared. However, the interviewer has the possibility to ask additional questions that is not included in the list.
- Unstructured interviews: The topic is introduced by the interviewer, and the participants talk freely about related events, behavior, or beliefs. The researcher has less control.

In this research, semi-structured interviews will be conducted as the supporting research method for qualitative data collections. Some advantages of using interviews are best trade-off between usage of time, and its ability to provide detailed textual description for the research questions. Most of the questions were formulated in an open fashion way, which allowed the participants to speak with more detail on the issue with their experiences and viewpoints [2, 72], hence resulting in qualitative and flexible answers.

3.1.2 Data collection

This research was performed by collecting data through a series of interviews with companies in the electronics and software business. The interviews focused around how companies encountered technical debt, how they are addressing it, what they were doing to handle it through the development and software evolution. In total, four interviews were conducted with four different companies, two of them from the embedded system industry, and the last two from traditional software development industry.

The companies to interview were chosen by the author and the supervisor of this project. The interviews were mainly conducted in Trondheim, because the companies could easily be interviewed in person. A proper request for participation for an interview were sent out to companies through e-mail. The participants were chosen by the companies, but the type of person needed for the interview were given by the author.

The interviews were conducted in a semistructured manner. The benefit of using a semistructured approach is that the interviewer are able to change the order of ques-

tions depending on the flow of the conversation, and to probe deeper into a subject by asking additional questions the interviewer had not prepared for [2].

The interview questions were created by the author with the assistance of the supervisor. These questions were used as a guideline to get an overview over the subjects to be asked, and appropriate follow-up questions were asked to gain in-depth understanding about the subject. The interviewer took notes while the interview was in progress.

A certain degree of structure in the interviews provided a basis for comparing the interviews afterwards. Covering the same main topics makes it possible to extract common trends and differences in the answers. After the interviews were conducted, it was analyzed by arranging the interview data in tables containing the most relevant characteristics, and conclusions were drawn from analyzing it.

This chapter presents the results of this study. The first section presents the participants. The main findings are presented in Section 4.2, 4.3, 4.4, and 4.5. Section 4.2 focuses the term technical debt. Section 4.3 reveals the causes of technical debt in the various systems. Section 4.4 takes a further look at how technical debt is addressed and prioritized in organizations. Finally, Section 4.5 reveals the way technical debt is managed.

4.1 Introduction of participants

Table 4.1: The participants and their background

ID	Role	Academical degree	Experience
P1	Software Developer	Bachelor	Over 10 years
P2	ICT Security and Quality Manager	Master	Over 10 years
P3	Research and Developer Manager	Doctoral	Over 10 years
P4	Project Manager	Doctoral	Over 10 years

The participants had various background within software development field. As mentioned in Chapter 3, two of the participants works with embedded systems, and the last two works with traditional software development. Table 4.1 summarizes their background.

The interview began by asking the participants about their responsibilities with the products they are working on, type of products they develop, size of the product, size of the projec team, and software development methodology used. Table 4.2 summarizes their responses.

Table 4.2: The participants and their project

ID	Product type	Responsibilities	Size of the solution (lines)	Process	Team size
P1	Web application	Working on an API, integrates multiple systems	101k - 1m	Scrum	5 to 10
P2	ERP solution, SAP project	Maintains and manages systems	More than 10m	Lean, Scrum	More than 20
P3	Embedded systems	Creates products by integrating other products with their own core product	101k - 1m	Lean, Scrum	11 to 20
P4	Embedded systems	Project leader, creates COTS solutions	101k - 1m	Scrum	More than 20

4.2 Definition of Technical Debt

To get an understanding of what the participants meant when they talked about technical debt, they were asked to define the term.

"Technical debt is resolving things in a way so that there will be need of refactoring the solution later. Technical debt is incurred intentionally or unintentionally. It may be due to external assumptions that were not taken into account when designing the system. You can have the perfect payment system, but if you change the currency, the system may fail." - P1.

P2 defines technical debt as follows:

"Systems or software that performs under the level we have set as requirement, or is of lower quality. Like if we have systems out-of-support, then we have incurred technical debt as the system operates with bigger risk than what we want. [...] We do not want to run the latest version of a software, but a version less because someone has run it before and fixed the bugs." - P2.

The definition of technical debt given by P3 is:

"A product where components are not available, operating system is outtechnical debtated and is not supported with updates anymore. It is expensive to maintain the product, and people with the competence might not be available anymore."

P4 defines technical debt as *"things that needs to be changed before you can add new functionality"*.

Why technical debt is a problem

The literature stated that technical debt is not always bad [10]. With that in mind, the participants were asked why they consider technical debt as a problem. Both P2 and P3 explained that incurring technical debt resulted in long-term risks.

"Technical debt is a problem because all the products we deliver, will incur technical debt, if developers do not care to maintain the product, and remove unhealthy dependencies along the way that you quickly get as a result of technical debt. Another problem is that people are afraid to bring up this subject. I talked with a guy from the oil sector who told me that some of their systems has 40 years of technical debt. I think that it is time for an upgrade. The world has changed radically over the past decade and will change in the future." - P3.

P2 expressed that their organization have many legacy equipments and systems. Additionally, their systems are using legacy technologies. Legacy solutions creates troubles in the long-run. *"An example of a system we have running on Windows 95, cannot be upgraded or replaced due to compatibility problems. You cannot install Windows 95 on a new PC, because Windows 95 will not understand the drivers that the new PC needs. There are many organizations that keeps buying systems without having a plan for how to manage it later"* - P2.

P4 explained that there is always a possibility of refactoring in projects. *"Our code base is almost 10 years old, and big parts of the architecture is still the same as it was 10 years ago. I am pretty sure that there are some parts of the system that can be improved"* - P4.

P1 remarked that technical debt is now always bad.

If you create systems that does not incur technical debt, there would not be so much for developers to do later. This is the reason for agile methods being used, you create visible technical debt. Your boss might give you a task to build a house without a roof. The house looks very nice, but the day it snows, things will go bad - P1.

4.3 Causes of Technical Debt

To identify the causes of technical debt, the participants were asked the reasons for technical debt accumulation in their project.

4.3.1 Time Pressure

Time pressure was frequently mentioned as a reason for incurring technical debt. For example, P1 and P4 mentioned that if they did not incur technical debt, it ran the risk of

not being able to deliver the solution at all.

"Sometimes, we need to take shortcuts in order to meet a deadline. We do not take shortcuts in terms of bad code, but rather to complete functionality and integrate them into our solution. Less important tasks can be postponed." -

P4.

Furthermore, P2 explained that both lack of time and resources is why technical debt keeps growing in their systems. He explained that the resources they are given by the management are mostly used to implement new functionality, or to fix critical errors. They do not have the time to upgrade their existing systems.

"We cannot prioritize upgrading existing systems. The resources that are given to us, are used to implement new functionality, or to fix critical errors such as system crashes. It is not certain that technical debt is behind a system crash, but it is something we need to prioritize. Additionally, we get new projects all the time, which makes software evolution a challenge." - P2.

Moreover, P2 mentioned that one of their systems did not work properly after upgrading the operating system it was running on. Given the lack of resources to pay down technical debt, they eventually had to isolate the problem by workarounds.

System Size

As the system complexity keeps increasing, the system also becomes harder to change. Increased system complexity makes it easier and cheaper to incur technical debt in short-run. P4 explained that many of their developers did not know the system well enough, so change requests made by developers did not fit with the intended design.

Architectural Decisions

Furthermore, P1 revealed a situation where an architectural decision was the reason for technical debt accumulation. The solution looked good to begin with, but the effects of the outcome was not optimal. The architecture design was a debt itself. He further stated that while the implementation was good, the outcomes was not as expected.

Technology Choices

The participants explained that technology and framework choices, may be a source of technical debt. Both technologies and frameworks gradually become obsolete by other solutions, resulting in legacy. P3 described a situation where their core product accumulated technical debt over time. The product used lots of legacy solutions from external suppliers. Replacing these was expensive because it would require changes at the infrastructure level

of the system. However, at some point, the organization decided to manage the technical debt by re-engineering the solution using open-source solutions.

4.4 Incurring Technical Debt

The participants were asked about the current level of technical debt in their products, and the reason for it. Both P1 and P2 revealed that requirements from the management or customers are the dominant factor in determining if the available resources should be used to address technical debt, while deadlines are the dominant factor for P4.

"There are some visible and some invisible technical debt in our project. Mostly visible that should have been fixed. The reason for it is that the product is relatively new and young, which makes it more important to deliver new functionality." - P1

P2 explained that they have accumulated lots of technical debt in their product, both known and unknown. Their product consists of outtechnical debtated solutions from external suppliers. They usually fix errors by workarounds, resulting in architecture violations. Legacy components are hard to change. Another problem that was mentioned is that there is no documentation or knowledge about the components, hence making it difficult to change the internal structure of the solutions.

However, the product P3 is currently working on has accumulated small amounts of technical debt. The reason is that the product is relatively new. Their previous solution of the product had accumulated a high amount of technical debt. They ended up replacing the old solution with a new, improved solution using open-source technologies. Furthermore, P3 stated that shortcuts are taken if they need to reach a deadline. However, they always makes a plan to address it in the future. Test-driven development is used a lot thorough the development, making sure that problems are addressed immediately.

P4 remarked that they have accumulated some technical debt in their product, but not more than they are able to handle. Most of the technical debt they have accumulated are related to integration and system tests. P4 also stated that technical debt is not bad. Sometimes, technical debt needs to be incurred, to meet business goals. *"You can take a loan from the bank to buy a house, as long as you are able to pay it back"* - P4.

Some of the participants revealed that technical debt is incurred intentionally to a certain extent. P1 explained that both developers and the management makes such decisions.

"Developers tries simple solution to understand the exact problem. They usually hard code some parts of the code that could have been coded dynamically. However, hard coding results in short-term benefits, but it causes long-time problems." - P1

Short-term vs. long-term effects

The interviews revealed that technical debt may affect the software development in the short-term. Incurring technical debts by taking shortcuts, is used to save development time and deliver a solution faster to the customers. P4 explained that technical debt is not the result of poor programming skills, but as a result of intentional decisions to trade off competing concerns during development. P1, P2, and P4 mentioned that such shortcuts in development are taken, in reaction to business pressure. Another short-term effect of technical debt that was identified is the customer satisfaction. The customers are interested in getting the product on time. They do not care about the technical details of implementation as long as it does not directly affect the product quality. *"The customers does not care how they get electricity from the wall, they just want electricity"* - P2. Moreover, P4 mentioned that sometimes short-term solutions might outweigh the future costs, depending on the implementation method.

The negative effects tended to be on the longer term, such as poor performance, increased complexity, and low maintainability. P3 brought up example where one of their systems that used legacy technologies, crashed during a scaling test. P2 mentioned that some of their systems are still running on Windows 95, and they do not have enough resources to upgrade the systems. Both P1 and P2 mentioned that technical debt is also incurred intentionally. Management and the customers often comes with requirements that needs to be prioritized.

"Lets say that we have two systems with technical debt. If we have resources to manage technical debt in one of the systems, the other system needs to be postponed. These types of situations occurs frequently, and the total technical debt keeps increasing." - P2

Prioritizing Technical Debt

Another aspect with software development we found interesting is how business decisions affects the development team and the overall software quality. With regards to that, the participants were asked about to what extent the company invest resources to measure the quality of the system.

Both P1 and P2 mentioned that business decisions do have effect on the amount of technical debt.

"Customer specified functionality have to be prioritized for business purposes. Such decisions often causes architectural erosion. The software may work, but the solution is dirty." - P1

"The management is not willing to see what people are doing during work, and how much time it takes to finish a task. This leads to problems in distribution

of resources.”” - P2

P1 further mentioned that there is a communication gap between the team and the management. *Not everyone from the management is known with the term technical debt. If something works, do not touch it - P1.*

P2 explained that the management is interested in the risks behind technical debt issues, before taking a choice. Management do not care about the consequences if the risk is low, as long as it does not have any big impacts for the business.

P4 remarked that the quality on their product is remarkable. He stated that it is hard to measure the quality of their product, so they use feedbacks and reviews from the customers as a way to measure the quality.

4.5 Management of Technical Debt

Another important aspect regarding technical debt is how it is managed. The participants were asked about the importance of reducing technical debt, and how they reduce technical debt. Most of the responses involved some form of communication within the development team, and with management. It included approaches such as risk management, and use of backlogs. Some of the participants also pointed out that technical debt can be controlled by the customer, based on the needs.

Importance of reducing technical debt

A common response regarding the importance of reducing technical debt is the products ability to remain stable.

Managing technical debt by upgrading the software is not necessary a goal, it is more important to use a version of the software that is stable. A small change in the software may affect many people. [...] It is important to finish a task early as possible, and to clean up the mess before you end up with too much dependencies - P1

However, reducing technical debt is a challenge itself. P3 commented that the challenge is to get project leaders attention on various problems. *”Sometimes, there is need of refactoring technical debt. Postponing technical debt problems create long-term issues. A challenge is to get an approval from the project manager to address the issue. It is important to fix technical debt problems while it is fresh. Keep the code agile.” - P3.*

Furthermore, the participants were asked how much time the team spent on reducing technical debt. P3 and P4 remarked that fixing software bugs is something they work with all the time. The goal is to get an overview of all the bugs before product shipment to customers. P4 stated further than bugs are not considered as technical debt. technical

debt is code that needs refactoring. P1 and P2 handled technical debt in parallel with development of new functionalities. Both of them spend around 25% to 30% each sprint on maintenance and evolution.

Tools and techniques

The participants stated different ways to handle technical debt. Refactoring and re-engineering were commonly mentioned. P3 had to re-engineer the core of their product using open-source software. Additionally, various tools are used to keep track of the technical debt. The most common mentioned tools was Jira, and issue trackers. However, one of the participants commented that they do not have any special tools for technical debt management.

We do not use any special tools or techniques to manage technical debt. We do have a system- and a service catalog. These catalogs gives an overview over all the systems we are working with. It displays information such as hardware and software version that is used, and age of the system. Administrators use these catalogs to create tasks for the different projects that are on-going. If some software or hardware is very old, the issues are addressed by risk management
- P2.

P3 considers technical debt issues all the time thorough the development. By using test driven development, they are able to address technical debt issues relatively fast. However, issues are raised if it is something important. Their goal is to fix it the next release.

The participants were asked how they would like to manage technical debt. P1 mentioned that encapsulating the code might help in the short-term. Software architecture does not get affected by encapsulation. Raise an issue, and put it on the backlog. Fix it the next sprint. Both P2 and P3 would like to use a fixed budget for managing technical debt. Using the budget, they would prioritize the tasks based on what effects it has for the customer. *"Systems are getting older and older each day, and the quality is getting worse. You should think about replacing it"* - P3. P4 would like to spend more time on managing technical debt on each sprint. Ideally, 20% of the time on each sprint.

The purpose of this study was to do a broader review of how technical debt in embedded systems.

5.1 Definitions of Technical Debt

The definitions of technical debt given by the participants have many similarities with the literature. technical debt was defined in terms of code refactoring, which matches Cunningham's definition of technical debt [8]. Furthermore, technical debt was also defined in terms of using legacy solutions. McConnell classifies technical debt as intentional and unintentional debt [13]. By looking at the material from the interviews, it is possible to see that technical debt was mostly associated with intentional debt. One example was that developers experimented different solutions to a problem by hard-coding. Solving problems by hard-coding results in short-term benefits, but refactoring is needed at a later point. Moreover, the interview data revealed that technical debt is not always bad. In some situations, intentional debt brought success in terms of reaching goals, such as delivering products quickly to customers. Additionally, the interview data revealed some situations where technical debt was incurred unintentionally. Such decisions affected the overall software quality.

Workaround is not something that has been mentioned by the literature, but it is enough to describe as some of the symptoms of technical debt. Workarounds are related to code design which is not obvious and easy to understand. It can be used to bypass difficult points in the code which may result in technical debt. Moreover, workarounds can be taken in other phases in software development, which makes the concept much closer to technical debt. As described in one of the interviews, workarounds were often taken in legacy systems.

5.2 Causes of Technical Debt

Time pressure was revealed as a common cause for technical debt accumulation. Issues with time has been present in multiple articles as a reason for incurring technical debt [8,16,18,19,22,24]. Moreover, Ebert et al. [7] states that companies compress schedules to a point that makes engineers compromise design-time qualities to run-time qualities. Time issues ultimately comes from business realities that needs to be met based on customer needs and market situations.

Time issues may lead to communication problems within the team, thus incurring technical debt intentionally. Fowler defines this situation as reckless and deliberate technical debt [14]. This could be due to stress or lack of knowledge. As the project moves along, the debt will eventually surface, and when it does it will suddenly need to be repaid. Nevertheless, the findings revealed that time pressure was manageable.

Furthermore, the research identified that technical debt is connected with many different aspects in the software development life cycle. Table 2.1 lists the subcategories of technical debt. We clearly see that there many similarities between. When the problem of technical debt is divided into subcategories, it is easier to become aware of the problems of technical debt. An architectural solution resulted in bad outcomes. This is very similar to architectural debt, and unintentional debt. Software architecture is one of the artifacts that is hard to change, and therefore the debt becomes much higher. Code refactoring is needed before new functionalities could be added. This is very similar to code debt [12]. The interviews also revealed some lack of tests, and that shortcuts were also taken in test code. This is known as test debt [12]. Use of older technologies and framework are related to infrastructure debt [12]. The findings also revealed lack of documentation in some of the third-party solutions, known as documentation debt [12]. By dividing technical debt problems into categories based on where and why they occur, the actual problem would become more specific and this easier to grasp. This makes it easier to allocate respectability for making sure that debt is correctly managed.

Poor choices of technologies and third-party solutions from external suppliers can be considered as a debt generating activity. The participants expressed how technical debt accumulates when suppliers are going out of business, or are unable to support the product with updates. It is very rare for companies to select a poor implementation intentionally, to get short term benefits. However, poor technology choices will eventually surface as the system grows and its bottlenecks are discovered. It is something that cannot be planned for, and requires time and effort to fix one discovered. When choosing technologies or third-party solutions from external suppliers, it is important that developers has the competence to use the technologies, or that enough documentation follows the solutions that is bought from external suppliers. One of the participants mentioned that lack of documentation led to major workarounds later on.

5.3 Incurring Technical Debt

Observations from the interviews indicated that there is a difference on incurring technical debt between developers and management. Project managers were more likely to incur technical debt, because the realization of meeting their business goals, without any plans on paying it back. It also revealed that if a product is working, it should not be touched. This indicates that the management does not trust the developers. Moreover, from the developer perspective, the management remains largely unaware of technical debt, and they cannot see the value behind technical debt management. It makes it hard to convince the management. Overall, the technical communication gap between the business and development departments is high. Klinger et al. [11] found similar causes in their IBM study. Based on this, we believe that larger companies has more challenges dealing with technical debt issues due to complex communication structure. We believe that management with no knowledge about technical debt is responsible for technical debt accumulation in this research. Development teams may incur small amount of technical debt, but not more than what they are able to handle.

Several studies argue that the short-term effect of time-to-market is a good thing about technical debt [19, 21]. The interviews revealed similar situations, where technical debt were used to deliver a solution faster to the customer, resulting in customer satisfaction. We believe that the customer and management will demand more over time, causing accumulation of technical debt, leaving them unhandled. Moreover, the long-term effects of technical debt tends to have more negative effects [19, 21, 23]. The findings revealed that technical debt in the long-term started to generate extra working hours, performance issues, scalability errors, and system failures. By taking technical debt, things might turn into a problem later if they are not paid back. This might be a reason for why the amount of technical debt has doubled since 2010 [9]. Business people often thinks that technical debt is something that can be incurred to reach a deadline, and just fix it later. This is very similar to reckless and deliberate debt [14].

The findings also pointed out difficulties with ongoing projects, where each project has low amount of available resources. It seems like the participants have problems on balancing technical debt in parallel with development of new functionality. Many of their systems does not get updated, which creates problems in the long-term.

5.4 Management of Technical Debt

Although technical debt is not considered in different projects, it is still managed thorough the development and evolution by using different practices. One of the practices that was mentioned is very similar to managing risks. Like risk management, technical debt management is a balancing act that aims to achieve a level of good quality while mitigating its failures. In particular, the participants had to not only evaluate the technical implications

before making a choice, but also the impact on the delivered business value. For example, such trade-offs included release of the product in time to capture the market share. This requires involvement of all the stakeholders. The impact and consequence of technical debt can be used to convince the stakeholders to agree upon the same strategy for managing technical debt. A similar strategy has been defined in the literature [19, 73]. The author believe that closing the communication gap between technical and non-technical stakeholders is important to increase the visibility of technical debt.

It was evident from the interviews that communication within the team played a big part in achieving higher software quality. A common practice to handle technical debt issues expressed by the participants, was to make sure that the developers are aware of the technical debt issues. For example, if a company knows that there are some lack of tests, or that a system has performance issues, the debt would be less significant. The different development team used a backlog to collect technical debt issues and their risk in parallel with new functionalities to be implemented. technical debt issues can be taken into account when planning feature implementation, which lessens the impact of the debt. Similar strategy has been suggested by Krutchen et. al [15]. A portfolio management strategy has been proposed by Guo et. al [20], where technical debt is stored in a backlog and development team can use that for management of technical debt. This backlog strategy may be beneficial in the long-run when older technical debt is traceable, instead of of forgotten.

The use of test-driven development was identified as a way to manage technical debt thorough the development. Using test-driven development makes sure that bugs are discovered in the software. This improves the quality of the software. Test-driven development has been suggested by Ebert et. al [7] as a practice to improve the software quality.

A common approach to keep the debt from growing over time is to conduct refactoring and re-engineering. Both refactoring and re-engineer has mentioned by the interview participants, and the literature [24]. Code refactoring was applied by developers in parallel with development. Sometimes, developers spent evenings on code refactoring. The author believes that refactoring keeps the software quality stable, and mitigates technical debt issues.

Besides the use of practices that has been mentioned, it would be preferable to set apart a certain amount of time during each iteration, to address and manage technical debt. Similar practice has been suggested by Codabux et. al [24]. These measures may not have a big impact for the first few iterations, but one would be able to see improvements in quality and productivity over time.

5.5 Research Questions

It is interesting to see that the way embedded system developers and traditional software developers handles technical debt. This research revealed that technical debt in traditional software projects are much higher than the embedded system projects. Despite the re-engineered product by one of the participants using open-source technologies, they were at least aware of the amount of accumulated technical debt in their old solution. Fowler defines this as inadvertent/prudent debt [14]. Additionally, this indicates that technical debt and software quality is taken much more seriously by the embedded system industry. Research has revealed that embedded software cannot contain any errors [7, 32, 69].

Despite the fact that embedded system developers are more serious addressing technical debt, this research also revealed that embedded system projects have some technical debt in their project. However, the amount of technical debt accumulated in the various products are manageable. The most important thing is that the product is stable, and the quality is good. By using open-source solutions, or developing own frameworks gives lots of benefits. Frameworks that has been developed in-house can be reused by other projects. However, developing own solutions poses risks. One of the participants pointed out that their testing framework were out-of-date. As mentioned in Section 2.6, reusing out-of-date solutions is the root cause of technical debt accumulation.

The research questions that were outlined in Chapter 1 are following:

- **RQ1:** What practices and tools exists for managing technical debt? How are they used?
- **RQ2:** What are the most significant sources of technical debt?
- **RQ3:** When should a technical debt be paid?
- **RQ4:** Who is responsible for deciding whether to incur, or pay off technical debt?

Table 5.1 contains a summary of the results of this research. Although, the results did not specify any specific approach for managing technical debt, this research were able to identify some practices for reducing technical debt. Moreover, this research were able to identify different sources for technical debt. This study was also able to identify when technical debt should be paid off, and the people responsible for taking such decisions.

5.6 Threats to Validity

Validity is related to how much the results can be trusted [72]. Wohlin et al. [72] states that adequate validity refers to that the results should be valid for the population of interest. First of all, the results should not be valid for the population from which the sample is drawn. Secondly, it may be of interest to generalize the results to a broader population.

Table 5.1: Research Questions and Their Findings

RQ1	Test-driven development: One of the participants used this methodology to address technical debt thorough development.
RQ2	Lack of time given for development was identified as the primary source for incurring technical debt. Time pressure was often caused by business decisions. Other sources of technical debt that were identified are lack of tests, architectural choice, and technology choices.
RQ3	The risk of technical debt items determined whether companies should invest resources in paying down the debt. If the risk was high, an evaluation of the risk was made. Additionally, the results also outlined that some debts does not need to be paid off.
RQ4	A combination of management and the customer were mostly responsible to incur, or pay off technical debt. The risk behind debt was measured before deciding whether to pay off or not. This research has also revealed situations where developers incurred technical debt, both intentionally and unintentionally.

Wohlin et al. [72] describes four types of validity threats; internal, external, construct, and conclusion validity.

5.6.1 Internal validity

Internal validity concerns about the "right data" [72]. If the outcome is caused by the treatment and not by other factors not measured, we can conclude that the result has internal validity. The relevant threats for this research are:

- **Maturation:** The participants may be affected negatively during the interview. They could be tired, or not motivated to answer some questions.
- **Instrumentation:** This effect is caused if the artifacts of the interview is badly designed. We do not think that the interviews were badly designed, and helped the participants with additional information if needed.

5.6.2 External Validity

External validity is the degree to which the results of an experiment can be generalized outside the experiment setting. External validity is affected by the chosen experiment design, the objects in the experiment, and the subjects chosen. There are three main risks: people, place, and time. This research is concerned about people. Only 4 people were able to participate in the interviews, and they are not possibly not representative for the larger population. Additionally, only two of the participants were from the embedded system industry. We cannot conclude that the results from this research reflects a more general trend. More studies will be necessary to generalize the results.

5.6.3 Construct Validity

Threats to construct validity refer to the extent to which the experiment setting actually reflects the construct under study. Questions that were asked during the interviews may have been misunderstood because they were improperly phrased. The participants may therefore answer something else. After conducting and analyzing the interviews, we realized that some questions could have been better phrased to gain deeper information about the topic. The interview guide was reviewed by the supervisor, but we did not test the interview guide before the interviews.

5.6.4 Conclusion Validity

Conclusion validity concerns the ability to draw correct conclusions from the relationship between the treatment and the outcome [72]. Because we only had 4 participants, the statistical power is very low. This is something we are aware of, and therefore more thorough studies need to be performed to confirm if our results have more general applicability. Additionally, the authors lack of experience creating the interview guide may affect the conclusion as well.

This research presents the results of empirical studies that has investigated technical debt in embedded systems. This study has revealed that technical debt is something that organizations are unable to avoid during software development projects. However, technical debt is not always a bad thing to take. Organizations can use it as a powerful tool to reach their customers faster and gain edge over the competition in the market. However, if technical debt is not paid back in time, it may generate economic consequences and quality issues to the software. It is necessary for organizations to create a strategy plan that includes practices and tools that decrease technical debt.

Empirical research has been performed to verify theories. This study is mainly used to gain an understanding about technical debt accumulation in embedded systems with the goal to find the best practices for managing technical debt. This study reveals that the way traditional software developers and embedded system developer handle technical debt is different. The reason is that embedded system developers needs to deal with multiple constraints. This research also revealed that the awareness of developers and managers regarding technical debt is an important factor. When they are aware of the technical debt, it is less dangerous as it can be accounted for when planning for the future. Unknown and hidden debt is more dangerous. Having the ability to predict the long-term business outcomes of short-term technical decisions would help the software practitioners to choose the right kinds of technical debt to incur.

As mentioned in Chapter 5, Section 5.6, the results cannot be generalized, since the amount of respondents are low. However, the observations from this study are very important for future research. There are still large amount of future work to be done in this subject area to develop strategies and tools that can help teams successfully managing their debt.

6.1 Future Work

The future work will consist of the upcoming Master's Thesis, which will have a limit of 20 weeks. There are still plenty of room to look deeper into a more narrow field. Conducting a deeper case study with several companies, studying their systems, is a possibility for the upcoming Master's Thesis. There will be meetings at the start of the next semester with the supervisor to discuss the work for the Master's Thesis.

BIBLIOGRAPHY

- [1] Gartner, “Gartner Says 4.9 Billion Connected ”Things” Will Be in Use in 2015,” 2014.
- [2] B. J. Oates, *Researching Information Systems and Computing*. Sage Publications Ltd., 2006.
- [3] J. Highsmith, “The financial implications of technical debt,” 2010.
- [4] H. v. Vliet, *Software Engineering: Principles and Practice*. Wiley Publishing, 3rd ed., 2008.
- [5] F. Mattern and C. Floerkemeier, “From the internet of computers to the internet of things,” in *From active data management to event-based systems and more*, pp. 242–259, Springer, 2010.
- [6] 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.
- [7] C. Ebert and C. Jones, “Embedded software: Facts, figures, and future,” *Computer*, no. 4, pp. 42–52, 2009.
- [8] W. Cunningham, “The wycash portfolio management system,” *SIGPLAN OOPS Mess.*, vol. 4, pp. 29–30, Dec. 1992.
- [9] 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.
- [10] 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.
- [11] 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.

- [12] 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.
- [13] S. McConnell, “Technical debt,” 2007.
- [14] M. Fowler, “Technical Debt Quadrant,” 2009.
- [15] P. Kruchten, R. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *Software, IEEE*, vol. 29, pp. 18–21, Nov 2012.
- [16] E. Allman, “Managing technical debt,” *Commun. ACM*, vol. 55, pp. 50–55, May 2012.
- [17] 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.
- [18] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. McCormack, 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] F. Buschmann, “To pay or not to pay technical debt,” *Software, IEEE*, vol. 28, no. 6, pp. 29–31, 2011.
- [24] 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.
- [25] 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.

- [26] 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.
- [27] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, (New York, NY, USA), pp. 1–8, ACM, 2011.
- [28] S. SonarSource, "Sonarqube," tech. rep., Technical report, last update: June, 2013.
- [29] J.-L. Letouzey, "The sqale method for evaluating technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*, pp. 31–36, IEEE Press, 2012.
- [30] R. Pressman, *Software Engineering: A Practitioner's Approach*. New York, NY, USA: McGraw-Hill, Inc., 7 ed., 2010.
- [31] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.
- [32] 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.
- [33] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, 3rd ed., 2012.
- [34] 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.
- [35] D. E. Perry, "State of the art: Software architecture," in *International Conference on Software Engineering*, vol. 19, pp. 590–591, IEEE COMPUTER SOCIETY, 1997.
- [36] S. Rugaber, "Software architecture and design."
- [37] T. Sherman, "Quality attributes for embedded systems," in *Advances in Computer and Information Sciences and Engineering*, pp. 536–539, Springer, 2008.
- [38] "IEEE Standard for Developing Software Life Cycle Processes," *IEEE Std 1074-1991*, 1992.
- [39] J. Radatz, A. Geraci, and F. Katki, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std*, vol. 610121990, no. 121990, p. 3, 1990.
- [40] J. Krogstie, *Model-based development and evolution of information systems: A Quality Approach*. Springer Science & Business Media, 2012.

- [41] H. Hijazi, T. Khmour, and A. Alarabeyyat, “A review of risk management in different software development methodologies,” *International Journal of Computer Applications*, vol. 45, no. 7, pp. 8–12, 2012.
- [42] P. Abrahamsson, *Agile Software Development Methods: Review and Analysis (VTT publications)*. 2002.
- [43] A. Alliance, “Agile manifesto,” *Online at <http://www.agilemanifesto.org>*, vol. 6, no. 6.1, 2001.
- [44] I. Sommerville, *Software Engineering (9th Edition)*. Pearson Addison Wesley, 2011.
- [45] H. Washizaki, Y. Kobayashi, H. Watanabe, E. Nakajima, Y. Hagiwara, K. Hiranabe, and K. Fukuda, “Quality evaluation of embedded software in robot software design contest,” *Progress in Informatics*, vol. 5, pp. 35–47, 2007.
- [46] M. Poppendieck, “Lean software development,” in *Companion to the Proceedings of the 29th International Conference on Software Engineering, ICSE COMPANION ’07*, (Washington, DC, USA), pp. 165–166, IEEE Computer Society, 2007.
- [47] O. Cawley, X. Wang, and I. Richardson, “Lean/agile software development methodologies in regulated environments—state of the art,” in *Lean Enterprise Software and Systems*, pp. 31–36, Springer, 2010.
- [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] W. Frakes and C. Terry, “Software reuse: metrics and models,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 415–435, 1996.
- [52] O. P. N. Slyngstad, A. Gupta, R. Conradi, P. Mohagheghi, H. Rønneberg, and E. Landre, “An empirical study of developers views on software reuse in statoil asa,” in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE ’06*, (New York, NY, USA), pp. 242–251, ACM, 2006.
- [53] W. C. Lim, “Effects of reuse on quality, productivity, and economics,” *Software, IEEE*, vol. 11, no. 5, pp. 23–30, 1994.
- [54] J. Sametinger, *Software engineering with reusable components*. Springer Science & Business Media, 1997.

- [55] M. Al Mamun, C. Berger, and J. Hansson, “Explicating, understanding, and managing technical debt from self-driving miniature car projects,” in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pp. 11–18, Sept 2014.
- [56] M. Morisio, M. Ezran, and C. Tully, “Success and failure factors in software reuse,” *Software Engineering, IEEE Transactions on*, vol. 28, pp. 340–357, Apr 2002.
- [57] *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [58] M. Kim, T. Zimmermann, and N. Nagappan, “A field study of refactoring challenges and benefits,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE ’12*, (New York, NY, USA), pp. 50:1–50:11, ACM, 2012.
- [59] T. Mens and T. Tourwe, “A survey of software refactoring,” *Software Engineering, IEEE Transactions on*, vol. 30, pp. 126–139, Feb 2004.
- [60] K. H. Bennett and V. T. Rajlich, “Software maintenance and evolution: a roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, pp. 73–87, ACM, 2000.
- [61] S. Dart, “Concepts in configuration management systems,” in *Proceedings of the 3rd international workshop on Software configuration management*, pp. 1–18, ACM, 1991.
- [62] J. Estublier, “Software configuration management: A roadmap,” in *Proceedings of the Conference on The Future of Software Engineering, ICSE ’00*, (New York, NY, USA), pp. 279–289, ACM, 2000.
- [63] I. Crnkovic, “Component-based software engineering for embedded systems,” in *Proceedings of the 27th International Conference on Software Engineering, ICSE ’05*, (New York, NY, USA), pp. 712–713, ACM, 2005.
- [64] I. Crnkovic, “Component-based approach for embedded systems,” in *Ninth International Workshop on Component-Oriented Programming (WCOP)*, 2004.
- [65] P. Koopman, “Embedded system design issues (the rest of the story),” in *Computer Design: VLSI in Computers and Processors, 1996. ICCD ’96. Proceedings., 1996 IEEE International Conference on*, pp. 310–317, Oct 1996.
- [66] H. Kopetz, “The complexity challenge in embedded system design,” in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pp. 3–12, May 2008.
- [67] Z. You, “The reliability analysis of embedded systems,” in *Information Science and Cloud Computing Companion (ISCC-C), 2013 International Conference on*, pp. 458–462, IEEE, 2013.

- [68] S. Patil and L. Kapaleshwari, “Embedded software-issues and challenges,” tech. rep., SAE Technical Paper, 2009.
- [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] M. Jiménez, R. Palomera, and I. Couvertier, *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. SpringerLink : Bücher, Springer, 2013.
- [71] A. Vulgarakis and C. Secleanu, “Embedded systems resources: Views on modeling and analysis,” in *Computer Software and Applications, 2008. COMPSAC’08. 32nd Annual IEEE International*, pp. 1321–1328, IEEE, 2008.
- [72] 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.
- [73] T. Theodoropoulos, M. Hofberg, and D. Kern, “Technical debt from the stakeholder perspective,” in *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD ’11, (New York, NY, USA), pp. 43–46, ACM, 2011.