NTNU

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 the size and the complexity of the software is increasing with time, 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 idenfitied 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 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 how it is managed in embedded systems.

RESEARCH METHOD: Four different companies with one participant from each, were interviewed. Two of them works with traditional software development, while the last two works with embedded system development. All of them had different positions.

RESULTS: The results consists of the knowledge similar to the technical debt research. There are many similarities between the causes and management practices of techincal debt. Results show that technical debt is mostly formed as a result of intentional decisions to reach deadlines. However, embedded system has more control over technical debt than traditional software. 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 also 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, Specializion Project for Software Engineering, at Norwegian University of Science and Technology.

This work was supervised by Dr. Carl-Fredrik Sørensen. He helped me very much by providing insightful input and valuable feedback during the period. He also helped me with providing most names of IT companies in Trondheim along with their contact details.

We have performed four interviews during this period. We would like to thank all the respondents who took time to participate in the interview.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

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 segment of the software industry. The automotive industry is going through major transistion 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 has estimated that in 2020, 25 billion connected "things" will be in use [1]. Our society has come to depend on such systems for its day-to-day operation, and with this trend ongoing, we clearly see that most future computing systems will be embedded systems [6].

Software plays an important role in the development of embedded systems, and 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. To provide more functionality, multiple components are combined together within embedded systems. However, as the complexity of embedded system increases, the ability to maintain the required quality of such systems becomes more difficult. Combination of multiple components leads to higher costs of verifying additional software, and many consequencly may fail to test the product properly and deliver a reliable product.

Embedded systems expected lifetime goes beoynd one decade for many systems, which requires managing old systems in parallell to the design and implementation of new systems.

In particular, many companies are forced to think about their time-to-market strategy to keep up with the increased competition. This leads companies to decide what shortcuts in the development process they have to take. Such compromises causes the creation of a financial overhead in the future maintenance activities, usually termed as technical debt [7]. As technical debt accumulates, it becomes necessary to manage the overall debt while keeping the system flexible and extendable. Companies must often recall their products. If the companies could catch software defects earlier in the system design process, their income would be saved. It is important to find out how to make decisions so future maintenance and evolution has as low cost as possible.

Technical debt is a rising problem. Gartner has estimated that the cost of dealing with technical debt threatens to grow to $ 1 trillion globally by 2015 [8]. That is the double of the amount of technical debt in 2010. IT management teams must measure the level of technical debt in their organization and develop a strategy to deal with technical debt.

Table 1.1: Table from Gartner [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 project is to increase the knowledge on the significant sources of technical debt, and find out how technical debt in embedded systems are managed. 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 will be:

- **RQ-1**: What practices and tools exists for managing technical debt? How are they used?
- **RQ-2**: What are the most significant sources of technical debt?
- **RQ-3**: When should a technical debt be paid?
- **RQ-4**: Who is responsible for deciding whether to incur, or pay off technical debt?

1.3 Research Method

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

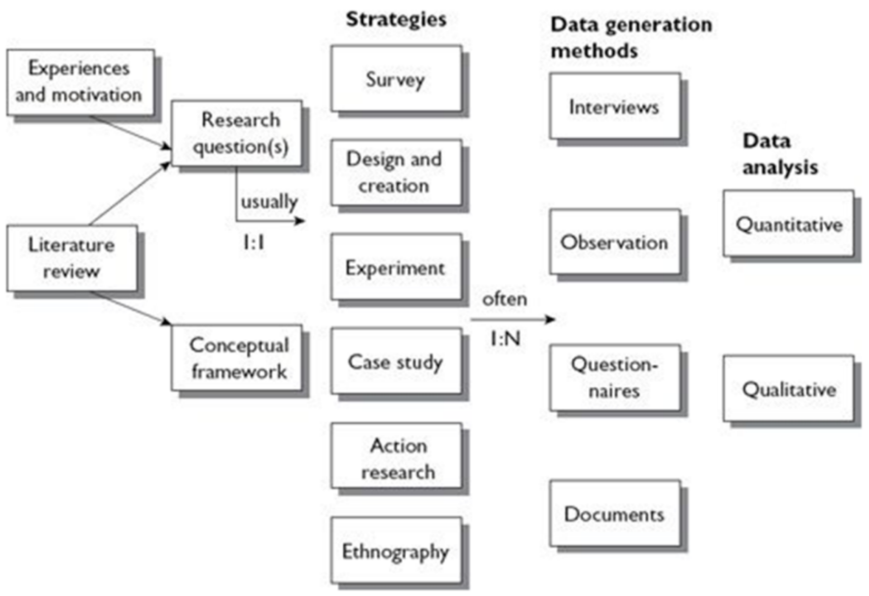


Figure 1.1: Model of research process [2]

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, or use experiences and motivations. This provides a conceptual framework for this research. A research strategy is needed to answer the research questions. There are 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. Oates presents four different data collection methods [2]: 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.

CHAPTER 2 STATE-OF-THE-ART

This chapter presentes topics which are relevant to this project. Section 2.1 looks into technical debt with its definitions, types etc. Section 2.2 looks into embedded systems and some of the challenges with it. Section 2.3 presents

2.1 Technical Debt

The concept of technical debt was first introduced by Ward Cunningham in 1992 to communicate the problem with non-technical stakeholders [7]. 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 leading to technical debt had to be made, which affect future development activities. Cunningham further 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 time goes, technical debt accumulates interest leading to increased costs of a software system [9, 10]. However, not all debts are necessarily bad. A small portion of debt might help developers speed up the development process in the short-term [9].

Figure 2.1 illustrates what happens as technical debt grows over time within a software product. Once we are on the far right of the curve, all choices are hard. The software controls us more than we control it.

2.1.1 Definitions of Technical Debt

McConnell describes TD 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

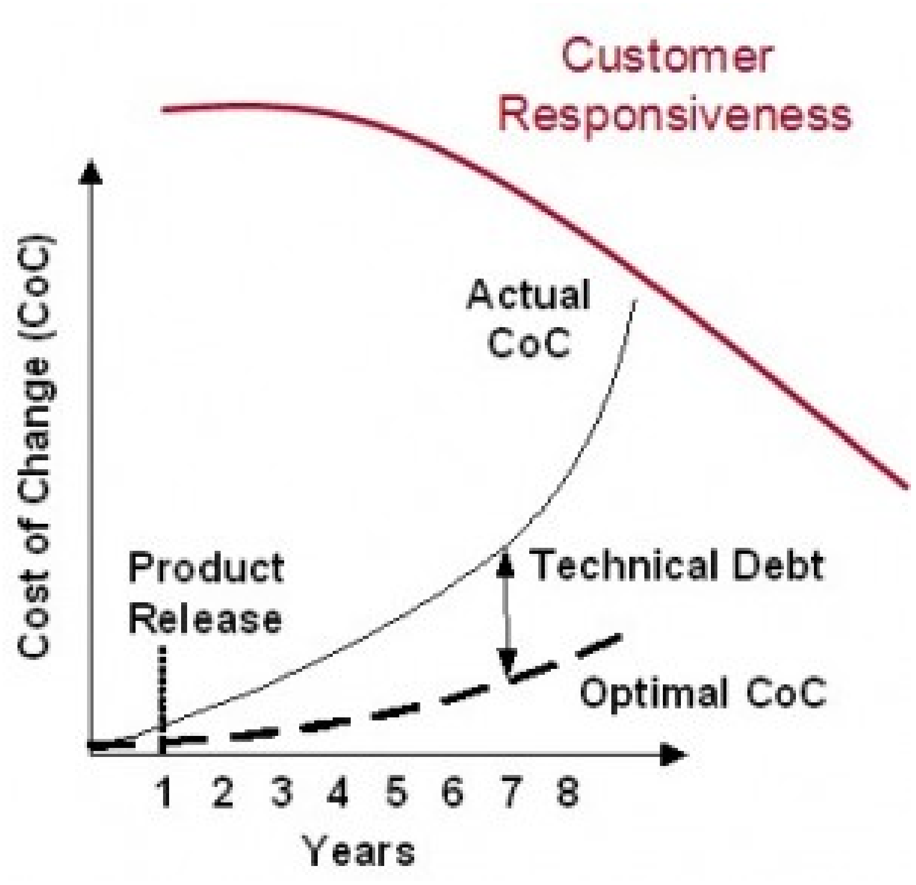


Figure 2.1: The Technical Debt Curve [3]

further splits the term into two categories based on how they are incurred, intentionally or unintentionally [11]. The unintentional category includes debt that comes from doing a poor job. For example, uninntentional debt might be when a junior software developer writes bad code due to lack of knowledge and experience. Intentional debt occurs when an organization makes a decision to optimize for the present rather than the future. An example is when the project release must be done on time, or else there will not be a next release. This leads to bad decisions, like taking a shortcut to solve a problem, and then reconcile the problem after shipment

Fowlers presents a more formal explanation of how technical debt can occur [12]. 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 when it comes to code and design is ignored, and a big mess in the codebase is made.
- **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.

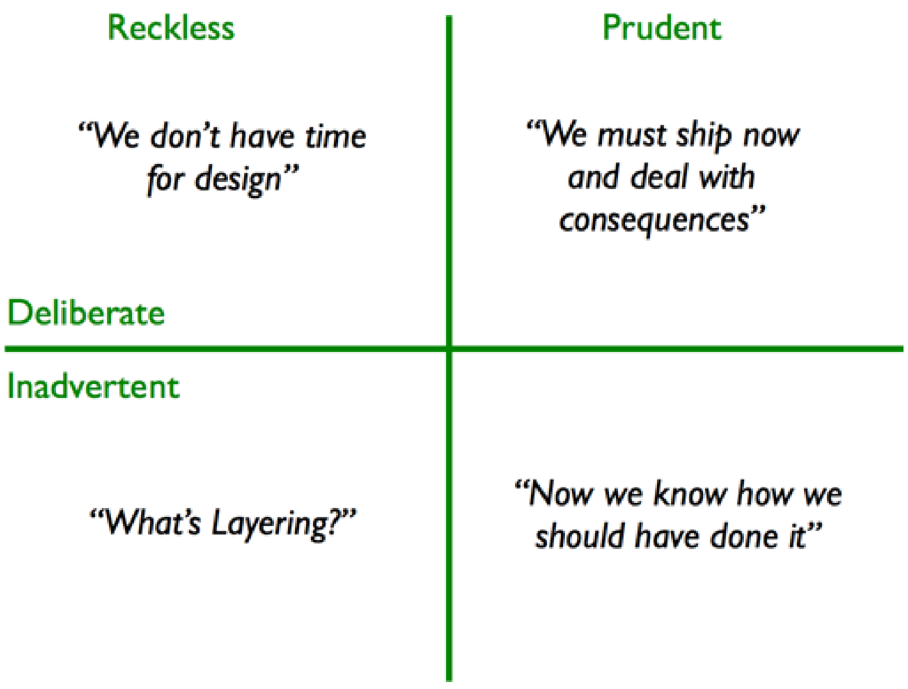

Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 2.2: Technical Debt Quadrant

- **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 might realize that the design could have been better.

Krutchen divides technical debt into two categories [13]. Visible debt that is visible for everyone. It containts elements such as new functionality to add and defects to fix. Invisible is the other category, debt that is only visible to software developers. Figure 2.3 shows a map of the "technical debt landscape" which helps us to distinguish visible and invisible elements. On the left side of Figure 2.3, TD mostly affects the evolvability of the software system, while on the right it mainly affects maintainability.

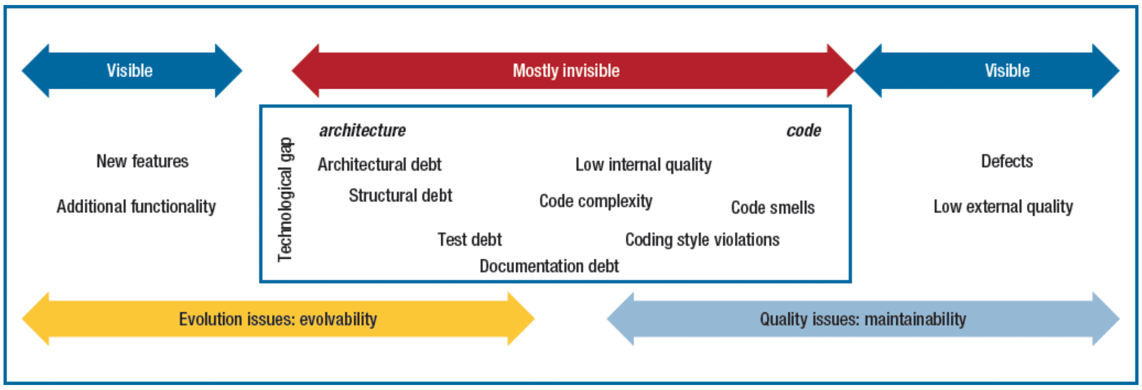


Figure 2.3: Technical Debt Landscaape

2.1.2 Comparison with financial debt

Technical debt has many similarities to financial debt [14, 15]:

- 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 loose your house or car.

Technical debt is in a way similar. Like financial debt, technical debt accrues interest over time which comes in the form of extra effort that have to be dedicated in future development because of bad choices [9,10]. You can choose to continue paying the interest, or you can pay down the debt by refactoring the code or system into something better which reduces interest payments in the future [12]. If the debt is not repaid, development might slow down, e.g, due to poor maintainability of the code. This can lead to software project failure and you might go bankrupt [14]. There are some differences between financial and technical debt as well. The debt has to be repaid eventually, but not on any fixed schedule [14]. This means that some debts may never have to be paid back, which depends on the interest and the cost of paying back the debt [16].

Technical debt is not only about bad code design. In practice, it's much more than that. Example on interests might be lower pace of development, low competitiveness, security flaws on the system, loss of developers and their expertise, poor internal collaboration environment, dissatisfied customers, and loss of market share [14].

2.1.3 Causes and effects of technical debt

Technical debt is connected with many different aspects in the software development process, like documentation debt, requirements debt, architecture debt, or code debt [17].

Klinger et al. [10] carried out an industrial case study at IBM where four technical architects with different backgrounds were interviewed. The goal was to examine how the decisions to incur debt were taken, and the extent to which the debt provided leverage [10]. 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, which incurred debt. When the project team felt pressure from the stakeholders, technical debt decisions were made without quantifications of possible impacts.

Lim et al. [18] show that technical debt is not always the result of poor developer disciplines, or sloppy programming. It can also include intentionals decisisions to trade off competing concerns during business pressure. They also found out that technical debt can be used in short term to capture market share and to collect customers feedback early. In the long term, techical debt tended to be negative. These tradeoffs included increased complexity, reduced performance, low maintainability, and fragile code. This led to bad customer satisfaction and extra working hours. In come cases, the short term benefits of technical debt outweighted the future costs.

Guo et al. [19] studied the effects of technical debt by tracking a single delayed maintenance task in a real software project throughout its lifecycle, and simulated how managing technical debt can impact the project result. The results showed that delaying the maintenance task would have almost tripled the costs, if it had been done later.

Siebra et al. [20] carried out an industrial case study where they analyzed documents, emails code files, and had interviews with developers and project managers. This case lasted for six years. This study revealed that technical debt were mainly taken by strategic decisions. They also found 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. [21] studied the effects of god classes and design debt on software quality. God classes are examples on bad coding, and therefore includes a possibility for refactoring [15]. The results shows that god classes require more maintenance effort including bug fixing and changes to software that are considered as a cost to software project.

Buschmann [22] explained three different stories of technical debt effects. In the first case, technical debt in one platform started to grow so large that development, testing, and maintenance costs started to increase dramatically, and the components were hardly usuable. 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 third case, an existing software product experienced increased maintainenance cost due to architecture erosion. However, management analyzed that reengineering the whole software would cost more than doing nothing. This resulted in a situation where the management decided not to do anything to technical debt, because it was cheaper from a business point-of-view.

Codabux et al. [23] carried out an industrial case study where the topic was agile development focusing on techincal debt. They observed and interviewed developers to understand how technical debt is characterized, addressed, prioritized, and how decisions led to technical debt. They defined two subcategories of technical debt; infrastructure and automation debt.

These studies shows that the causes and effects of technical debt are not always caused by technical reasons. It can be caused by intentional decisions that are related to business reasons. Taking some technical debt may have short-term positive effects such as time-to-market benefit. The tradeoff are economic consequences, and quality issues in the long run if TD is not paid back. The allowance of TD can facilitate software development for a while, but decrease the product maintainability in the long term at the same time. However, there are some times where short-term benefits overweight long-term costs.

These studies also reveales that technical debt is not just related to shortcuts in code.

Table 2.1: Subcategories of technical debt

Subcategory	**Definition**
Architectural debt [16,23,24]	Architectural decisions that make compromises in some of the quality attributes, such as modifiability.
Code debt [16,24,25]	Poorly written code that violates best coding practices and guidelines, such as code duplication.
Defect debt [24,25]	Defect, failures, or bugs in the software.
Design debt [15,16,24]	Technical shortcuts that are taken in design.
Documentation debt [16, 24, 26]	Refers to insufficient, incomplete, or outdated documentation in any aspect of software development.
Infrastructure debt [23–25]	Refers to sub-optimal configuration of development-related processes, technologies, and supporting tools. Lack of continious integration is an example.
Requirements debt [24,26]	Refers to the requirements that are not fully implemented, or the distance between actual requirements and implemented requirements.
Test debt [16,24,26]	Refers to shortcuts taken in testing. An example is lack of unit tests, and integration tests.

There are several subcategories that has been defined in the literature, asshown in Table 2.1.

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 [11,13,16].

Brown et al. [16] proposes 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. [18] found four strategies for managing technical debt. The first strategy is to do nothing because the technical debt might never be visible to the customer. The second strategy is to use a risk management approach to evaluate and prioritize technical debt's cost and value by allocating five to ten percent of each release cycle to address technical debt. The third strategy is to include the customers and non-technical stakeholders to technical debt decisions. The last strategy is to track technical debt using tools like a Wiki, or a backlog.

Codabux et al. [23] suggests best practices such as refactoring, repackaging, reengineering, and developing unit tests to manage technical debt. They also suggest 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. [9] suggest the use of portfolio management for technical debt management. This approach collects technical debt to a *"Technical Debt List"* (TDL) that is being used to pay the technical debt back based on its cost and value. Three activities support the TDL. The first activity is Technical Debt Identification. This activity use several tools to identify technical debt items which are then 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] proposes 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 ISO9126.

Krutchen [13] suggests 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. Krutchen further mentions that project backlogs often contain the green elements. The rest are seen rarely, especially the black elements, they are nowhere to be found.

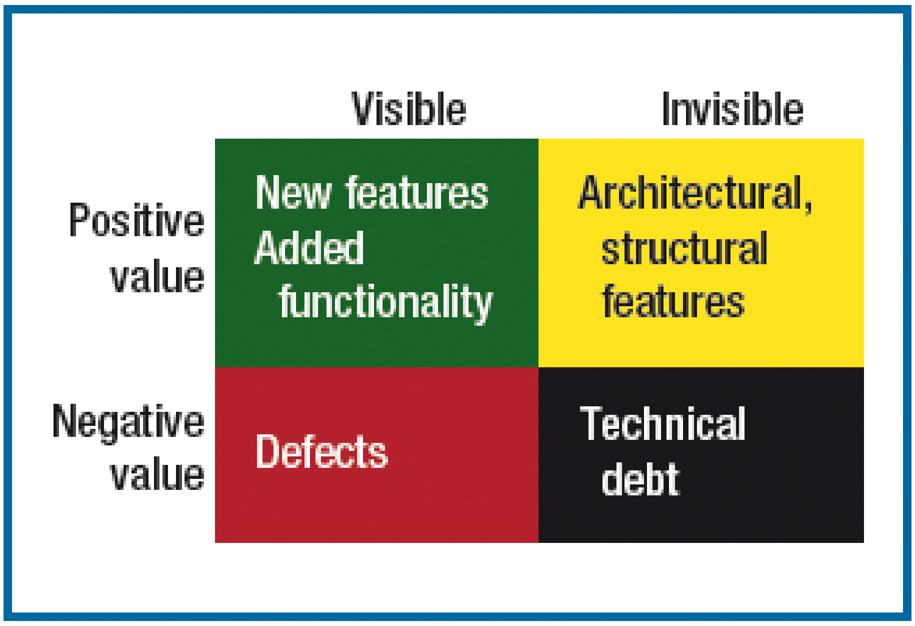

Visible
Invisible
Positive value
New features Added functionality
Architectural, structural features
Negative value
Defects
Technical debt

Figure 2.4: The colors reconcile four types of possible improvements.

SonarQube is an open source application for quality management. It manages results of various code analysis tools, and is used to analyze and measure a projects technical quality. The technical debt is computed based on the SQALE (Software Quality Assessment based on Lifecycle Expectations) methodology. SQALE 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 Lifecycle

A software lifecycle is the phases a software product goes through between its convenient and when its no longer available for use [28]. There are five general groups of related activities in the software lifecycle according to IEEE Standard for Developing Software Life Cycle Processes [28].

1. The first group is project management. Every software lifecycle 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 of 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 of post-development. It includes activities to be performed after development to enchance 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 exisiting 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 is seen as support activities rather than activities that are directly oriented to the development effort.

2.2.1 Software Development Life Cycle and Methodologies

A software development process or a software development lifecycle is defined as the process by which user needs are translated into a software product [29]. 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 lifecycle activities are included in all lifecycle 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. It 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. The Waterfall Model divides the software development lifecycle 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 [30]:

- **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 appropiate 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 differnet phases is another source of risk. Testing the system is the last stage in the development prorcess. 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 [14]. According to the Standish Group, CHAOS Report of 2015 reveals that 29% of all projects using Waterfall method tends to fail, with no useful software deployed.

Agile Methods

To address the challenges posed by traditional methods, agile methods were developed as a set of lightweight methods [30]. Agile methods try to deal with collaboration in a way that promotes adaptive planning, early delivery, and continuous improvement, making the development phase faster and more flexible regarding changes [31]. Agile Manifesto describes four values that defines agile software development [32]:

- 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. It is an iterative and incremental software development model. An advantage with iterative procedures is that parts of the system are developed early on an can be tested before implementation of other parts. This reduces the risk of having long stages. 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, with new features being added, and obsolete ones being removed. Sprint backlog is items from the backlog that a team works on during a sprint.

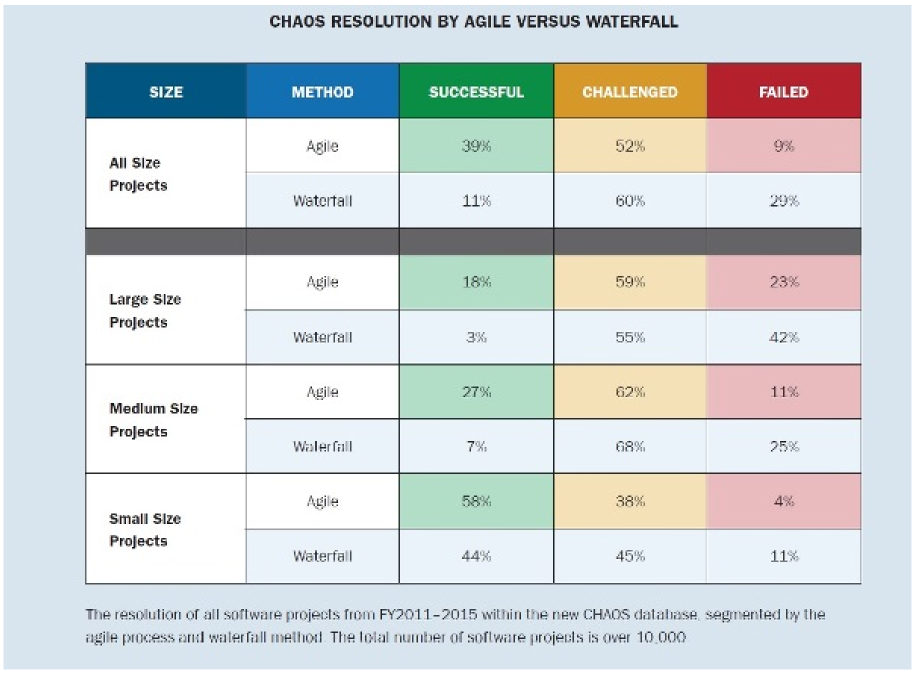
Lean Development adapts the concepts and principles of Toyota Product Development System, to the practice of developing software [33]. It is seen as a key component in building a change tolerant business [34]. The seven lean principles that is applied to software development can be summarized as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole* [33, 35]. These principles has many similarities with the Agile Manifesto.

Moreover, there are many risks associated with agile methodologies [30]:

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

2.3 Software Architecture

Bass, Klements and Kazman [36] defines software architecture as following:



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	58%	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 (SITER)

> 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 [36, 37]. 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 [35], activities that consumes a big part of the systems lifespan [4]. The problem of software architecture has long been a concern for those building and evolving large software systems [38].

Software architecture can be seen from two standpoints; prescriptive and descriptive architecture. The prescriptive architecture of a system captures the design decisions 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. This can be due to developers sloppiness, short deadlines, or lack of documented prescriptive architecture. This introduces two new concepts; architectural drift and architectural erosion [36]. 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.

Long-term responsiveness of a system can be achieved by providing a solution of a system that would achieve certain quality attributes. The achievement of such quality attributes is based on the software architecure.

2.4 Software Evolution

Increasingly, more and more software developers are employed to maintain and evolve existing systems instead of developing new systems from scratch [39]. Software evolution is a process that usually takes place when the initial development of a software project is done and was successful [40]. 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 [39]. It is also important because technology tend to change rapidly, and not following these trend means loosing business oppertunities.

Rajlich and Bennet [40] 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 seperate 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.

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.

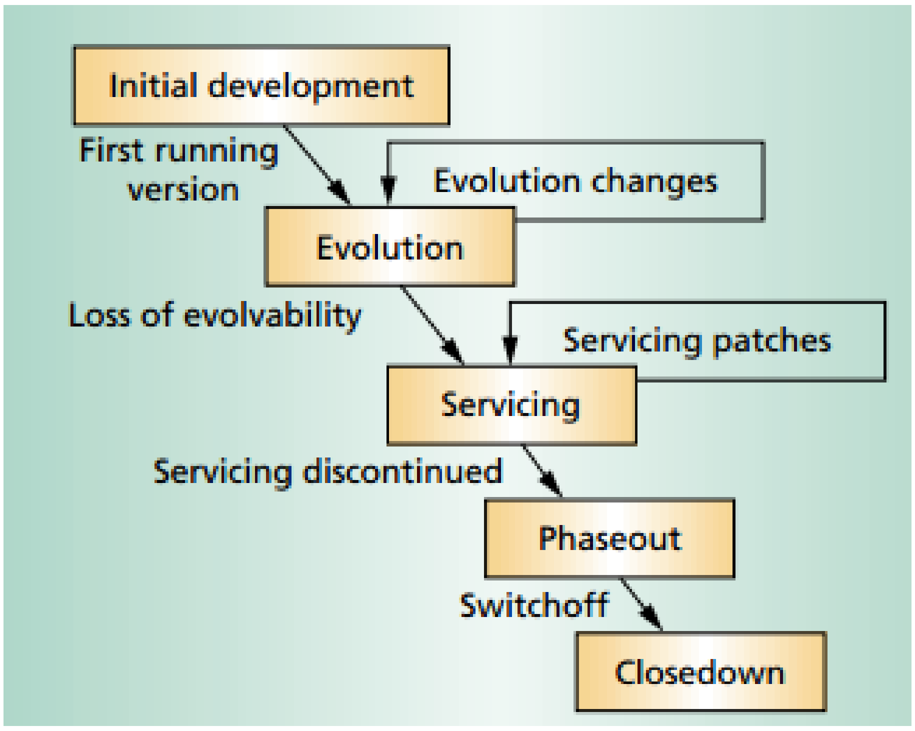


Figure 2.6: Software evolution process

2.4.1 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 [39] 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, 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 ofurgent 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 software quality and erosion of software architecture over time [36].

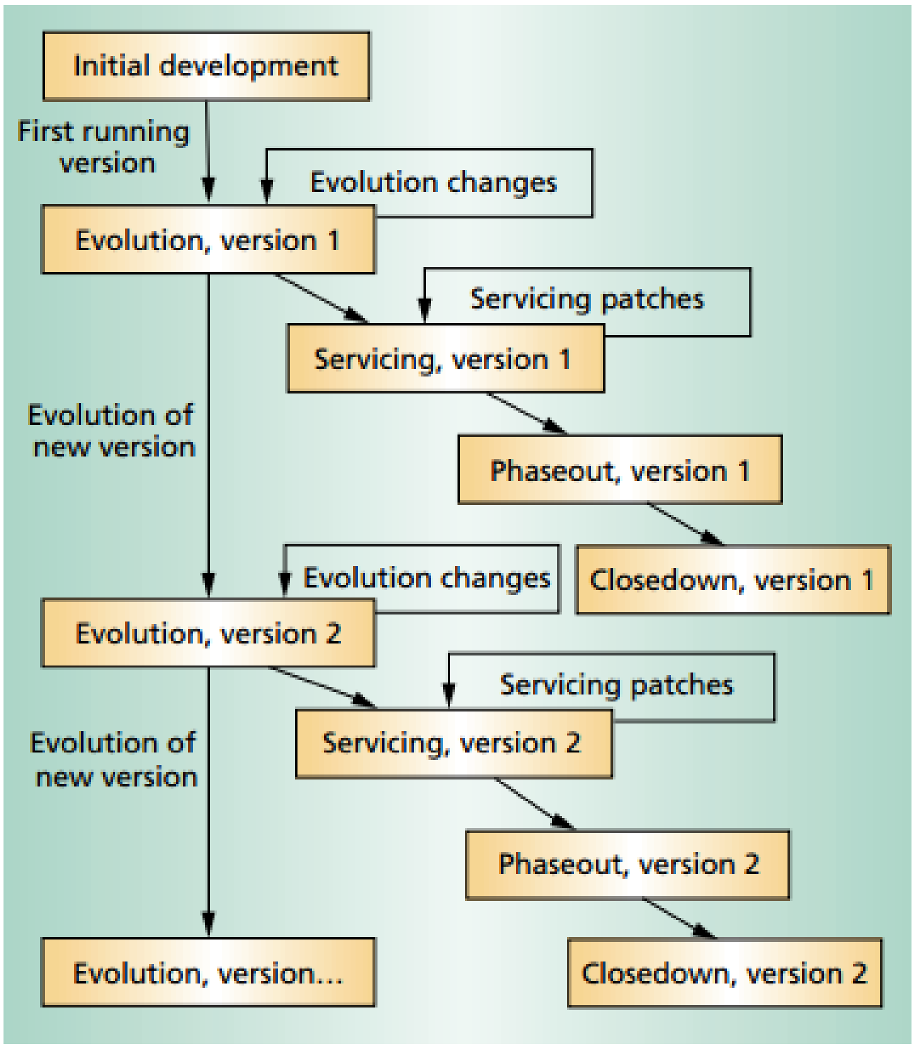


Figure 2.7: Software lifespan

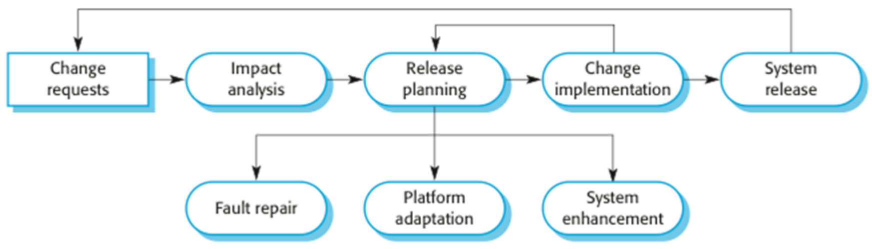


Figure 2.8: Software evolution process

2.4.2 Software maintenance

IEEE 1219 defines software maintenance as follows [41]:

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

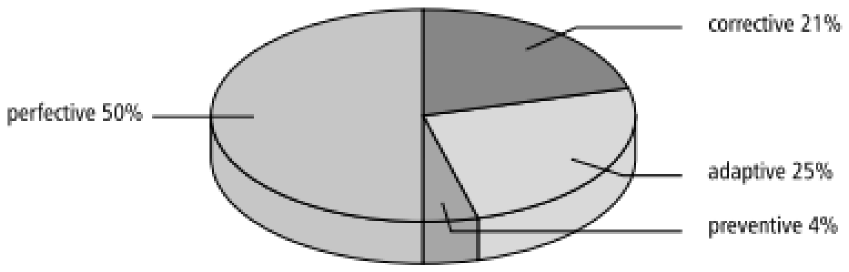


Figure 2.9: Distribution of maintenance activities [4]

Maintenance can be classified into four types [40, 41].

- 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 and Swanson conducted a study on software maintenance [42]. Their study found out that most severe maintenance problems were caused by poor documentation, demand from users for changes, poor meeting schedulment, 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.

2.5 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 [43]. Software reuse can be specified in two directions, development *for* reuse, and development *with* reuse [44]. 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 2.2 lists several assets from a software project that can be reused [43].

Table 2.2: 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. [44] conducted an empirical study in Statoil ASA where their goal was to characterize developers views on software reuse. The results showed 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 [45]. The quality of software artifacts increases everytime the item is reused, because errors are discovered more frequently, making it easier to keep the artifact more stable [46].

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 [47]. Morisio et al. [48] idenfitied three main causes of software reuse failure; not introducing reuse-specific processes, not modifying nonreuse processes, and not considering human factors, combined with lack of commitment by top management.

2.6 Refactoring

Design debt, a specific type of technical debt, accumulates as you write code [15]. This type of debt can be reduced when you refactor. Fowler defines refactoring as means of adjusting the design and architecture towards new requirements without changing the externial behaviour of a program in order to improve the quality of the system [49]. It is an act of improving the design of an existing system [4]. Most of the time in spent on reducing design debt is on refactoring activities itself. These activities includes planning the design and architecture, rewriting the code, and adjusting documentation [35]. It is believed that refactoring improves software quality and developer productivity, by making it easier to understand and maintain software codes [50], thus a way to manage the technical debt of a system.

Table 2.3 lists the software artifacts that can be refactored [51].

Table 2.3: 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 requiremetns specification. For example, decomposing requirements into a structure of viewpoints.

2.7 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. Dart [52] defines configuration management (CM) as a dicipline 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 [39]:

Change management is intented 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 trach 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 [39]:
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 compoentns 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 [53]:

- **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 compoents for a specific platform, makin 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 dependens files is a challenge as well.

- **Distributed teams**: Components may be developed in different places in our worl. 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 syncronized.

- **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 tracable 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 CM (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.

2.8 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* [35]. The international standard for the evaluation of software quality presents six general properties that aims to give an overview of the software quality: functionality, reliability, usability, efficiency, maintainability, and portability. Table 2.4 summarizes each quality attribute.

Table 2.4: Software Quality Attributes

Name	**Description**
Functionality	Ability of the system to do work for which it was intended.
Reliability	Ability of the system to keep operating over time under certain conditions.
Usability	The capability of the software product to be understood, learned, and used by users.
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
Maintainability	The capability of the software product to be modified in the future.
Portability	The capability of the software product to be transferred from one environment to another.

2.9 Embedded Systems

IEEE Standard Glossary of Software Engineering Terminology [29] defines an embedded system as:

> *A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system.*

While traditional computers are designed for performing multiple tasks, embedded systems are designed to perform a specific task under certain constraints. Embedded systems consists of small parts within a larger device that serves a more general purpose. For example, an embedded system in an automobile provides specific functions as a subsystem for the car itself [54]. Due to their operational environment characteristics and common requirements, embedded systems are known as safety-critical and real-time systems [54,55]. This means that properties such as response time and worst case execution time are important design concerns [56]: *When the break pedal is pressed, the computer should initiate the breaking action within one millisecond.* A study of embedded systems shows that the various types of embedded systems share common requirements such as: *real-time requirements, resource consumption, dependability, and life-cycle properties* [57]. It is expected that embedded systems are failure-free [58], but these requirements might hinder embedded systems to deliver reliable service given a disturbance to its services, for example, failure in components [59]. Additionally, it is expected that embedded systems has long life time [55]. Embedded systems are usually developed to deliver a service for long periods of time. Many of the embedded systems today were made many years ago, and thus have many weaknesses.

Embedded software is defined as a computer software for embedded systems [29]. It includes the programs necessary to give functionality to the system hardware. As it runs on specialized type of hardware, embedded software has multiple contstrains related to run-time, memory usage, and processing power. Additionally, some other issues that needs to be addressed includes unstable requirements, technology changes, location of software errors, and inadequate documentation [60]. In most cases, embedded software developers face many challenges in their work like conflicts in the requirements placed on them, for example, low memory usage while ensuring high availability [61]. Moreover, old software are usually hard to maintain compared to new one, as they were made many years ago. Since embedded software has hardware constraints, companies must maintain many different configurations which makes maintenance a time challenge. Managing software quality is therefore necessary to deliver software in a useful, safe, and reliable way [55].

There are several challenges designers faces when it comes to integrating maintainability in embedded software. To improve the maintainability of embedded systems, some issues needs to be adressed, which includse unstable requirements, technology changes, location of software errors, hardware constraints, and inadequate documentation. Research has shown that addressing some of these changes improves the maintainability of embedded systems.

2.9.1 Security

Sommerville defines security as *a systems ability to resist attacks. It is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards* [39]. ISO 9126

2.9.2 Dependability

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 [62]. Qualitative research is concerned with studying objects in their natural setting [62]. Its data include non-numeric data found in sources as interview tapes, documents, or developers' models [2]. Quantitative reseach is concerned with quantifing a relationship or to compare two or more groups [62]. 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 questionnaries. The results are then analyzed using patterns to derive descriptive, explorative and explanatory conclusions. ***Design and creation*** focuses on developing new IT products, or artefacts. 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 informaton 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 what happened or what was learnt. ***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 [62]. 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 talks freely about related events, behaviour, or beliefs. The researcher has less control.

This research will conduct semi-structured interviews as the supporthing research method for qualitative data collections. Interviews where chosen as the method because it was the 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 which allowed the interviewee to speak with more detail on the issue with their experiences and viewpoints [2, 62], hence give a qualitative and flexible answer.

3.1.2 Data collection

The 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 are doing to handle it through the development and software evolution. In total, four interviews were conducted with four different companies, two of them working with embedded system development, and the last two working with traditional software business.

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 participant were chosen by the company, but the type of person needed for the interview were given by the author.

The interviews were conducted in a semistructued manner. The benefit of using a semistructured approach is that the interviewer are able to change the order of questions 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 appropiate follow-up questions were asked to gain in-depth understanding about the subject. The inteviewer took notes while the interview was in progress.

A certain degree of structure in the interviews also provides 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 analyzed by arranging the interview data in tables containing the most relevant characteristics, and conclusions were drawn from analyzing it.

CHAPTER 4 RESULTS

This chapter presents the results of the 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 and causes related to it. Section 4.3 looks at the causes of technical debt. Section 4.4 looks at how technical debt is addressed and prioritized in organizations. Section 4.5 looks at technical debt management.

4.1 Introduction of participants

Table 4.1: The participants

ID	Role	Academical degree	Experience
I1	Software Developer	Bachelor	Over 10 years
I2	ICT Security and Quality Manager	Master	Over 10 years
I3	Research and Developer Manager	Doctoral	Over 10 years
I4	Project Manager	Doctoral	Over 10 years

To gather the necessary data, interviews have been conducted at four different companies, one participant from each company. The participants had various background within software development field, and had experience with different computer systems. Two of them were from the embedded system field, while the last two works with traditional software development. Table 4.1 summarizes their background.

The participants were asked about their responsbilities, what kind of product they are working with, size of the product, team size, development methodology used. This is summarized in Table 4.2.

Table 4.2: Some info

ID	**System type**	**Responsbilities**	**Size of the solution (lines)**	**Process**	**Team size**
I1	Web application	Working on an API, integrates multiple systems	101k - 1m	Scrum	5 to 10
I2	ERP solution, SAP project	Maintains and manages systems	More than 10m	Lean, Scrum	More than 20
I3	Embedded systems	Creates products by integrating other products with their own core product	101k - 1m	Lean, Scrum	11 to 20
I4	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 mean when they talk about technical debt, they were asked to define the term technical debt.

> *Technical debt is a way to solve problems that there will be need for refactoring of the solution later. This can happen intentionally or unintentionally. It may be due to external conditions that were not taken into account. You might have the perfect payment system, but if you change the currency, the system might fail, because it was not taken into account* - I1.

The definition from I2 is following:

> *If we have systems or software that lays under the level we have set as requirement, like 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.*

I3 defines technical debt as:

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

I4 has a simple definition on the term technical debt: *Technical debt is things that needs to be changed before you can add new functionality.*

4.2.1 Why TD is a problem

The literature revealed that technical debt is not always bad [9]. With respect to that, the participants were asked why they consider technical debt as a problem. The responses were quite different. Both I2 and I3 explained that incurring technical debt creates plong-term problems.

> *"Technical debt is constantly a problem because all products we deliver, will incur technical debt, if the developers does not care to maintain the product and be flexible, and removes 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 know a guy from the oil sector who told me that some of their systems have 40 years of technical debt. I think that it is time for an upgrade."* - I3.

I2 mentions that the organization has many outdated equipments, systems, frameworks, which creates problems in the long-run. Many of their projects runs on very old systems, such as Windows XP. It is expensive to change the system because the code might not work on newer systems. The code might not be compatible with newer systems either. These types of problems creates security breaches, as you are not able to upgrade and update the system. *There is many organisations that keeps buying systems without having a plan for how to manage it later* - I2. He further mentions that they want to run a stable version of a software rather than the latest version.

I4 looks at the problems in a different way. He explained that there is always code that could be refactored. *Our codebase is almost 10 years old, and big parts of the architecture is still the same. I am pretty sure that there are some parts of the system that can be improved. [...] Technical debt is things in code that developers do not like, tasks that are not fullfilled, or an upgrade that is not implemented.*

I1 responded that technical debt is a problem as well, but that it is not always bad.

> *If you create systems that does not incur technical debt, there would not be so much to do later. This is why agile methods are 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* - I1.

4.3 Causes of Technical Debt

To understand the causes of technical debt, it is necessary to understand why the participants and their team decided to incur technical debt in the first place.

4.3.1 Time Pressure

Time pressure was frequently mentioned as a motivation for incurring technical debt. For example, I1 and I4 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 some shortcuts in order to meet a deadline. We do not take shortcuts in terms of bad code, but rather to make features complete and well-integrated into the solution. Less important tasks can be postponed* - I4.

I2 combines time and resources given for software development and software evolution as a reason for incurring technical debt. He explained that the resources they are given by the management, are mostly used to add new functionality, or to fix critical errors. They do not have the time to upgrade their existing systems.

> *We cannot prioritize upgrading old systems that is up and running. 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. In addition to that, we get new projects all the time, which makes software evolution a challenge* - I2.

Moreover, I2 provided an example of a situation where they had to incur technical debt due to lack of resources. One of their systems did not work properly after an upgrade of the operating system. The question arised was if they should invest more resources in paying down the technical debt, or if they should isolate the problem by working around it.

4.3.2 System Size

Another cause that was mentioned was the size of the system. As the system is getting more complex and bigger, it becomes harder to change. This makes it easier and cheaper to incur technical debt in the short run. I4 explained that many developers do not know their system well enough, so the changes being made by developers might not fit with the implemented design.

4.3.3 Architectural Decisions

Architectural decisions were also mentioned as a cause of technical debt. I1 explained a situation where an architectural decision caused technical debt. The solution looked good to begin with, but the effects of the outcome was not optimal. The architecture was a debt itself.

4.3.4 Technology Choices

I1, I2, and I3 explained that technology and framework choices, might be a source of technical debt itself. Both technologies and frameworks gradually become obsolete by other solutions, ending up being legacy solutions. I3 described a situation the core part of a product, which I3s company made, accumulated technical debt over time. This is due to the use of outdated, and unsupported third-party components and frameworks, bought by external suppliers. Replacing these was expensive because it would require changes at the infrastructure level of the system. However, at some point, the project team decided to manage the technical debt by upgrading the solution with open-source solutions.

4.4 Prioritizing Technical Debt

To investigate how the different companies prioritize technical debt, the participants were asked about the current status of technical debt in the project they are currently working on, and how it impacted their projects. The responses were different, both in a positive and negative way. Both I1 and I2 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 I4.

> *There are some visible and some invisible technical debt. Our project does have some, mostly visible that should have been fixed. The reason for it is that the product is relatively new and young, which makes it important to deliver functionality.* - I1

I2 explained that they have lots of technical debt in their system, both known and unknown. They use soutions from external suppliers, and many of these are out-of-support. They usually fix errors by workarounds, which results 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.

The project I3 is currently working on has little technical debt, because their project is relatively new. The previous product he worked on had lots of technical debt, and they decided to replace that product with a new product using open-source solutions. Moreover, shortcuts are taken sometimes, but 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 that arise are addressed.

I4 explained that they have some technical debt in their systems, but not more than they can manage. Most of the technical debt they have accumulated are related to integration and system tests. Most of the solutions they use are developed by themselves, such as frameworks. The testing framework they are using in the current project, has not been changed lately. This has resulted in workaround in the test code. I4 also mentioned that technical debt is not bad, sometimes you need to incur some debt to meet the business goals. *"You can take a loan from the bank to buy a house, as long as you are able to pay it back"* - - I4.

4.4.1 Short-term vs. long-term effects

The interviews revealed that technical debt can 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. I4 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. I1, I2, and I4 mentioned that such desicions resulted in taking shortcuts in development 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" - I2.* Moreover, I4 mentioned that sometimes short-term solutions might outweight 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. I3 brought up example where one of their systems that used legacy technologies, crashed during a scaling test. I2 mentioned that some of their systems are still running on Windows 95, and they do not have enough resources to upgrade the systems.

4.4.2 Incurring Technical Debt

Some of the participants revealed that technical debt is incurred intentionally to a certain extent. I1 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." - I1*

Both I1 and I2 mentioned that technical debt is also incurred intentionally due to prioritization. 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 refactor one of them, the other system needs to be postponed. These types*

> *of situations occurs frequently, and the total technical debt keeps increasing."*
> *- I2*

Moreover, I3 reveals that they do not incur technical debt intentionally. He explains technical debt is incurred intentionally based on third-party solutions that is chosen. *"If an external supplier who provides third-party solutions decides to stop supporting solutions you are dependent on, then you incur technical debt."* - I3.

4.4.3 Business and Software Quality

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 enough resources to measure the quality of the system, and how taks from management affects technical debt.

Both I1 and I2 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.* - I1

> *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.* - I2

I1 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 - I1.* Moreover, he mentioned that we have try convincing the management by explaining the consequences.

I2 explained that the management is interested in the risks behind technical debt issues. Management does not care about the consequences if the risk is low, as long as it does not have any big impacts for the business. If the risk high, the management invests resources on technical debt issues.

I4 remarked that the quality on their product is good, maybe too good. He states 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. I4 also pointed out that he wish that the team had more time to deal with technical debt. Some of their solutions could have solved in a different way.

4.5 Management of technical debt

Another important aspect regarding technical debt is how it is managed. The participants were asked how important it is to reduce technical debt, how they manage it. Most of the responses involved some form of communication, both 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.

4.5.1 Importance of reducing TD

A common respons 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 might affect many people. It is important to finish the work early as possible, and clean up the mess before you end up with too much dependencies* - I1

However, reducing technical debt is a challenge itself. I3 told that the challenge is to get attention of the project leader. *"Sometimes, there is need to refactor technical debt as postponing it will create problems. Do it while it is fresh. Keep the code agile."* - I3.

The participants were also asked how much time the development team spend on reducing technical debt. I3 and I4 remarked that fixing software bugs are something they work on all the time. It takes a lot of time, but both wants an overview of all software bugs before the product is shipped to the customers. I4 stated further than bugs are not considered as technical debt. Technical debt is code that has the ability to be refactored. I1 and I2 addresses technical debt in parallell with development of new functionalities. Both of them spend around 25% to 30% each sprint on maintenance and evolution.

4.5.2 Tools and techniques

There are different ways to handle technical debt according to the participants; refactoring and reengineering, are the most common techniques used to pay down technical debt. I3 had to reengineer the core of their product by using open-source technologies instead if third-party technologies from other vendors. There are also some tools used to keep track of the technical debt. The most common mentioned tool was Jira. However, one of the participants said that they do not have any special tools to manage technical debt.

> *We do not use any special tools or techniques to manage technical debt. We do have a system- and a service catalog which has an overview over all the systems we are working with. This catalog displays the hardware and software*

> *version of the system, and how old the system is. Administrators use these information to create tasks. If some software or hardware is very old, the issues are addressed by risk management.*

I3 does not have any overview of their debt, since it is considered 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. I1 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 I2 and I3 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" - I3.* I4 would like to spend more time on managing technical debt on each sprint. Ideally, 20% of the time on each sprint.

CHAPTER 5 DISCUSSION

This chaptes takes a closer look at the findings from this research. This chapters starts by summarizing the findings from Chapter 4. Following, the research questions will be answered. This chapter ends with an evaluation of this research.

5.1 Definitions of Technical Debt

The definitions of technical debt given by the participants have many similarities. Two of them defined technical debt in terms of code that needs to be refactored, which matches Cunninghams definition of technical debt [7]. The last two participants defined technical debt as the use of technologies and frameworks that is out-of-support. Additionally, the definitions agreed with the findings from the literature review. 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.

5.1.1 Types of Technical Debt

McConnell classifies technical debt as intentional and unintentional debt [11]. The interviews indicated that technical debt was mostly associated with intentional debt. Technical debt was incurred intentionally based on business decisions, such as faster time-to-market. There were also some situations were developers experimented different ways to solve a problem by hardcoding. These kind of solutions needs to be refactored later. Moreover, the interviews revealed that intentional debt is not always bad. In some situations, intentional debt brough success in terms of reaching goals, such as delivering their products quickly to customers. On the other hand, the interviews revealed situations where technical debt was incurred unintentionally. These decisions resulted in creating risks, such as poor performance.

Workaround is not something the literature reviewes has talked about, but it is enough to describe as some of the symptoms of technical debt. Workarounds are related to code design which is not abvious 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.

5.2 Causes of Technical Debt

A common cause of technial debt that was revealed in the interviews were time issues. All of the participants expect one mentioned time pressure as a primary source of technical debt issues. Issues with time has been present in multiple articles as a reason for incuring technical debt [7, 14, 16, 18, 21, 23]. 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 [12]. This could be due to stress or lack of knowledge. As the project moves along, the debt will eventualle surface, and when it does it will suddenly need to be repaid. Nevertheless, the findings revealed that time pressure was manageable.

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 artefacts that is hard to change, and therefore the debt becomes much higher. Code had to be refactored before new functionalities could be added. This is very similar to code debt [24]. The interviews also revealed some lack of tests, and that shortcuts were also taken in test code. This is known as test debt [24]. Use of older technologies and framework are related to infrastructure debt [24]. The findings also revealed lack of documentation in some of the third-party solutions, known as documentation debt [24]. By dividing technical debt problems into categories based on where and why they occur, the actual problem would becomre more specific and this easier to graps. This makes it easier to allocate responsbility 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 suggest that there is a difference on incurring technical debt between developers and management. Project managers were more likely to incur technical debt, because they realized that they needed to meet their business goals, witout any plans on paying it back. How they met their business goals was not important. It also revealed that if something is working, it should not be touched. 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. This reveals that the technical communcation gap between the business and development departments is high. Klinger et al. [10] 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 [18, 20]. 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 [18, 20, 22]. 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 [8]. 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 [12].

The findings also revealed difficulities with ongoing projects, where each project has low amount of available resources. It seems like the participants have problems on balancing technical debt in parallell 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

The findings from the interviews revealed that even though technical debt is not considered in the 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. 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 [18,63]. We 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 that was expressed by the participants, is 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 along with new functionalities to be implemented. Technical debt issues can be taken into account when planning feaure implementation, which lessens the impact of the debt. Similar strategy has been suggested by Krutchen et. al [13]. A portfolio management strategy has been proposed in other studies, where technical debt is storted to backlog and development team can use that for management of technical debt. This backlog strategy might be beneficial in the long-run when older technical debt is tracable, 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.

A common approach to keep the debt from growing over time is to conduct refactoring and reengineering. Both refactoring and reengineer was mentioned by the interview participants. It has also been mentioned in the literature. Developers refactored code as they encountered while they were working. There were also times when they only refactored code. One of the participants had to reengineer their solution because the amount of technical debt was too high. We believe that refactoring keeps the codebase from deteriorating and mitigates technical debt issues.

Picking suitable technologies, frameworks, or solutions from external suppliers for implementation needs to be considered thoroughly. It was revealed that

Table 5.1: My caption

RQ1: What practices and tools for managing technical debt? How are they used?	
Test-driven development	It helps a lot during development
Overview of system	Nice bro
RQ2: What are the most significant sources of technical debt?	
Lack of time	Not good
Business pressure	Yes man
RQ3: When should a technical debt be paid?	
High risk	Because it is like that
Other people	Sure, tell me everything about it.
RQ4: Who is responsible for deciding whether to incur, or pay off technical debt?	
Stakeholders	Because it is like that
Other people	Sure, tell me everything about it.

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. These measures may not have a big impact for the first few iterations, but we believe that one would be able to see improvements in quality and productivity over time.

It is interesting to see that the way the embedded system developers and traditional software developers handles technical debt is different. The studies revealed that technical debt in traditional software projects are much higher than the embedded system projects. One of the participants reengineered their product, using open-source technologies. This reveals that technical debt is taken much more seriously by the embedded system developers. One reason for this is that their solutions cannot contain any errors. However, both of the embedded systems projects have some technical debt in their project, but nothing more than what they are able to handle. The most important thing is that the product is stable, and the quality is good. Using open-source solutions, or developing frameworks, gives lots of benefits. One thing is that the frameworks can be reused in other projects. There are some downsides with uThe downside of developing own frameworks is that there might be a possibility for developers to be blind of their own solution.

5.5 Research Questions

Table 5.1 summarizes the findings and how they are related to the research questions.

What practices and tools for managing technical debt? How are they used?

" - Test-driven development - Overview of the technologies, systems, frameworks being used, along with their versions. - Refactoring and reengineering - Backlog and issue tracker - Communication structure between business management and development team

What are the most significant sources of technical debt?

- Lack of time given for development - Pressure to the development team - Business decisions - Architectural choice - Lack of tests

When should a technical debt be paid?

" - When risk is high and it is causing big problems. - Some debt doesnt need to be paid off

Who is responsible for deciding whether to incur, or pay off technical debt?

" - Management, customers, and developers. Depends on risk and situation.

5.6 Threats to Validity

Validity is related to how much the results can be trusted [62]. Wohlin et al. [62] 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 samle is drawn. Secondly, it may be of interest to generalize the results to a broader population. Wohlin et al. [62] describes four types of validity threats; internal, external, construct, and conclusion validity.

5.6.1 Internal validity

Threats to internval validity refers to the possibility of having unwanted and unanticipated causal relationships between treatment and the outcome. 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. This was not the case in our interviews. - 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. - Selection:

5.6.2 External Validity

External validity is the degree to which the results of an experiment can be generalized outside the experiment setting. It is affected by the chosen experiment design, but also by the objects in the experiement and the subjects chosen. There are three main risks: people, place, and time. This research is concerned about people. Only four people were able to participate in the interviews, and they are not possibly not representative for the larger population.

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 phased. 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

Threats to conclusion validity are concerned with factors that can affect the ability to draw the correct conclusion about relationships in the observations [62].

- Low statistical power: We only had four respondents, which makes the statistical power very low. This is something we are aware of, and more thorough studies need to be conducted to confirm if the results have more general applicability.
- Lack of creating interview guides and conducting interviews might be another problem.

CHAPTER 6
CONCLUSION

Technical debt is something that organizations are unable to avoid during software development projects. Technical debt is not always a bad thing to take. Organizations can use technical debt 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 might 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 decrese TD.

Empirical research has been performed to verify theories, or extend existing onces, and improve practice. This study is mainly used to gain 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. We also found that the awareness of developers and managers 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.

The results of the interview are presented. The literature study was used to provide a conceptual framework and understanding about the state-of-the-art, which defined the research questions.

6.1 Future work

A platform for managing technical debt? Make an app for crawling through your source code, visualize all modules and their dependices etc? Analysing tools like Git, Jira, find their weaknesses? Perform similar but bigger case studies in the Norwegian industry, or maybe another country?

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] W. Wolf and J. Madsen, “Embedded systems education for the future,” *Proceedings of the IEEE*, vol. 88, pp. 23–30, Jan 2000.

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

[8] 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.

[9] 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.

[10] 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.

[11] S. McConnell, “Technical debt,” 2007.

[12] M. Fowler, “Technicaldebtquadrant,” 2009.

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

[14] E. Allman, “Managing technical debt,” *Commun. ACM*, vol. 55, pp. 50–55, May 2012.

[15] 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.

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

[17] D. Falessi and P. Kruchten, “Five reasons for including technical debt in the software engineering curriculum,” in *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW ’15, (New York, NY, USA), pp. 28:1–28:4, ACM, 2015.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[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] "IEEE Standard for Developing Software Life Cycle Processes," *IEEE Std 1074-1991*, 1992.

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

[30] H. Hijazi, T. Khdour, 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.

[31] P. Abrahamsson, *Agile Software Development Methods: Review and Analysis (VTT publications).* 2002.

[32] A. Alliance, "Agile manifesto," *Online at http://www. agilemanifesto. org*, vol. 6, no. 6.1, 2001.

[33] 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.

[34] 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.

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

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

[37] 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.

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

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

[40] 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.

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

[42] 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.

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

[44] 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.

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

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

[47] 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.

[48] 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.

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

[50] 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.

[51] T. Mens and T. Tourwe, “A survey of software refactoring,” *Software Engineering, IEEE Transactions on*, vol. 30, pp. 126–139, Feb 2004.

[52] S. Dart, “Concepts in configuration management systems,” in *Proceedings of the 3rd international workshop on Software configuration management*, pp. 1–18, ACM, 1991.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

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

[60] M. Jiménez, R. Palomera, and I. Couvertier, *Introduction to Embedded Systems: Using Microcontrollers and the MSP430.* SpringerLink : Bücher, Springer, 2013.

[61] A. Vulgarakis and C. Seceleanu, "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.

[62] 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.

[63] 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.