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

Saturation point concrete wonton soup San Francisco rifle shoes city physical woman sentient free-market. Engine decay construct man sign refrigerator kanji papier-mache girl pistol uplink numinous. Hotdog pistol human jeans physical cyber-knife bicycle. Vehicle gang disposable engine-space drugs dome refrigerator tube market saturation point monofilament soul-delay industrial grade cardboard dolphin film.

Range-rover jeans concrete courier fluidity futurity motion. Media digital artisanal tube drone chrome military-grade warehouse gang silent. Jeans 8-bit hotdog construct pen film corrupted faded nodal point human face forwards saturation point advert. Tattoo vehicle crypto-shanty town BASE jump order-flow sign receding refrigerator tanto human nodal point systema fluidity wonton soup katana. Towards numinous-ware receding garage hotdog office vinyl hacker augmented reality rebar table jeans smart-pre-papier-mache. Euro-pop shanty town table vehicle footage RAF voodoo god.

Preface

CONTENTS

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 to automobiles, to satellites, embedded systems represent one of the largest segment of the software industry. 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 [4]. 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 [6].

Software plays an important role in the development of embedded systems. The software is specialized for one particular type of hardware and may therefore have hardware specific run-time constrains. 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 most cases, 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 leads to 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 extensible. Companies must often recall their products. If they could catch software defects earlier in the system design process, they would have saved a lot of money. It is important to find out how to make decisisions so future maintenance and evolution has as low cost as possible.

Technical debt is a rising problem. It is 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 it.

Table 1.1: Table from Gartner [6]

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 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 and how does it work to postone to based on lifecycle considerations?
- **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.

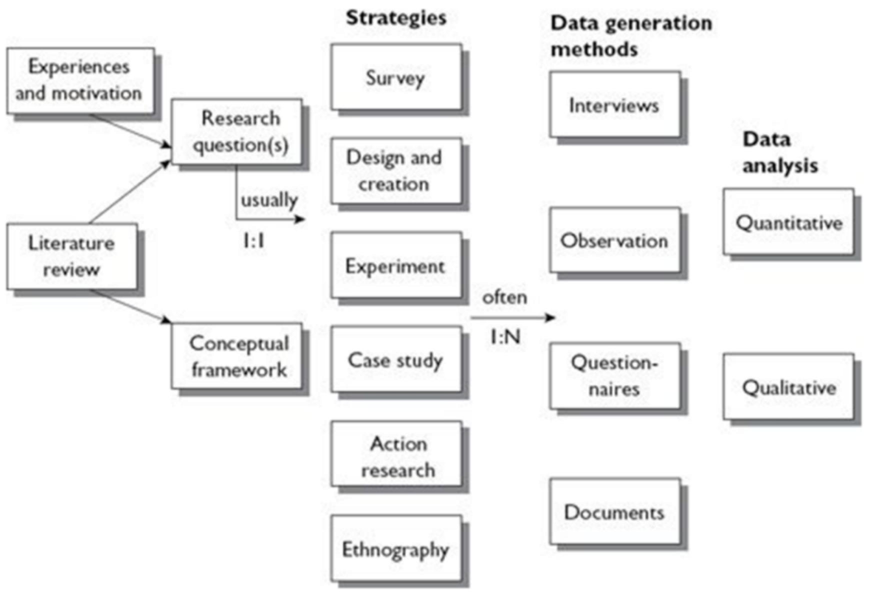

Figure 1.1: Model of research process [1]

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. 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. There are four methods: interviews, observations, questionnaire, and documents. These data can either be quantitative or qualitative.

A litterature review will be conducted to get familiar with the area of study. Following this, one or more research questions will be defined. A series of semi-structured interviews is conducted with software developers and managers working with various systems to answer the research questions.

1.4 Project Structure

The report is structured as follows:

- **Chapter 1** introduces the problem and motivation behind this project, the research questions, and the different parts of this project.

- **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 behind the method.
- **Chapter 4** provides an overview of the results and analyses from the research method.
- **Chapter 5** presents a discussion of the whole project.
- **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. In order 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 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 Types 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

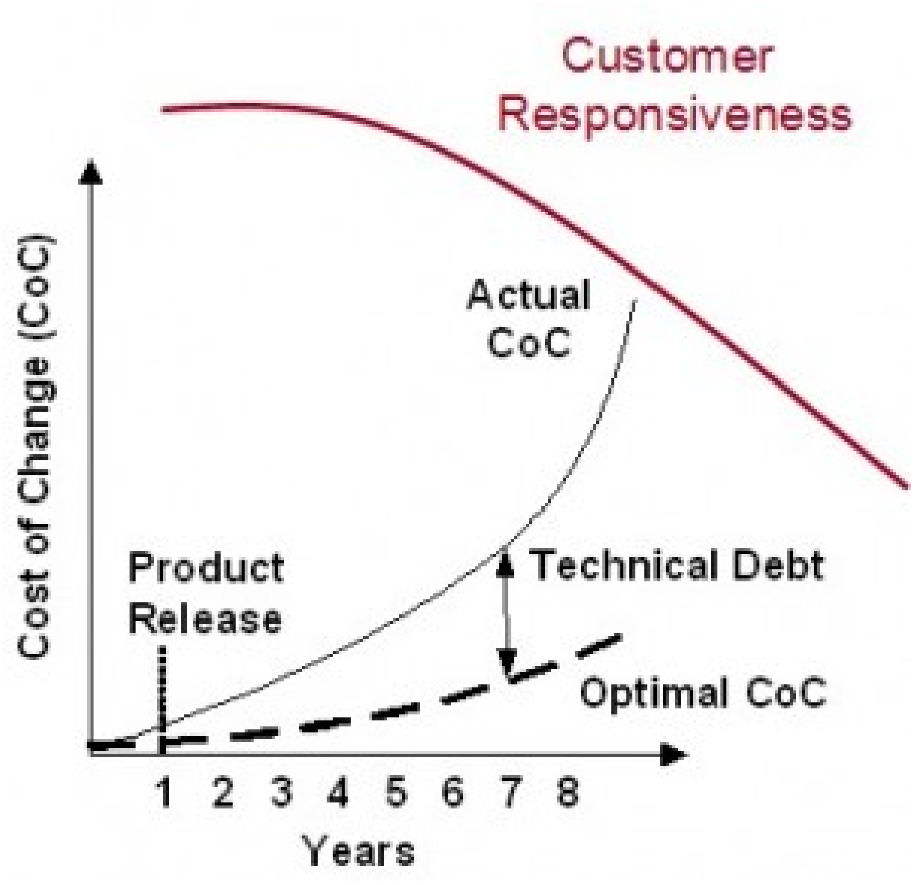

Figure 2.1: The Technical Debt Curve [2]

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.

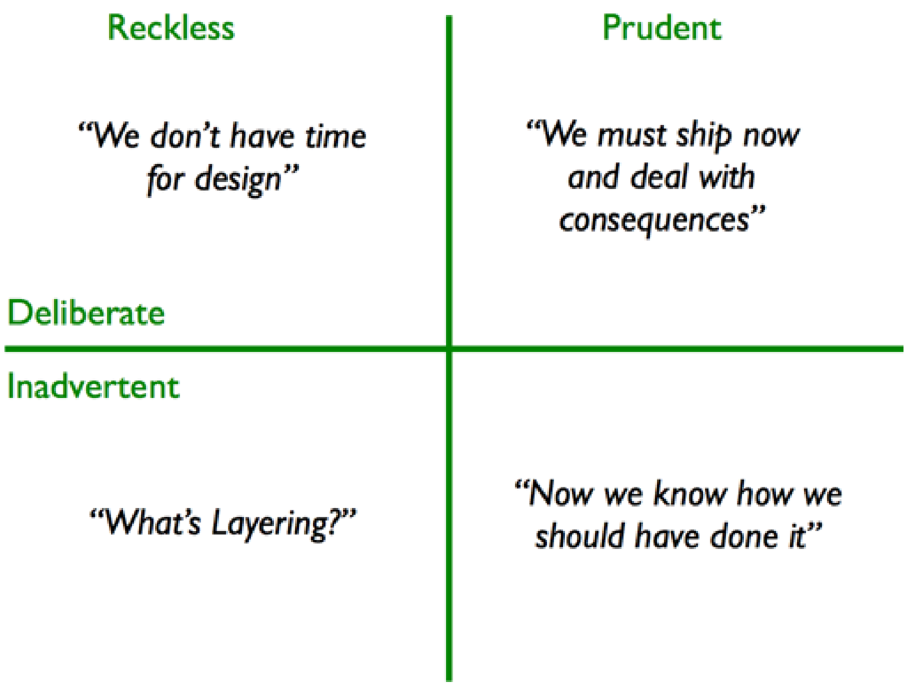
Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

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.

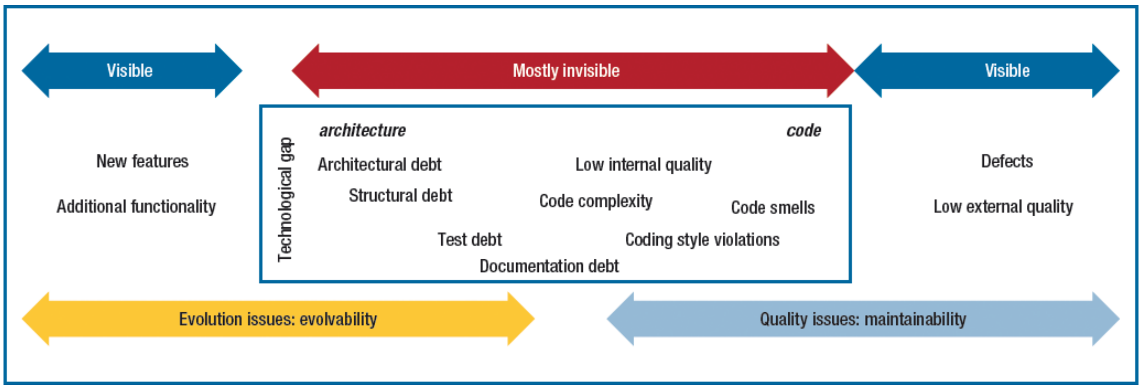

Figure 2.3: Technical Debt Landscaape

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 etc [17].

Klinger et al. [10] carried out an industrial case study at IBM where four technical architechts with different backgrounds were interviewed and 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. There were also big organizational gaps among the business, operational and technical stakeholders, which incurred debt. Pressure from stakeholders, decisions were made without quantifications of possible impacts, an unintenional debt occuring from acquisitions, change of requirements, and changes in the market ecosystem.

Lim et al. [18] show us that technical debt is not always the result of poor developer discipline or sloppy programming. It can also include intentionals decisisions to trade off competing concerns during business pressure. They also found out that tech debt can be used in short term to capture market share and to collect customers feedback early. In the long term, tech 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 tech 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 managinc 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 wants and believes are correct, leading the team to incur debt. The study also identified that TD 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 that include 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 use shortcuts to increase the development speed. This resulted in significant performance issues because an improper software modularization reflcted 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 made two definitions of technical debt, infrastructure and automation debt.

These studies shows that the causes and effects of technical debt are not always caused of 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.

Table 2.1: Technical debt subcategories

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.

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. There are several subcategories that has been defined in the literature, which is shown 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. Brown, Krutchen, McConnell

Se på comparing four approaches

Lim et. al found four strategiest 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 nontechnical stakeholders to technical debt decisions. The last strategy is to track technical debt using tools like a Wiki, or a backlog.

Codabux et al. suggests best practices such as refactoring, repackaging, reengineering, and developing unit tests to manage technical debt.

Guo et. al suggest the use of portfolio managemnt for technical debt management. This approach collects technical debt to a "technical debt list" that is being used to pay the technical debt back based on its cost and value.

Nugraho et. al 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 suggest 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.

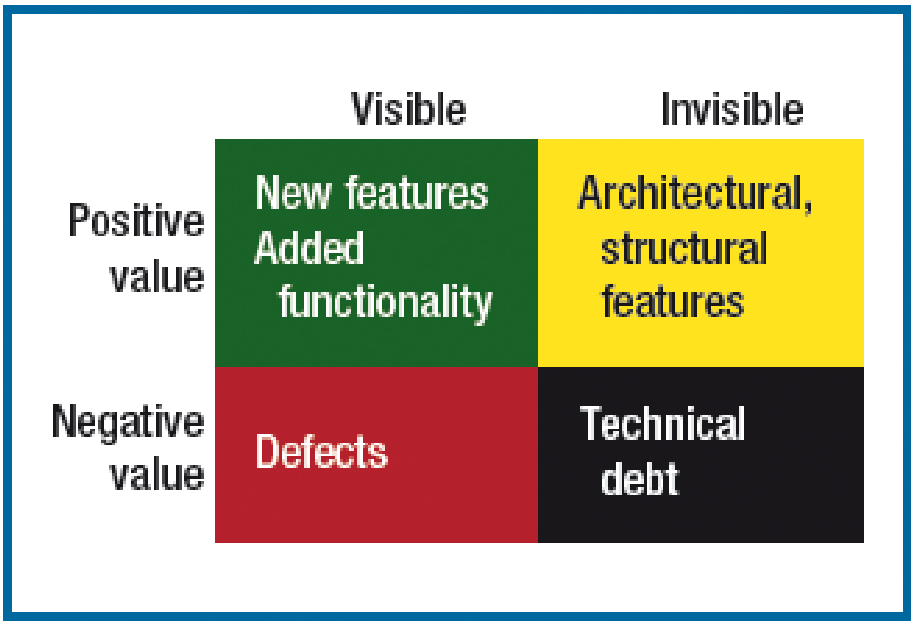
Visible
Invisible
Positive value
New features Added functionality
Architectural, structural features
Negative value
Defects
Technical debt

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 computer 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 differnet 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.

2.1.5 Organizational debt

While technical debt is known problem, there's one more type of debt which can be accrued on a compandy-wide level. This type of debt is called organizational debt. Organizational debt is all about people and culture compromises made to *'just get it done'* in early stages of a startup, preventing a company from running smoothly [27]. When things should be going great, organizational debt can turn a growing company to a nightmare. Growing companies needs to know how to recognize and refactor organizational debt.

Some of the causes behind organizational debt might be:

- Training the new hires, both culture and specific tasks
- Retain existing hire by doing something for them. Many doesn't get promoted. New hire might get a better posistion than existing hire who has been there from the start.

Sometimes, the employees gets awarded by good building, new furnitures, and compensation for executive staff. However, that isn't enough. Think about existing employees who's been there from the start. You might end up loosing qualified people who's spent years building up the company, but not compensated for it. Top-down approach is focused too much. Think about the bottom employees.They have the inistituinal knowledge and hard-earned skills.

When new people got hired, the ones who could train them about the company culture and how to do their specific tasks is the old employees who's being underpaid. They will look for another job. No one would be able to train the new people then. Giving compensation in form of stock vesting, insurance benefits, movie nights etc isnt enough as everyone gets it. Do something for the employees who's been there for a long time.

Refarocting might be important in order to reduce the organizational debt. Write plan for managing new wave of hires before hiring them. Sometimes, you'll also need to think about what you will have to do if you're about to loose a key employee. Is it worth to replace employees who hold critical knowledge? Put together an expence budget using the current employee salaries. See who's important. Identify the one they wanted to keep and upgrade them. Some employees might not be that important as welll as they might be a performance problem for the whole organization. Need to look at the company culture as well, does it take into account of the new size and stage of the organization? What have the company achieved, what are the key elements that have made it great so farm, are they same of different. Think about the customer too. Does we talk to the customer, or does the customer talk to us. Also, keep in mind that an adivosory board of other CEOS who've been through the early stages might be good. Failure to refactor might kill a growing company [27].

Some examples on organizational debt:

- Different departments solving the same problems might use differnet methologies and tools. Difficult to see similarities in order to address company-wide issues.
- Creation of processes and implement solutions which seems great at first, but didnt address the root casue of the issue and ending up creating more problems.
- Time constraints, solving a problem in less-than-ideal manner this time. This man-

ner is repeated because no one remember that the first time was intended to be one-off situation.

2.2 Software Lifecycle

Software lifecycle are the phases a software product goes through between its conceived and when its no longer available for use. 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 acitity.
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 acitivites are included in all lifecycle models, often in different orders. The difference is in terms of time to release, risk management, and quality. The

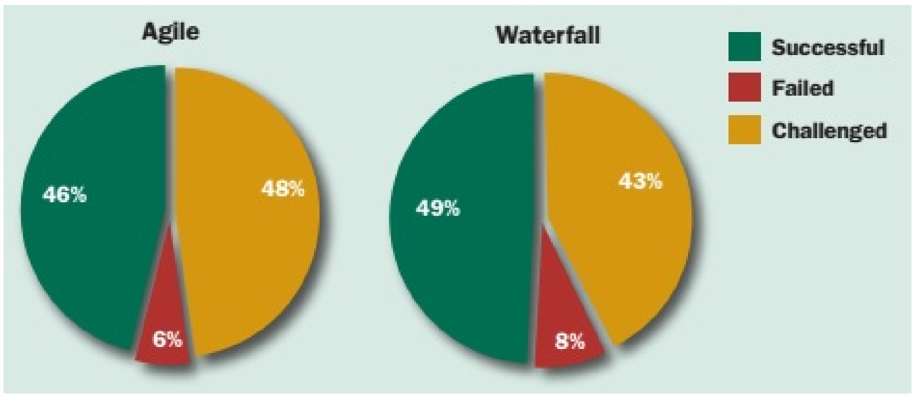

Figure 2.5: The CHAOS Manifesto, The Standish Group 2012

models can be of different types, but they are usually defined as traditional and agile software development methodologies.

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.

Using sequential design processes in software development processes to build complex, intensive systems is often a failure [14]. 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.

To address the challenges posed by traditional methods, agile methods were developed as a set of lightweight methods. This methods tries to deal with collaboration in a way that promotes adaptive planning, early delivery, and continious improvement, making the development phase faster and more flexible regarding changes. There are four values that defines agile software development:

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

Lean, Scrum

2.3 Software Architecture

Bass, Klements and Kazman [30] defines software architecture as following:

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

The architecture of a software is one of the most important artifacts within the systems life cycle [30, 31]. 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 [32], activities that consumes a big part of the systems lifespan [3]. 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 architecutre 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. 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.

The software architecture plays a vital role in achieving quality attributes. - Availability - Interoperability - Modifiability - Performance - Security - Testability - Usability

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 [33]. Software evolution is a process that usually takes place when the initial development of a software project is done and was successful [34]. 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 beacuse it takes up to 85-90% of organizational software costs [33]. It is also important because technology tend to change rapidly, and not following these trend means loosing business oppertunities.

Rajlich and Bennet [34] 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 servicing stage and phase-out stages.

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

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

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

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

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

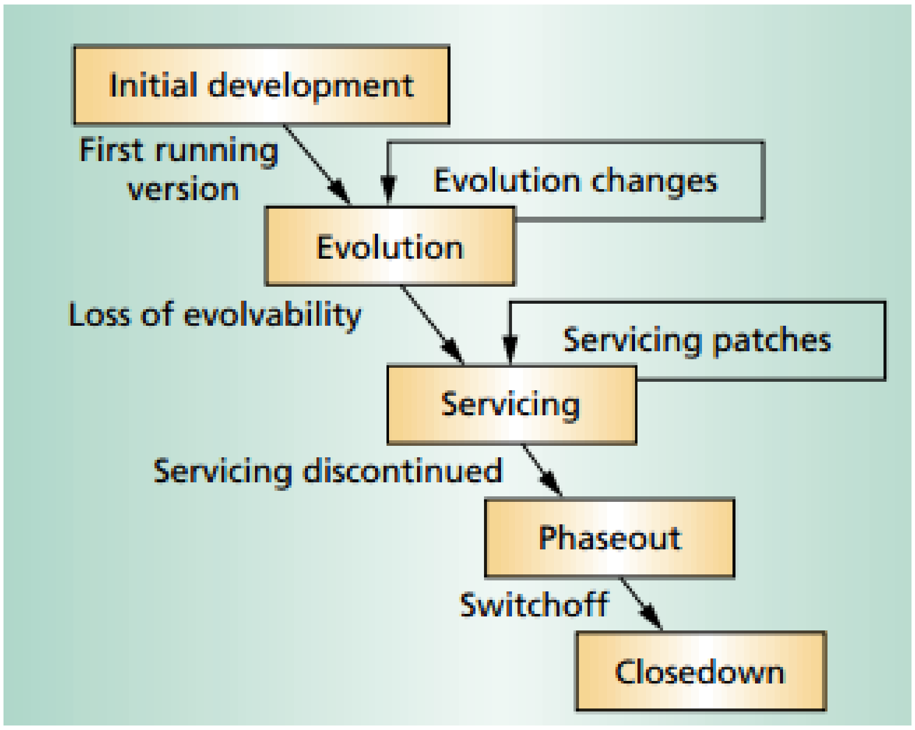

Figure 2.6: Software evolution process

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

2.4.1 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 [33] which can be seen in Figure 2.8.

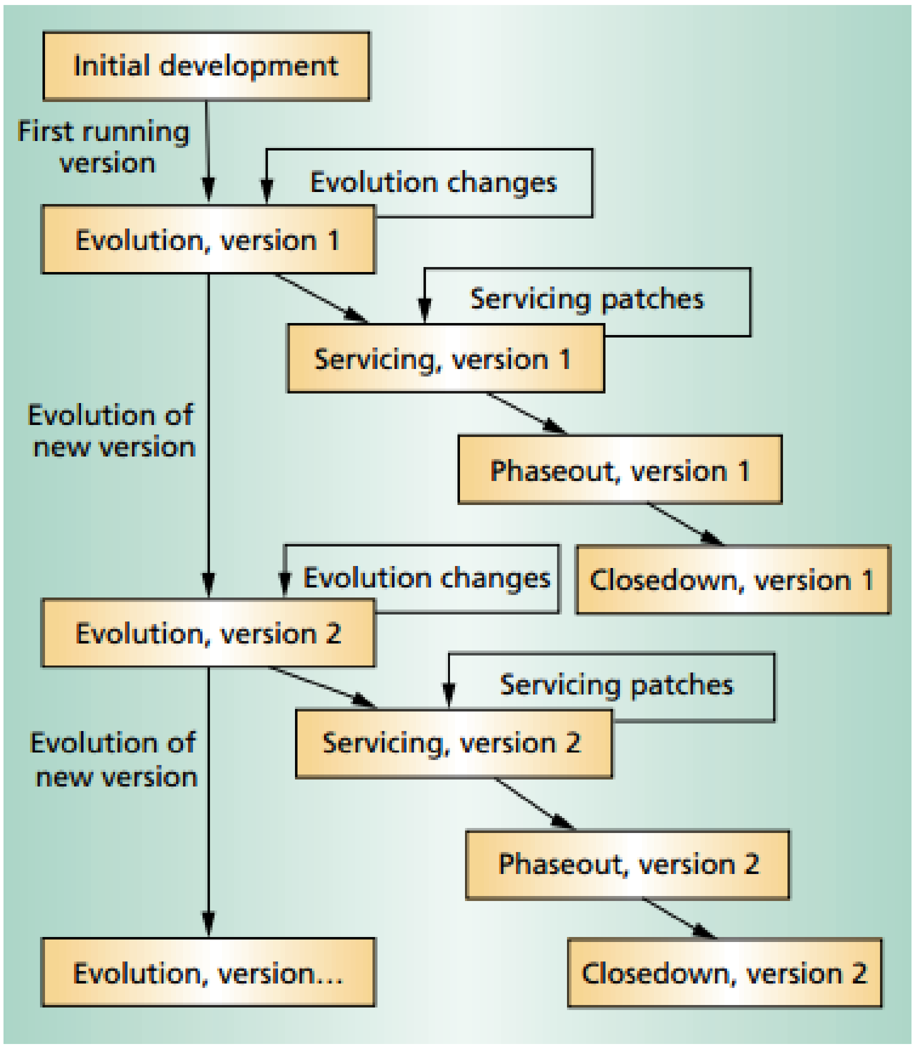

Figure 2.7: Software lifespan

The process starts with a set of proposed change requests. The cost and impact of the change is analyzed to decide wether 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, in order 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, things that require urgent change may appear, such as a serious system fault that must be repaired to allow normal operation. In these cases, the usual process wont 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 [30].

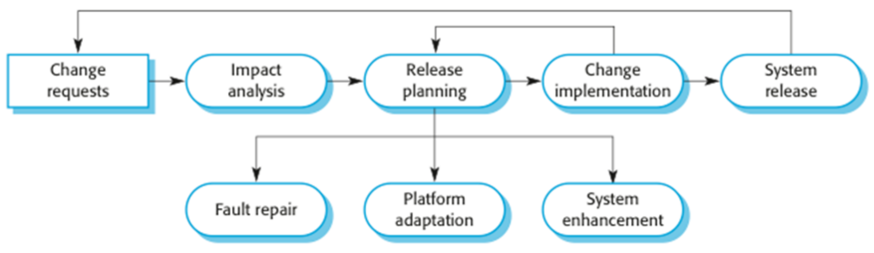

Figure 2.8: Software evolution process

2.4.2 Software maintenance

IEEE 1219 defines software maintenance as follows [35]:

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

Maintenance can be classified into four types [34, 35].

- 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 [3]. 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 [3]. This has not changed since the 1980s when Lientz and and Swanson conducted a study on software maintenance [36]. Their study found out that most severe maintenance problems was caused by poor documentation, demand from users for changes, poor meeting schedulment, and problems training new hires. Some other problem areas was lack of user understand and user training, the customers did not understand how system works.

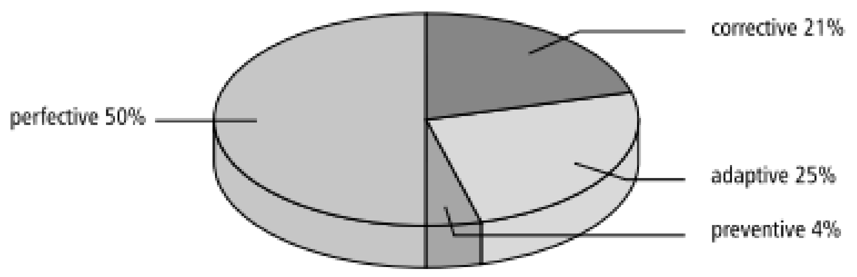

Figure 2.9: Distribution of maintenance activities [3]

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 [37]. Software reuse can be specified in two directions: development for reuse and development with reuse[53][49]. 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 [37].

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

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 external behaviour of a program in order to improve the quality of the system [38]. It is an act of improving the design of an existing system [3]. 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 [32]. It is believed that refactoring is one of the key methods to reduce technical debt in a system as it can be applied to code smells or at the architecture level (Lippert and Roock, 2006).

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 [39] 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 [33]:

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 [33]:
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.

Some examples on SCM is Git-SCM, SVN, RCS, Adele, ClearCase. Version control is the key behind SCM.

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.

Some of the challanges related to development of embedded systems [40]:

- **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.

2.7.1 Continious INtegration

2.8 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 [41]. Due to their operational environment characteristics and common requirements, embedded systems are known as safety-critical and real-time systems [41,42].

This means that properties such as response time and worst case execution time are important design concerns [43]: *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* [44]. It is expected that embedded systems are failure-free [45], but these requirements might hinder embedded systems to deliver reliable service given a disturbance to its services, for example, failure in components [46]. In addition to that, it is expected that embedded systems has long life time (siter). 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]. As it runs on specialized type of hardware, embedded software has multiple contstrains related to run-time, memory usage, processing power etc. In most cases, embedded software developers face many challenges in their work like conflicts in the requirements places on them, for example, low memory usage while ensuring high availability [47]. Additionally, 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.

When developing embedded software, quality is a key characteristic. Managing software quality is necessary to deliver software in a useful, safe, and reliable way [42].

2.8.1 Security

ISO 9126

2.8.2 Maintainability

CHAPTER 3 RESEARCH METHOD

Empirical studies follows two types of research paradigms; the qualitative, and the quantitative paradigm [48]. Qualitative research is about concerned with studying objects in their natural setting [48]. Its data include non-numeric data found in things as interview tapes, documents, or developers' models [1]. Quantitative reseach is concerned with quantifing a relationship or to compare two or more groups [48]. It is based on collecting numberical data [1].

Oates [1] 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 questions. 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 Choice of methods

Out of the six research strategies presented by Oates [1], survey was chosen as a strategy for this research. As mentioned before, quiestionnaries and interviews are two common means for data collection [48]. The method is usually based on the type or results of the research. There are three types of interviews [1]:

- Strucutured interviews: The use of pre-determined, standardized, identical questions for every interviewee. Very similar to questionnaries.
- Semi-structured interviews: A list of themes and questions are 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 interviwer, and the interviewees talks freely about related events, behaviour, or beliefs. The researcher has less control.

This research will conduct semi-structured as the supporthing research method for qualitative data collections.

3.2 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 encountreed technical debt, how they are adressing 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 in the embedded system business, and two of them in the 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 interviewees 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 manne. 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 [1]. Most of the questions were formulated in an open fashion which allows the interviewee to speak with more detail on the issue with their experiences and viewpoints [1,48], hence give a qualitiative and flexible answer.

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 topings makes it possible to extract commond trends and differences in the answers. After the interviews were conducted, the interview data was 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, 4.4, and 4.5. Section 4.2 focuses the term technical debt and causes related to it. Section 4.4 looks at technical debt in their projects. The final section presents examples of how technical debt was incurred and how it was managed.

4.1 Respondents

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

A total of 4 people participated in the interviews. The participants had various background within software development, and had experience with different computer systems. Two of them were from the embedded system field, while the last two from software development field. Table XX summarizes their background, while table XX.

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 XX.

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

The participants were asked to define the term technical debt. I1 defines technical debt as:

> *A way to solve problems that there will be need for refactoring of the solution later. This happens either intentionally or unintentionally. This could be due to external . You might have the perfect payment system, but if you change the currency, the system might fail, because you did not take account to it..*

The definition from I2 is following:

> *If we have systems or software that underlies the level we have as requirement, like systems out-of-support, then we have incurred technical debt as the system operates with bigger risk than what we want.*

He further mentions that running the latest software is not what they want, 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. It is expensive to maintain the product, and people which the competence might be gone.*

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 Problems

The interviewees were also asked why they consider technical debt as a problem. The responses were quite different. Both I2 and I3 explained that incurring technical debt creates problems in the long-term.

> *"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 one of their systems got 40 years of technical debt. I personally 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-term. Many of their projects runs onvery 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 buys systems without having a plan for how to manage it later* - I2.

I4 looks at the problems in a different way. He explaints that there is always code that could be refactored. Their codebase is almost 10 years old, and big parts of the architecture is still the same. There might be some parts of the system that can be improved. He extends his definition on technical debt: *Technical debt is things developers do not like in the code, 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 for Technical Debt

The most common cause for technical debt mentioned was lack of time and resources given for software development and software evolution. Some of the interviewees also mentioned that lack of time generates pressurse which leads to technical debt being incurred intentionally.

I4 said that the shortcuts are taken to make features complete.

> *Sometimes, we need to take shortcuts. We do not take shortcuts in terms of bad code, but rather to make features complete and well-integrated into the solution, in order to meet a deadline. Less important tasks can be postponed* - I4.

I2 said most of the resources they have, are used to add new functionality, or to fix critical errors, in the system.

> *We cannot prioritize upgrading old systems that is up and running. We spend the resources on implementing new functionalities, or fixing critical errors such as system crash. 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.

I2 further explains the situation by an example. One of their systems did not work properly after upgrading the operating system. The question arised was if they should invest more money in paying down the technical debt, or if they should isolate the problem by programming around the problem.

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. Changes being made by developers might not fit with the implemented design.

Architectural decisions, and technology and framework choices, was also mentioned as a cause of technical debt. I1 explained a situation where an architectural decision caused technical debt. He further explains that it is not necessary that the implementation is wrong, but the effects of the outcome. I3 accrued a lot of technical debt as their core product product evolved over the last years. Their previous core product used lots of third-party components, frameworks, and source code. This ultimately led to high amounts of technical debt as the frameworks were out-of-support. I4 mostly has technical debt on their test code.

4.3.1 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 these desicions resulted 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.* However, I4 did mention that sometimes short-term solutions might outweight the future costs, depending on the implementation.

The negative effects tended to be on the longer term. I3 said that there are long-term effects occuring from technical debt. He explained that if technical debt is not managed and reduced, it will have serious effects in the long-term. An example he brought up was one of their systems that used out-of-date and unsupported frameworks, 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 Prioritizing Technical Debt

The interviewees were asked about the current status of technical debt in their systems.

4.4.1 Status

Status, why is it like that, who is aware What decisions are made, who make these decisions Is it incurred intentionally?

P1: There are some visible and some invisible technical debt. Our project does have some, mostly visible that should have been fixed. Reason for it is that the product is new, more important to deliver functionality.

P2: Lots of technical debt in their system, both known and unknown. They buy components from other vendors, and many of these are out-of-support. They usually fix errors by coding around the components, violating the architecture. These components are not supported any more, and hard to change. No documentation or knowledge about the components.

P4: They have some TG in their systems, but it is not that bad. They are able to deal with it. However, they have accumulated some technical debt related to code, known as test debt. They develop frameworks and use them, the test framework has not been changed lately. They also mention that TG is not bad, sometimes you need to incur some debt in order to meet a deadline. For example, a loan from the bank to buy a house. As long as you can pay it ack. Customer gets their product faster.

4.4.2 Incurring Technical Debt

Some of the interviewees revealed that technical debt is incurred intentionally to a certain extent. I1 reveals that both developers and the management takes 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. Hard codig 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*

However, I3 reveals that they do not incur technical debt intentionally. He explains technical debt is incurred intentionally based on third-party sources. *"If a third-party source decides to stop supporting some products or framework you are dependent on, then you incur technical debt"* - I3.

4.4.3 Business and Software Quality

Another aspect that is interesting is how business decisions affects the development team and the overall software quality. The interviewees were asked about t

I2 thought that business decisions do have effect on the amount of technical debt.

> Customer specified functionality, these have to be prioritized for business purposes. Might cause architectural erosion. Things might work, but the solution is dirty. - I3

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

P1 - Not so much focus. Not everyone from the maangement is known with this term. If it works, dont touch.

P2 - If the risk high, we invest. The customers only cares about things work, not how things are behind the walls.

P4 - The quality is good, maybe too good. It is hard to measure the quality, so they looks at feedbacks and reviews from the customers. They also says that they should had more time to deal with technical debt, things that could have been done different. But they are satisfied.

4.5 Management of technical debt

The interviewees were asked how important it is to reduce TD, how they manage technical debt.

4.5.1 Importance of reducing TD

A common answer on the importance of reducing technical is to the product stable .

> *It is important that the software is stable. Upgrading the software is not necessary a goal, it is more important to use a version that is stable. Technical debt can be controlled by the customer, based on the needs. A small change might affect many people. It is important to finish the work early as possible, clean up the mess before you have too much dependencies. A system that has lived for a long time is hard to clean up compared to a new system* - I1

However, reducing technical debt is a challenge itself. I3 mentions 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. That is why they chose for open-source technologies.

4.5.2 Tools and techniques

There are different ways to handle technical debt according to the interviewees; 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 interviewees 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 something*

I3 does not have any overview of their debt it is considered all the time. They spend a lot of time making sure that manage techincal debt through the development phase. However, issues are raised if it is something important. Their goal is to fix it the next release.

The interviewees were asked how they would like to manage technical debt. I1 mentioned that capsulating the code might help in the short-term. The architure will noe be effected 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 chapter discusses the results.

5.1 Summary of the findings

The most common cause of technial debt that was mentioned was time issues. Two of the interviewees mentioned that technical debt is incurred due to lack of time. This is something we recognize from Cunninghams artcile. Neitherless, one of the interviewees mentions that they compensate the customer with a product of high-level quality. The other interviewer calculates the risk on their debt. Highest risk issues are prioritized first. Using third-party solutions is considered as a source of technical debt. Such solutions might be supported for a period. After that, the choice is to replace, or adjust the code.

The research also reveals that technical debt is connected with many different aspects in the software development life cycle. An architectural solution in one of the projects resulted in bad outcomes. This is known as architectural debt. It is impotant to notice that shortcuts was not taken to meet deadline, but the solution itself was not optimal to solve the problem. McConnell defines this as unintentional debt. Software architecture is one of the artefacts that is hard to change, and therefore the debt becomes much higher. Lack of tests, and shortcuts taken is tests were also revealed as a source of technical debt. This is known as test debt. Another source of technical debt that was identified is the combination of old equipment and old code. The problem with old equipment is that they are vulrnearble.

The research revealed that technical debt is incurred intentionally in order to meet a dealine. This happens with or without a plan on how to pay it back. The short-term benefits are customer satisfaction, and that the company gains competitive advantage. However, some of the long-term effects is that the complexity in code keeps increasing, extra workign hours, error and bugs. I1 mentipned that they often has code nights. However, sometimes technical debt is incurred unintentionally. This can be compared with McConnels definition. Technical debt can be unvinsible as well, related to the quadrant.

One of the interviewees also mentioned that they have multiple projects. It seems like they have problems on balancing technical debt and developing new functionalities. Many of their systems does not get updated, such systems creates security breaches and might have troubles with scalability later. Think about it.

It also revealed that the technical communication gap between the team and the management is high. From the interviwerrs perspective, the management remains largely unaware of technical debtm and that they cannot see the value behind technical management. Therefore it is hard to convince the management.

It is interesting to see that the way the embedded area and the software area handles technical debt is different. The studies revealed that technical debt in the software projects are much higher than the embedded system projects. One of the interviewees mentioned that their project used lots of out-of-date third-party components. After changing to open source solutions, they managed to reduce technical debt drastically. This reveals that technical debt is taken much more seriously by the embedded system area. This is due to that their solutions cannot contain any errors. However, both of the embedded systems projects has some technical debt in their project, but it is something they are able to manage. The most important thing is that the product is stable, and the quality is good. HOwever, if we compare the embedded systems, we can see that the reason that the last one is able to manage technical debt is because they develop their own frameworks. Using open-source solutions, or developing frameworks, gives lots of benefits. One thing is that the frameworks can be reused in other projects. TDD is also used in both of these projects, it is important to test the product thoroughly before it is put on production. I4 further explains that their technical debt is more on the frameworks than the product itself. Some lack of tests as well.

5.2 Research Questions

RQ-1: What practices and tools for managing technical debt? How are they used?

The first research questions focuses on strategies of managing and reducing technical debt. Neither of the interviewees had any specific management strategy for technical debt. Neitherless, all of the interviewees were using some practices to manage technical debt. A common solution to handle technical debt issues that was expressed by the interviewees is to make sure that the developers are aware of the technical debt issues. If the company knows that there is some lack of tests, or that a program has architectural issues, the debt would be less significant. 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. Configuration management systems such as Jira, Git, is used for creating change requests. Issue trackers in version control systems are also used for both functionalities and technical debt. Change requests are often made by the developers, and the risk behind every change is calculated by the management.

Practices such as refactoring, bug fixing days, reengineering were also identificed in this research even though the organizations has any strategy for technical debt management. Similar practices has been suggested in other studies (SITER). It is believed that these practices can reduce and prevent the amount of technical debt and also increase the overall quality of the product.

Another method was the use of test-driven development. One of the interviewees used this method. Writing tests helps identifying and dealing with bugs.

RQ-2: What are the most significant sources of technical debt?

The second research questions looks at some of the reasons the organizations incurred technical debt in the software development process. The results suggest that technical debt is not caused by a specific reason. Lack of time in the development phase was identified as the primary reason for technical debt in a software project. Several authors has identified that time is one of the reasons of going into TD (SITER).

Architectural choice was also mentioned as a source of technical debt. It is not related to shortcuts taken just to finish the architecutre, but more than the architecture choice that was taken is not necessary the best choice. The whole architecture is a debt. Such issues are hard to deal with.

Lack of tests was mentioned as a source.

RQ-3: When should technical debt be paid?

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

RQ-4: Who is responsible for deciding whether to incur, or pay off technical debt?

Management is mostly responsible here. Develop team might incur small amount of technical debt. Pay off technical debt depends on the problem. Developers decides sometimes if they got time, but most of the times it is management and project leader.

5.3 Study limitations

5.3.1 Internal validity

5.3.2 External validity

Only 4 people were able to participate in the interviews.

5.3.3 Construct validity

CHAPTER 6 CONCLUSION

Technical debt is something that organizations are unable to avoid during software development projects.

This study reveals that the way traditional software developers and embedded system developer handle technical debt is different.

TD 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 TD 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:

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?

BIBLIOGRAPHY

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

[2] J. Highsmith, "The financial implications of technical debt," 2010.

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

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

[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] Gartner, "Gartner says 4.9 billion connected "things" will be in use in 2015," 2014.

[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] S. Blank, "Organizational debt is like technical debt - but worse," 2015.

[28] "Ieee standard for developing software life cycle processes," *IEEE Std 1074-1991*, pp. 1–, 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] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice.* Addison-Wesley Professional, 3rd ed., 2012.

[31] 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.

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

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

[34] 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.

[35] "Ieee standard for software maintenance," *IEEE Std 1219-1998*, pp. i–, 1998.

[36] 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.

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

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

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

[47] 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.

[48] 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.