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

Who needs an abstract?

Sammendrag

Hvem trenger et sammendrag?

Preface

Here is the preface

Acknowledgements

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

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

1.1 Motivation and Background

The field of embedded systems is growing rapidly based on the evolution in electronics and widespread use of sensors and actuators. From consumer electronics, automobiles, to satellites, embedded systems represent one of the largest segments of the software industry. Software plays a central role in the development of embedded systems. Embedded software is the primary driving force for implementing different functionalities of todays embedded systems. The software is specialized for one particular hardware, and may therefore introduce hardware specific run-time constraints. Embedded systems are growing exponentially [3], and is raising issues in design, development, and maintainability. The main challenge is the technical debt that is not paid by the organization during the software life cycle. Technical debt addresses the debt that software developers accumulate by taking shortcuts in development to meet the organizations business goals. For example, early software release versus maintainability. According to Gartner [4], the cost of dealing with technical debt threatens to grow to $1 trillion globally by 2015. That is the double of the amount of technical debt in 2010.

1.2 Research Context

Type of systems studied

1.3 Research Questions

1.4 Thesis Structure

The thesis is structured into several chapters with sections and subsections. The outline of the thesis is as follows:

- **Chapter 1**: Introduction contains a brief and general introduction to the study and the motivation behind it.
- **Chapter 2**: State-of-the-Art looks at important aspects of the research question.
- **Chapter 3**: Research Method describes how the literature review was carried out throughout the research, as well as a description of the case study to be performed.
- **Chapter 4**: Results presents the results from the case study, and takes a closer look at the findings from the case study.
- **Chapter 5**: Discussion contains a summarized look at the findings from the case study, and connects it with the literature review and to the research questions. An evaluation of the research is also given in this chapter.
- **Chapter 6**: Conclusion concludes the research by providing a summary of the most important points of the results and discussion chapter. Additionally, in outlines possible routes to take in the research field.

CHAPTER 2
STATE-OF-THE-ART

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

2.1 Technical Debt

The concept of technical debt was first introduced by Ward Cunningham in 1992 to communicate technical problems with non-technical stakeholders [5]. The concept was used to describe the system design trade-offs that are made everyday. To deliver business functionality as quickly as possible, *'quick and dirty'* decisions had to be made, which affected the future development activities. Furthermore, Cunningham describes technical debt as *"shipping first time code is like going into debt. A little debt speeds development as long as it is paid back promptly with a rewrite"*. As the time goes, technical debt accumulates *interest*, leading to increased costs of a software system [6, 7]. Li et al. [8] defines interest as *the extra effort needed to modify the part of the software system that contains technical debt*. However, not all debts are necessarily bad. A small portion of debt may help developers speed up the development process, resulting in short-term benefits [6].

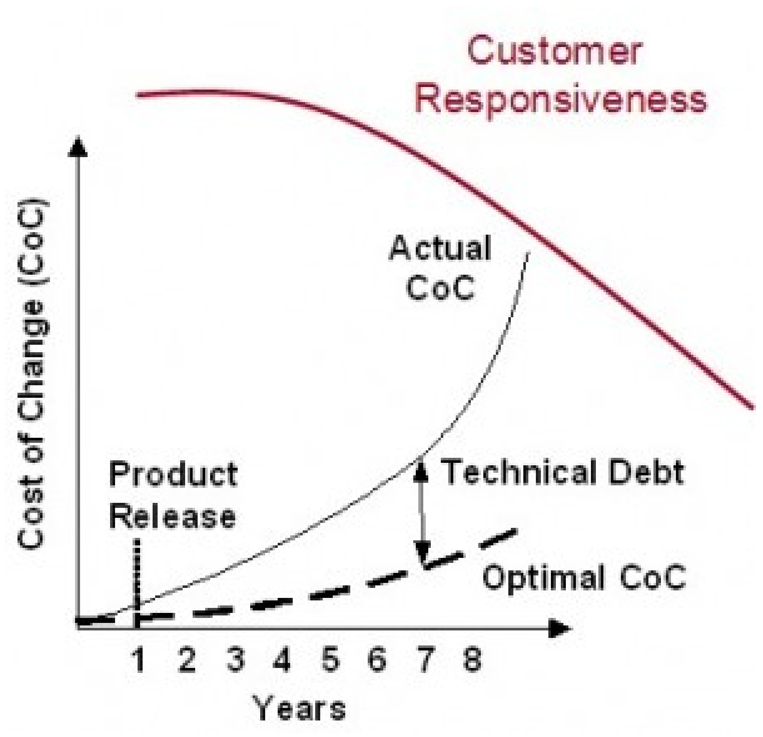


Figure 2.1: The Technical Debt Curve [1]

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

2.1.1 Definitions of Technical Debt

McConnell [9] describes technical debt as *a design or construction approach that is expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including the increased cost over time)*. He splits the term into two categories based on how they are incurred; technical debt that is incurred *intentionally*, and technical debt that is incurred *unintentionally*. For instance, unintentional debt accumulates when a junior software developer writes low quality code due to lack of knowledge and experience. Intentional debt commonly occurs when an organization makes a decision to optimize for the present rather than the future. These type of decisions results in shortcuts being taken to solve a problem.

Fowler [10] presents a more formal explanation of how technical debt can occur. He categories technical debt into a quadrant with two dimensions, which he calls the *"Technical Debt Quadrant"*. As seen in the Figure 2.2, the debt is grouped into four categories:

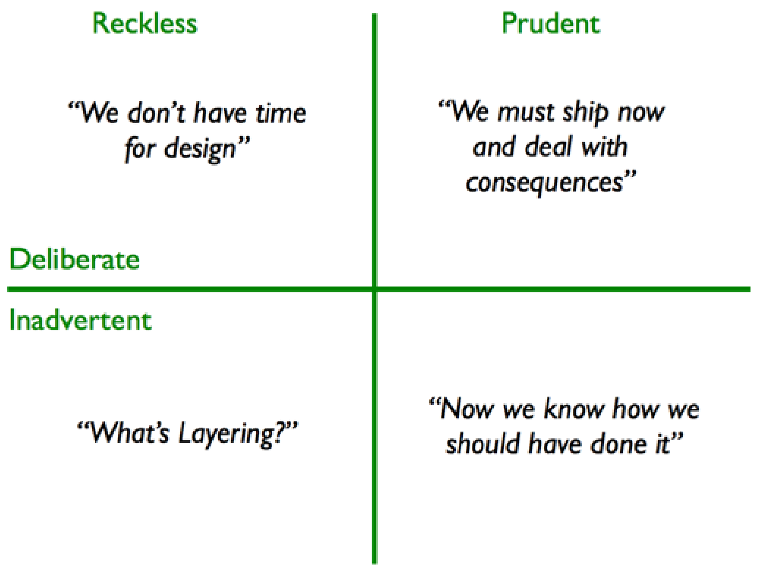

Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 2.2: Fowler's Technical Debt Quadrant

- **Reckless/Deliberate debt**: The team feels time pressure, and takes shortcuts intentionally without any thoughts on how to address the consequences in the future.
- **Reckless/Inadvertent debt**: Best practices for code and design is ignored, and a big mess in the code base is made.
- **Prudent/Deliberate debt**: : The value of taking shortcuts is worth the cost of incurring debt in order to meet a deadline. The team is aware of the consequences, and has a plan in place to address them in the future.
- **Prudent/Inadvertent debt**: Software development process is as much learning as it is coding. The team can deliver a valuable software with clean code, but in the end they may realize that the design could have been better.

Krutchen et al. [11] divides technical debt into two categories; *Visible debt*, that is visible for everyone. It contains elements such as new functionality to add, and defects to fix. *Invisible debt* is the other category, debt that is only visible to software developers. Figure 2.3 illustrates a map of the *"Technical Debt Landscape"*, in which distinguish visible and invisible elements. On the left side of Figure 2.3, technical debt mostly affects evolvability of the software system, while on the right side, technical debt mainly affects maintainability.

2.1.2 Comparison with Financial Debt

Technical debt has many similar characteristics to financial debt [12, 13]:

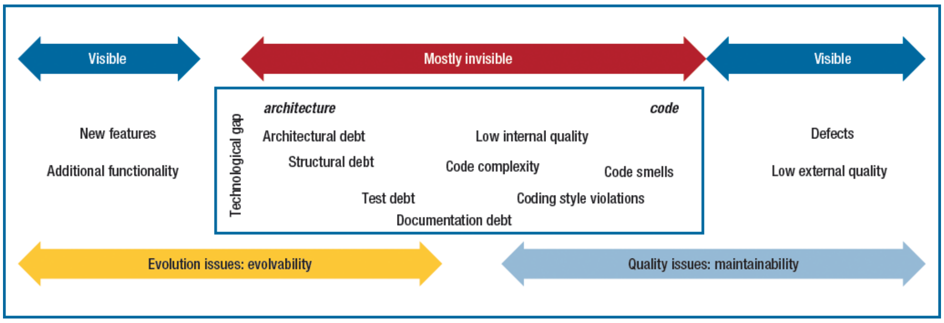


Figure 2.3: Technical Debt Landscape

- 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.

Like financial debt, technical debt accrues interest over time which comes in the form of extra effort that has to be dedicated in future development [6, 7]. Stakeholders can choose to continue paying interest, or to reduce future interest payments by refactoring the problem [10].

However, if the debt is not repaid, the development may slow down, resulting in software project failure or bankruptcy [12]. Moreover, there are some differences between financial debt and technical debt. The debt has to be repaid eventually, but not on any fixed schedule [12]. This means that some debts may never have to be paid back, depending on the interest and the cost of paying back the debt [14].

2.1.3 Causes and Effects of Technical Debt

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

Klinger et al. [7] conducted an industrial case study at IBM where four technical architects with different backgrounds were interviewed. The goal was to examine how decisions to incur debt were taken, and the extent to which the debt provided leverage [7]. The study revealed that the company failed to assess the impact of intentionally incurring debt on projects. Decisions regarding technical debt were rarely quantified. The study also revealed big organizational gaps among the business, operational, and technical stakeholders. When the project team felt pressure from the different stakeholders, technical debt decisions were made without quantifications of possible impacts.

Lim et al. [15] pointed out that technical debt is not always the result of poor developer disciplines, or sloppy programming. It can also include intentional decisions to trade off competing concerns during business pressure. Furthermore, Li et al. explains that technical debt can be used in short term to capture market share and to collect customers feedback early. In the long term, technical debt tended to be negative. These trade-offs included increased complexity, reduced performance, low maintainability, and fragile code. This led to bad customer satisfaction and extra working hours. In many cases, the short term benefits of technical debt outweighed the future costs.

Guo et al. [16] studied the effects of technical debt by tracking a single delayed maintenance task in a real software project throughout its life-cycle, and simulated how managing technical debt can impact the project result. The results indicated that delaying the maintenance task would have almost tripled the costs, if it had been done later.

Siebra et al. [17] carried out an industrial case study where they analyzed documents, emails, and code files. Additionally, they interviewed multiple developers and project managers. The case study revealed that technical debt were mainly taken by strategic decisions. Furthermore, they commented out that using a unique specialist could lead the development team to solutions that the specialist wanted and believe were correct, leading the team to incur debt. The study also identified that technical debt can both increase and decrease the amount of working hours.

Zazworka et al. [18] studied the effects of god classes and technical debt on software quality. God classes are examples on bad coding, and therefore includes a possibility for refactoring [13]. The results indicated that god classes require more maintenance effort including bug fixing and changes to software that are considered as a cost to software project. In other words, if developers desire higher software quality, then technical debt needs to be addressed closely in the development process.

Buschmann [19] explained three different stories of technical debt effects. In the first case, technical debt accumulated in a platform started had growth to a point where development, testing, and maintenance costs started to increase dramatically. Additionally, the components were hardly usable. In the second case, developers started to use shortcuts to increase the development speed. This resulted in significant performance issues because an improper software modularization reflected organizational structures instead of the system domains. It ended up turning in to economic consequences. In the last case, an existing software product experienced increased maintenance cost due to architectural erosion. However, management analyzed that re-engineering the whole software would cost more than doing nothing. Management decided not to do anything to technical debt, because it was cheaper from a business point-of-view.

Codabux et al. [20] carried out an industrial case study where the topic was agile development focusing on technical debt. They observed and interviewed developers to understand how technical debt is characterized, addressed, prioritized, and how decisions led to technical debt. Two subcategories of technical debt were

Table 2.1: Types of Technical Debt

Subcategory	**Definition**
Architectural debt [8, 14, 20]	Architectural decisions that make compromises in some of the quality attributes, such as modifiability.
Code debt [8, 14, 21]	Poorly written code that violates best coding practices and guidelines, such as code duplication.
Defect debt [8, 21]	Defect, failures, or bugs in the software.
Design debt [8, 13, 14]	Technical shortcuts that are taken in design.
Documentation debt [8, 14, 22]	Refers to insufficient, incomplete, or outdated documentation in any aspect of software development.
Infrastructure debt [8, 20, 21]	Refers to sub-optimal configuration of development-related processes, technologies, and supporting tools. An example is lack of continuous integration.
Requirements debt [8, 22]	Refers to the requirements that are not fully implemented, or the distance between actual requirements and implemented requirements.
Test debt [8, 14, 22]	Refers to shortcuts taken in testing. An example is lack of unit tests, and integration tests.

commonly described in this case study; infrastructure and automation debt.

The studies indicates that the causes and effects of technical debt are not always caused by technical reasons. technical debt can be the result of intentional decisions made by the different stakeholders. Incurring technical debt may have short-term positive effects such as time-to-market benefits. Not paying down technical debt can result economic consequences, or quality issues in the long-run. The allowance of technical debt can facilitate product development for a period, but decreases the product maintainability in the long-term. However, there are some times where short-term benefits overweight long-term costs [16].

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

2.1.4 Current Strategies and Practices for Managing Technical Debt

Managing technical debt compromises the actions of identifying the debt and making decisions about which debt should be repaid [9, 11, 14]. This section examines some methods that has been proposed by several authors.

Brown et al. [14] proposed open research questions to understand the need to manage technical debt. The questions includes refactoring opportunities, architectural issues, identifying dominant sources of technical debt, and identifying issues that arise when measuring technical debt.

Lim et al. [15] suggested four strategies for managing technical debt. The first strategy is to do nothing because the technical debt may never be visible to the customer. The second strategy is to use a risk management approach to evaluate and prioritize technical debts cost and value by allocating 5-10% of each release cycle to address technical debt. The third strategy is to include the customers and non-technical stakeholders to technical debt decisions. Finally, the last strategy suggests to track technical debt items using tools like a Wiki, or a backlog.

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

Guo et al. [6] suggested using of portfolio management for technical debt management. This approach collects technical debt items to a *"Technical Debt List"* (Technical DebtL). TDL is used to pay technical debt back based on its cost and value. Three activities support the TDL. The first activity is Technical Debt Identification. Technical Debt Identification uses several tools to identify technical debt items, in which are automatically placed in the TDL. The second activity is Technical Debt Estimation. Each item in the list is assigned the estimates for the debt principal, and the interest. The third activity, Decision Making, is used to determine which debts should be addressed first, and when they should be addressed.

Nugraho et al. [23] proposed an approach to quantify technical debt and its interest by using a software quality assessment method. This method rates the technical quality of a system in terms of the quality characteristics of ISO/IEC 9126.

Krutchen et al. [11] suggested listing debt-related tasks in a common backlog during release and iteration planning. Figure 2.4 illustrates how these elements can be organized in a backlog. Moreover, Krutchen mentioned that project backlogs often contain the green elements. The rest are seen rarely, especially the black elements, they are nowhere to be found.

SonarQube is an open source application for quality management [24]. It manages the results of various code analysis tools, and is used to analyze and measure a projects technical quality. Technical debt is computed based on the SQALE (Software Quality Assessment based on Life-cycle Expectations) methodology [25]. 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.

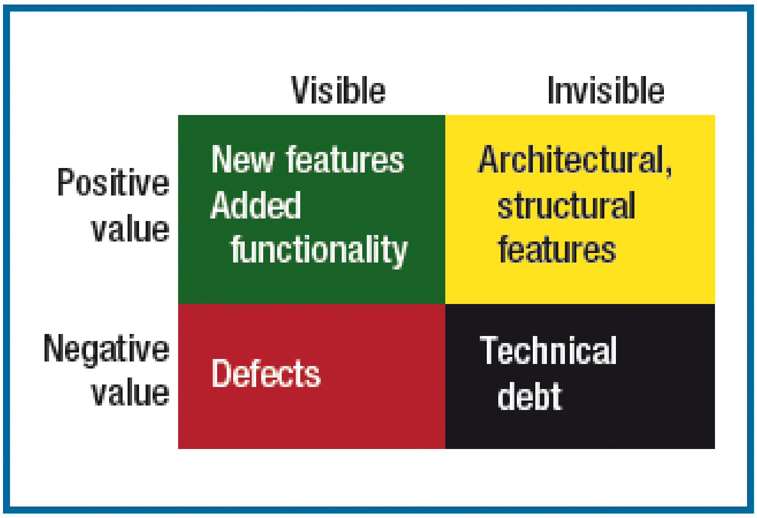

Visible
Invisible
Positive value
Negative value
New features Added functionality
Architectural, structural features
Defects
Technical debt

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

2.2 Software Quality

Software Quality (SQ) is defined as *an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it* [26]. ISO/IEC 9126-2001 is an international standard for evaluating software [27]. It is currently one of the most widespread quality standards [28]. The standard defines quality as *the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.* Additionally, ISO/IEC 9126-2001 offers a valuable conceptual framework for SQ, where it makes a distinction between *"quality in use"*, *"external quality"*, and *"internal quality"* [27, 28]. *Quality in use* is the users view of the quality of a system. *External quality* reflects the dynamic aspect of a software application, and is subdivided into six quality characteristics. *Internal quality* is reflected by a subdivision of the six external quality characteristics into internal quality attributes.

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

2.3 Software Architecture

Bass et al. [29] 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.*

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

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

The architecture of a software is one of the most important artifacts within the systems life cycle [29,30]. 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 [26], activities that consumes a big part of the systems lifespan [2]. The issues of software architecture has long been a concern for those building and evolving large software systems [31].

Software architecture can be seen from two standpoints [32]; *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 [29]. This may be due to developers sloppiness, short deadlines, or lack of documented prescriptive architecture. These principles introduces two new concepts; *architectural drift* and *architectural erosion* [29]. Architectural drift occurs when the documents are updated according to the implementation. The software architecture ends up as an architecture without vision and direction. Architectural erosion occurs when the implementation drifts away from the planned architecture.

System requirements can be categorized as *functional requirements*, *quality attribute requirements* (QA requirements), and *constraints* [29]. *Functional requirements* states what a system must do. *QAs* are the non-functional requirements of a system. *Constraints* is a design decision with zero degrees of freedom. Moreover, long-term responsiveness of a system can be achieved by providing a solution of a system that would enable and achieve systems driving QA.

As a final step in the architecture design phase, more and more organizations are evaluating their design decisions using Architecture Trade-off Analysis Model [29,33]. The goal of the ATAM is to understand the consequences of architectural decisions with respect to the quality attribute requirements of the system.

2.4 Software Life-cycle

A Software Life-Cycle (SLC) is the phases a software product goes through between its convenient, to when its no longer available for use [34]. According to *IEEE Standard for Developing SLC Processes* [34], there are five general groups of related activities in the SLC:

1. The first group is project management. Every software life-cycle starts with the project initiation. Project planning, and project monitoring and control are two other, necessary activities withing this group for each project iteration.

2. The second group is pre-development. This group consists of activities that needs to be performed before the software development phase. Concept exploration is a good example of such activity.

3. The third group is the development itself. It includes the activities that must be performed during the development.

4. The fourth group is post-development. It includes activities to be performed after development to enhance the software project. The retirement activity involves removal of the existing system from its active support by ceasing its operation or support, or replacing it with a new system or an upgraded version of the existing system.

5. The final group is called integral. This group consists of activities that are necessary to ensure successful completion of a project. These activities are seen as support activities rather than activities that are directly oriented to the development effort.

2.4.1 Software Development Life Cycle and Methodologies

A software development process or a software development life-cycle (SDLC) is defined as the process by which user needs are translated into a software product [35]. The process involves translating user needs into requirements, transforming requirements into design, implementing design into code, testing the code, and sometimes, installing and checking out the software for operational use.

A software development methodology is defined as a framework to structure, plan, and control the software development process. Many software development methodologies exists, and the basic life-cycle activities are included in all life-cycle models, often in different orders. The difference is in terms of time to release, risk management, and quality. The models can be of different types, but they are usually defined as traditional and agile software development methodologies.

Traditional Software Development

Traditional software development methodologies are based on a sequential series of steps. They usually starts with elicitation and documentation of a complete set of requirements, followed by architecture and high level design, development, testing, and deployment. The most well-known of these traditional software development methodologies is the Waterfall method, the oldest software development process model [36]. The Waterfall Model divides the software development life-cycle into five distinct and linear stages [2]; requirements engineering, design, implementation, testing, and maintenance. There are many risks associated with the use of Waterfall model [37]:

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

Using sequential design processes such as Waterfall model in software development processes to build complex, intensive systems is often a failure [12]. According to the Standish Group, CHAOS Report of 2015, 29% of all projects using Waterfall method failed in 2015, with no useful software deployed[1].

Agile Methods

To address the challenges posed by traditional methods, agile methods were developed as a set of lightweight methods [37]. 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 [38]. Agile Manifesto describes four values that defines agile software development [39]:

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

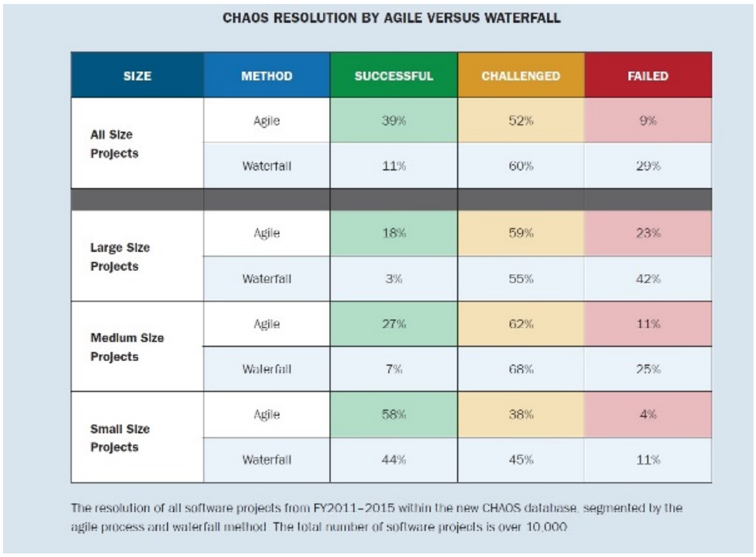
Scrum is one of the most popular agile software development methodologies [2, 26, 40]. It is an iterative and incremental software development model. An advantage with iterative procedures is that parts of a system is developed early on, and can be tested before implementation of other parts. This reduces the risk of having long stages [41]. The idea with Scrum is to divide the development into short periods, called sprints. Unlike the approach in the Waterfall model, the team can estimate how long it will take to implement tasks which can be accomplished during each sprint. To implement the requirements step by step, a product backlog is kept containing the features that have yet to be implemented. The product backlog is not static as it changes to the needs of the project, adding new features, and removing obsolete items. Sprint backlog contains items from the backlog that a team works on during a sprint.

Lean Development adapts the concepts and principles of Toyota Product Development System, to the practice of developing software [42]. It is seen as a key component in building a change tolerant business [43]. The seven lean principles that is applied to software development are summarized as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole* [26, 42].

Despite the fact that agile methods addressed the challenges posed by traditional methods, there are many risks associated with agile methodologies [37]:

- **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.

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



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

- **Large development team**: Having large teams results in difficulties in managing communication between team members.
- **High reliance on human factor**: Agile methodologies relies on the development team, and their abilities to communicate with the customers.
- **Inappropriate customer representative**: This factor can influence the development process and the team members.
- **Distributed development environment**: Agile methodologies requires close interaction between the development team. Having a distributed development environment might challenge the communication between team members due to different time zones.
- **Scope creep**: With minimal planning conducted, developers can be easily distracted from the project main objectives.

2.5 Software Evolution

Increasingly, more and more software developers are employed to maintain and evolve existing systems instead of developing new systems from scratch [40]. Software evolution is a process that usually takes place when the initial development of a software project is done and was successful [44]. 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 [40]. Software evolution is also important because technology tend to change rapidly.

Rajlich et al. [44] proposed a view of the software lifespan, as shown in Figure 2.6. This view divides the software lifespan into five stages with initial development as the first stage. The key contribution is to separate the maintenance phase into an evolution stage, followed by a service stage, and at last the phase-out stage.

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

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

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

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

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

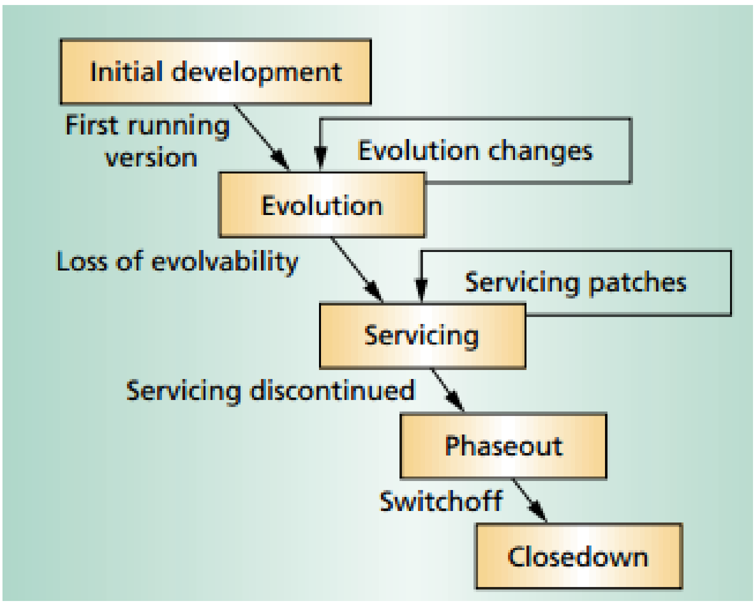


Figure 2.6: Software evolution process

A variation of this process is the versioned stage model, as shown in Figure 2.7.

When a software version is completed and released to the customer, the evolution continues with the company eventually releasing another version and only servicing the previous version.

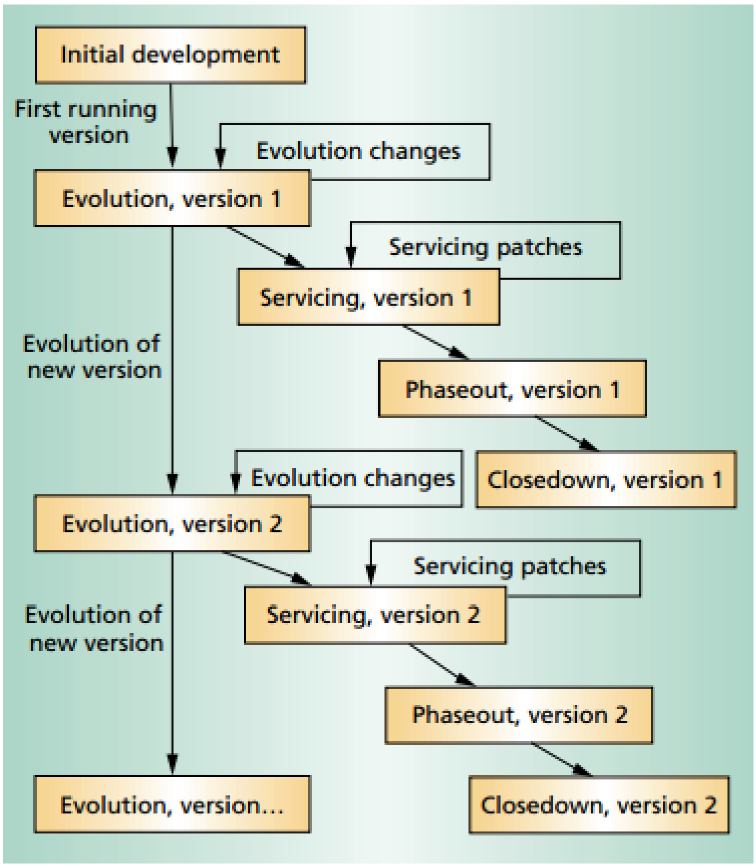


Figure 2.7: Software lifespan

2.5.1 Evolution Processes

Software evolution usually starts with change proposals, which may be new requirements, existing requirements that have not been implemented, or bug reports from stakeholders. The process of implementing a change goes through these stages [40] 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 of urgent changes may appear, such as a serious system fault that must be repaired to allow normal operation. In these cases, the usual process will not be beneficial as it takes time. An emergency fix is usually made to solve the problem. A developer choose a quick and workable solution rather than the best solution. The trade-off is that the the requirements, the software design, and the code become inconsistent. As a system changes over time, it will have impact on the systems internal structure and complexity. Software evolution might cause poor SQ and erosion of software architecture over time [29].

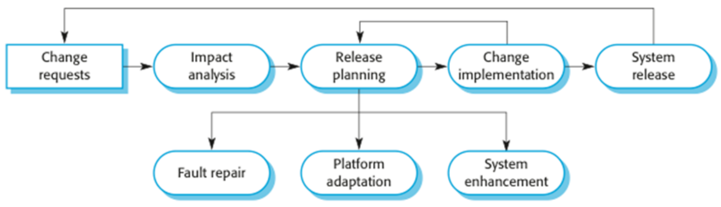


Figure 2.8: Software evolution process

2.5.2 Software Maintenance

IEEE 1219 defines software maintenance as follows [45]:

> *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 [44, 45].

- 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.

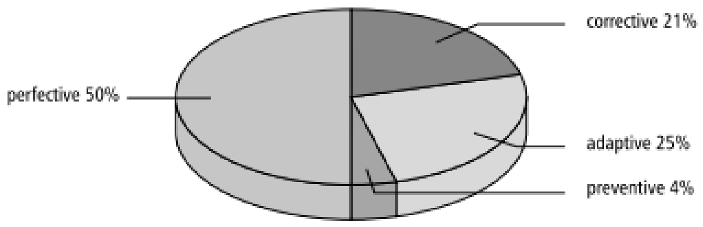


Figure 2.9: Distribution of maintenance activities [2]

According to van Vliet, the real maintenance activity is corrective maintenance [2]. 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 [2]. This has not changed since the 1980s when Lientz and and Swanson conducted a study on software maintenance [46]. Their study found out that most severe maintenance problems were caused by poor documentation, demand from users for changes, poor meeting scheduled, and problems training new hires. Some other problem areas were lack of user understanding, user training, and that customers did not understand how the system worked.

2.6 Software Reuse

Software reuse is the process of using existing software artifacts, or knowledge, to create new software, rather than building it from scratch. Software reuse is a key method for improving software quality [47]. Software reuse can be specified in two directions, development *for* reuse, and development *with* reuse [48]. Development *for* reuse is related to components for reuse or system generalization. Development *with* reuse is related to how existing components can be reused in new system.

Table 2.3 lists several assets from a software project that can be reused [47].

Table 2.3: Reusable assets in software projects

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

Slyngstad et al. [48] conducted an empirical study in Statoil ASA where they investigated developers views on software reuse. The results indicated that reuse include lower costs, shorter development time, higher quality of the reusable components, and a standardized architecture. These findings are very similar to the key benefits of reuse that Lim has described [49]. The quality of software artifacts increases every time the item is reused, because errors are discovered more frequently, making it easier to keep the artifact more stable [50].

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 [51]. Morisio et al. [52] identified three main causes of software reuse failure; not introducing reuse-specific processes, not modifying non-reuse processes, and not considering human factors, combined with lack of commitment by top management.

2.7 Refactoring

technical debt accumulates as developers write code [13], and can be reduced by refactoring. Fowler defines refactoring as means of adjusting the design and architecture towards new requirements without changing the external behavior of a program in order to improve the quality of the system [53]. It is an act of improving the design and quality of an existing system [2]. Most of the time spent on reducing technical debt is on refactoring activities itself. These activities includes planning the design and architecture, rewriting the code, and adjusting documentation [26]. It is believed that refactoring improves software quality and developer productivity, by making it easier to understand and maintain software codes [54], thus a way to manage the technical debt of a system.

Table 2.4 lists the software artifacts that refactoring applies to [55].

2.8 Configuration Management

Systems always change to cope with bugs and introduce new features. A new version of a system is created when changes are made [56]. Dart [57] defines configuration management (CM) as a discipline for controlling the evolution of software systems. CM identifies every component in a project and has an overview of every suggestions and changes from day one to the end of the product. CM involves four related activities [40]:

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

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

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

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 [40]:
a) The consequences of not making the change *b*) The benefits of the change *c*) The number of users affected by the change *d*) The costs of making the change *e*) The product release cycle

Version management is the process of keeping track of different and multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other. This is often done with version management tools, which provide features like
a) version and release identification; *b*) storage management; *c*) change history recording; *d*) independent development; or *e*) project support

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

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

There are some challenges related to development of embedded systems [58]:

- **Complex file sets**: Embedded systems consists of multiple diverse components, both hardware and software. This makes the system complex. Embedded system may also have different adjustable components for a specific platform, making it easier to sell a product by tweaking some parameters. Dealing with these variates is a major challenge. Another challenge is that a product requires correct version of a component. Ensuring the consistency between components and their dependents files is a challenge as well.
- **Distributed teams**: Components may be developed in different places in our world. Two team might for example work on the same components, especially when development are being outsourced. Such collaboration needs every developer to access each others work. The challenge is keep the team synchronized.
- **Management and versioning of intellectual property**: Embedded systems, or software generally might use third-party technologies. It is important that those technologies are up-to-date, and maintained. These updates needs to be traceable such that each components has the right, compatible and stable version of its software. If something is not outsourced, it might be a challenge for developers to contribute and trace their changes.

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

2.9 Embedded Systems

Embedded systems are special-purpose computer systems designed to perform certain dedicated functions under certain constraints. For instance, an embedded system in an automobile provides specific functions as a subsystem for the car itself [59]. According to Crnkovic [60], the various types of embedded systems share common requirements such as: *real-time requirements, resource consumption, dependability, and life-cycle properties*. Moreover, embedded systems are known as safety-critical and real-time systems due to their operational environment characteristics and common requirements [59, 61]. Properties such as response time and worst case execution time, are important design concerns [62]: *"When the break pedal is pressed, the computer should initiate the breaking action within one* *millisecond"*. Embedded systems are expected to be failure-free [63], but these complex requirements may hinder embedded systems to deliver reliable service given a disturbance to its services [64].

2.9.1 Embedded Software

In modern embedded systems, there is always a software element. The *IEEE Standard Glossary of Software Engineering Terminologies* defines embedded systems software as *a part of a larger system which performs some of the requirements of that system* [35]. By 2010, premium class vehicles are expected to contain one gigabyte on-board software [65, 66]. Current BMW 7 series implements about 270+ functions that a user interacts with, deployed over up to 67 embedded platforms [65]. Software constitutes only one part in embedded systems. Integration of these subsystems is always a challenging task for complex systems [65]. Additionally, other issues that needs to be addressed includes unstable requirements, technology changes, location of software errors, and inadequate documentation [67]. According to Ebert, in 2008, there were about 30 embedded microprocessors per person in developed countries with at least 2.5 million function points of embedded software. The volume of embedded software is increasing at 10 to 20% per year depending on the domain [66]. Embedded software developers face many challenges in their work like conflicts in the requirements placed on them. For instance, low memory usage while ensuring high availability [68]. Consider that the role of embedded systems is often critical, the importance of managing software quality is therefore necessary to deliver software in a useful, safe, and reliable way. Table 2.2 in Section 2.2 describes the ISO/IEC 9126:2001 quality attributes.

Embedded Software Quality

Embedded software has many common characteristics with traditional computer-software. However, compared to traditional computer-software development, embedded software development has been proven to be more difficult [33, 66]. Sherman [33] points out that is important that embedded systems not only meet the functional requirements of 'what' they should do, but also the non-functional or 'quality' requirements expected of them. Non-functional requirements have come to be referred to as software quality attributes. Ebert et al. [66] states that quality is difficult to measure in embedded systems. Concerning the design of embedded systems, often discussed behaviors are related to run-time qualities depending on the system domain. For instance, reliability and safety qualities are important in automotive domain [65], while performance and reliability qualities are important in mission critical systems [28]. However, other important design-time qualities directly related to business goals, such as maintainability and usability, are often compromised compared to run-time qualities.

Graaf et al. [3] studied seven European firms to determine the state of the practice in embedded software engineering. One of the key findings in the study was that system engineering was mostly driven by hardware constraints. Software development did not start until the project hit a phase where hardware development changes would be expensive. This led to suboptimal solutions in terms of software architecture, because the system architecture was more or less fixed. Existing software development technologies do not consider the specific needs of embedded systems development.

Risks and malfunctions in embedded software are much higher than in traditional application software. Security rapidly grows in relevance as embedded software communicates autonomously with other computing devices [66]. In the automotive industry, more and more vehicles are getting connected to the cyber-infrastructure [65]. This has resulted in attacks carried out via this cyber-infrastructure. For instance, earlier in 2015, two research discovered the possibility to start Tesla Model S using a laptop[2].

Despite the growth of embedded systems, research has revealed that many embedded system projects do not start from scratch [3, 65]. Several authors have pointed out that software reuse is applied a lot in the embedded system industry. According to Graaf et al. [3], software reuse is rather ad hoc. Most of the products that is developed today is based on previous products. Reuse artifacts, such as requirements and code, are therefore reused by copying them. Furthermore, Pretschner et al. [65] states that the difference between functionality changes from one vehicle generation to the next is relatively small. Most of the old functionality remains and can be found in a new car generation. Functionality differs mostly not more than 10%. However, more than 10% of the software is re-written.

Given the lack of focus on design-time aspects of SQ, the amount of technical debt accumulated in embedded systems grows continuously. This happens because of several factors, and especially because the current level of technical debt is not visible to the developers and managers. According to Graaf et al. [3] and Ebert et al. [66], insufficient requirements and testing are two major cost drivers in embedded software development. 40% of all software defects in embedded systems results from insufficient requirements, especially non-functional requirements [3, 41, 66]. Furthermore, testing after code completion consumes 30 to 40% of embedded development resources, and depending on the project life cycle, testing requires a lead time of 15-50% of total project duration [66]. Moreover, time pressure from the management has been identified as a negative impact on embedded projects, which usually leads to suboptimal solutions.

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

2.10 Technical Debt

2.10.1 Definitions

2.10.2 Causes and Effects

2.10.3 Strategies and Practices for Managing

2.10.4 Identifying

2.10.5 Architectural Technical Debt

2.11 Software Quality

2.12 Software Architecture

2.12.1 Component

2.12.2 Architectural Patterns

2.12.3 Design Patterns

2.13 Software Evolution and Maintenance

2.14 Software Patterns and AntiPatterns

2.15 Software Reuse

2.16 Refactoring

2.17 Component Software

2.17.1 Dependencies

2.18 Embedded Systems

CHAPTER 3

RESEARCH METHODOLOGY

This chapter describes the research methodology that has been used in this thesis to answer the research questions.

3.1 Empirical Strategies

According to Wohlin [69], empirical studies follows two types of research paradigms; the qualitative, and the quantitative paradigm. Qualitative research is concerned with studying objects in their natural setting [69]. It is based on non-numeric data found in sources as interview tapes, documents, or developers' model [69]. Quantitative research is concerned with quantifying a relationship or to compare two or more groups [69]. It is based on collecting numerical data [69].

Oates [70] presents six different research strategies; **survey**, **design and creation**, **case study**, **experimentation**, **action research**, and **ethnography**. ***Survey*** focuses on collection data from a sample of individuals through their responses to questions. The primary means of gathering qualitative or quantitative data are interviews or questionnaires. The results are then analyzed using patterns to derive descriptive, explorative and explanatory conclusions. ***Design and creation*** focuses on developing new IT products, or artifacts. It can be a computer-based system, new model, or a new method. ***Case study*** focuses on monitoring one single 'thing'; an organization, a project, an information system, or a software developer. The goal is to obtain rich, and detailed data. ***Experimentation*** are normally done in laboratory environment, which provides a high level of control. The goal is to investigate cause and effect relationships, testing hypotheses, and to prove or disprove the link between a factor and an observed outcome. ***Action research*** focuses on solving a real-world problem while reflecting on the learning outcomes. ***Ethnography*** is used to understand culture and ways of seeing of a particular group of people. The researcher spends time in the field by participating rather than observing.

3.2 Research Design

- Open source systems - Object oriented

Data generation: Documents

Type: Descriptive, exploratory

Steps: - Pre study (state of the art) - Case study design - Preparation for data collecton - Collecting data - Analysis of data - Discussing the data

Strategy: - Literature review, and experience and motivation -¿ research questions -¿ case study -¿ documents -¿ qualitative/quantitative

3.3 Case Study

CHAPTER 4 RESULTS

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

4.1 Something

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

CHAPTER 5 DISCUSSION

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

CHAPTER 6 CONCLUSION

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

BIBLIOGRAPHY

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

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

[3] B. Graaf, M. Lormans, and H. Toetenel, "Embedded software engineering: the state of the practice," *Software, IEEE*, vol. 20, no. 6, pp. 61–69, 2003.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, (New York, NY, USA), pp. 42–47, ACM, 2013.

[23] 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.

[24] S. SonarSource, "Sonarqube," tech. rep., Technical report, last update: June, 2013.

[25] J.-L. Letouzey, "The sqale method for evaluating technical debt," in *Proceedings of the Third International Workshop on Managing Technical Debt*, pp. 31–36, IEEE Press, 2012.

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

[27] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

[28] J. J. Trienekens, R. J. Kusters, and D. C. Brussel, "Quality specification and metrication, results from a case-study in a mission-critical software domain," *Software Quality Journal*, vol. 18, no. 4, pp. 469–490, 2010.

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

[30] 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.

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

[32] S. Rugaber, "Software architecture and design."

[33] T. Sherman, "Quality attributes for embedded systems," in *Advances in Computer and Information Sciences and Engineering*, pp. 536–539, Springer, 2008.

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

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

[36] J. Krogstie, *Model-based development and evolution of information systems: A Quality Approach.* Springer Science & Business Media, 2012.

[37] 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.

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

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

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

[41] H. Washizaki, Y. Kobayashi, H. Watanabe, E. Nakajima, Y. Hagiwara, K. Hiranabe, and K. Fukuda, "Quality evaluation of embedded software in robot software design contest," *Progress in Informatics*, vol. 5, pp. 35–47, 2007.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

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

[48] 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.

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

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

[51] 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.

[52] 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.

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

[54] 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.

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

[56] K. H. Bennett and V. T. Rajlich, “Software maintenance and evolution: a roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, pp. 73–87, ACM, 2000.

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

[58] 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.

[59] 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.

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

[61] 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.

[62] 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.

[63] 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.

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

[65] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, “Software engineering for automotive systems: A roadmap,” in *2007 Future of Software Engineering*, pp. 55–71, IEEE Computer Society, 2007.

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

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

[68] 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.

[69] 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.

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