NTNU

Managing Technical debt in Embedded systems

Shahariar Kabir Bhuiyan

August 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

Today, hardware and software based systems, further known as embedded systems, are growing rapidly. Ranging from microprosessors in cellphones to sensors in cars and surveillance systems, embedded computing is becoming a big part of our lives. With the rapid growth of embedded systems, we clearly see that most future computing systems will be embedded systems [24]. With the new era of Internet of Things, more and more embedded systems are getting a networked interconnection. This leads to a distributed network of devices communicating with other devices as well as human beings. Gartner has estimated that in 2020, 25 billion connected "things" will be in use [12]. To provide more functionality, multiple components are combined together with embedded systems. However, as the complexity of embedded system increases, the ability to maintain the quality of such systems becomes more difficult. Combination of multiple components leads to higher costs of verifying additional software and many fails to test the product properly and deliver a reliable product. Such systems accumulates technical debt. As debt accumulates, it becomes necessary to manage the overall debt while keeping the system flexible and

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**

Table 1.1: Table from Gartner [12]

extensible. Companies must often recall their products. If they could catch these software defects earlier in the system design process, they would have saved a lot of money. Embedded systems also has long lifetime and its important to find out how to make decisisions so future maintenance and evolution has 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 [18]. That is the double of the amount 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.

1.2 Research Questions

The main objective of this project is to increase the knowledge on how 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 these people might not be available.

The research questions will be:

- **RQ-1**: Are there any 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 debt be refactored?
- **RQ-4**: Who is responsible for deciding whether to incur, and pay off technical debt?

1.3 Research method

The most relevant research methologies in software engineering is summarized in Figure 1.1. This model will be used as a basis in this thesis. In order to define the research questions, it is necessary to an overview of the research field. To do this, one can either conduct a review of published research within the selected area of study, or use experiences and motivations. Following this, 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. To produce empirical data or evidence, a data generation method is needed. There are four methods: interviews, observations, questionnaire, and documents. These data can either be quantitative or qualitative.

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

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 state-of-art within the field of techincal debt, embedded

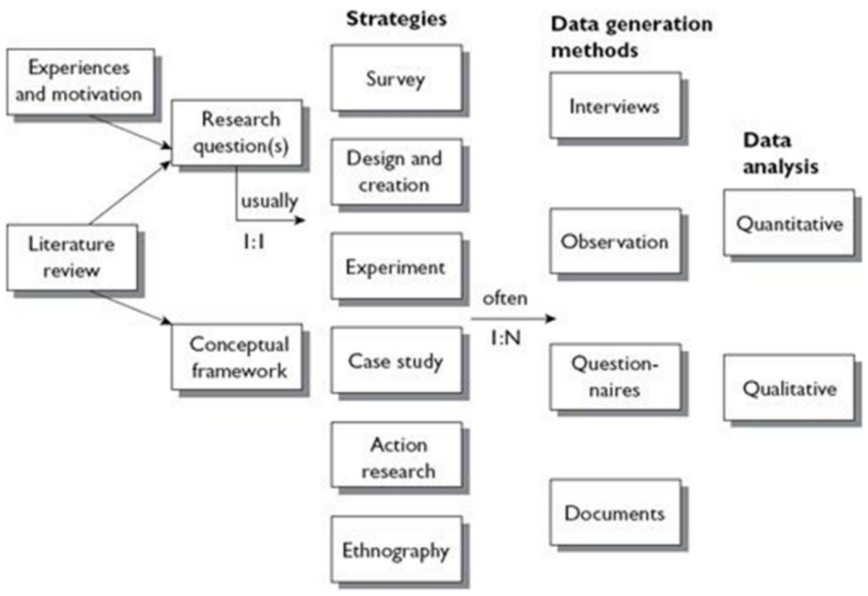

Figure 1.1: Model of research process [21]

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 the state-of-the-art topics which is relevant to this research 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 in order to communicate the problem with non-technical stakeholders [8]. 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 affects 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 [13, 15]. However, not all debts are necessarily bad. A small portion of debt might help developers speed up the development process in

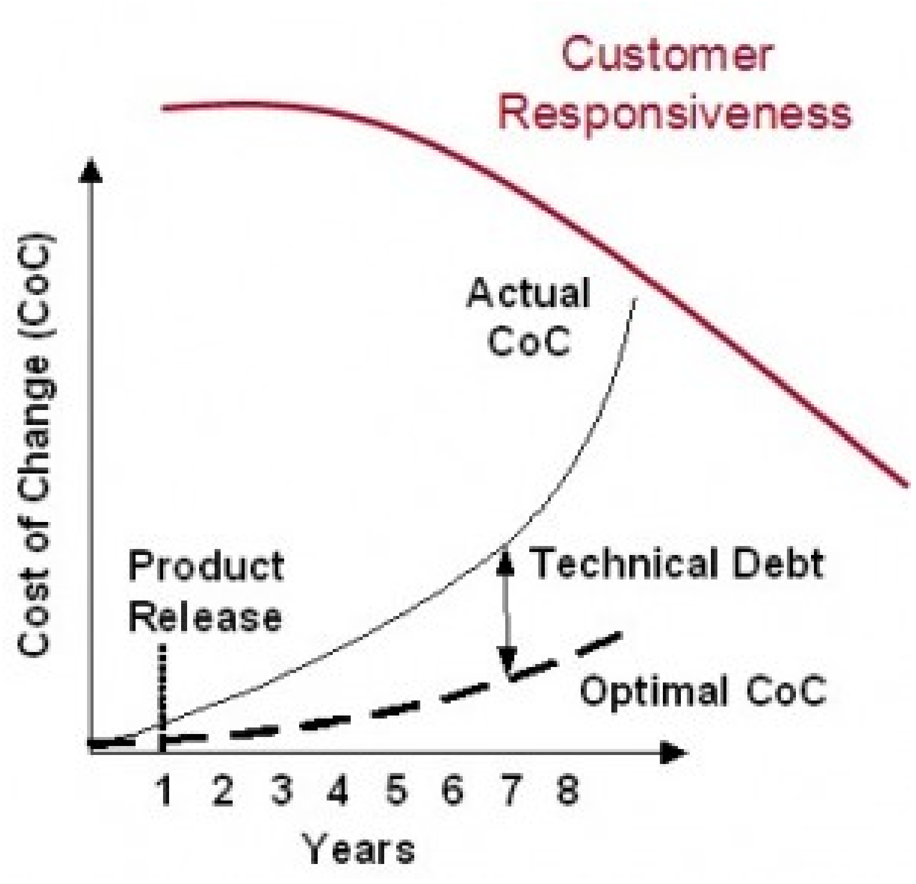

Figure 2.1: The Technical Debt Curve [14]

short term [13].

Figure 2.1 illustrates what happends as technical debt grows within a software product over time. 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 Comparation with financial debt

Techincal debt has many similarities as financial debt [3, 25], which can be seen as the following:

- You take a loan that has to be repaid later
- You usually repay the loan with interest

- If you can't 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 have to be dedicated in future development because of bad choices [13, 15]. You can choose to continue paying the interest, or you can pay down the debt by refactoring the code into something better which reduces interest payments in the future [10]. 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 [3].

There are some differences between financial and technical debt as well. The debt has to be repaid eventually, but not on any fixed schedule [3]. 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. The person who takes the debt is not necessarily the one who has to pay it off. A software project which moves from development mode to maintencance mode might change the engineers as well. So the engineers who has to maintain the system are the one who has to pay back the debt which occured in the development mode. Developers are often rewarded for their implementation speed. 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 [3].

2.1.2 Types of technical debt

Since the original definition by Cunningham, numerous people have proposed definitions to the term technical debt. McConnels splits the term into two categories based on how they are incurred, intentionally or unintentionally [20]. The unintentional category includes debt that comes from doing a poor job. For instance, 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 wont be a next release. This leads to bad decisions, like taking a shortcut to solve a problem, and reconcile the problem after shipment

Fowlers presents a more formal explanation of how techincal debt can occur [10]. He categories technical debt into different debt types, in which he calls *"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.
- **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 [17]. 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.

2.1.3 Causes

Many thinks that technical debt is mostly related to code decide. However, technical debt might occur already in the requirement specification phase in software

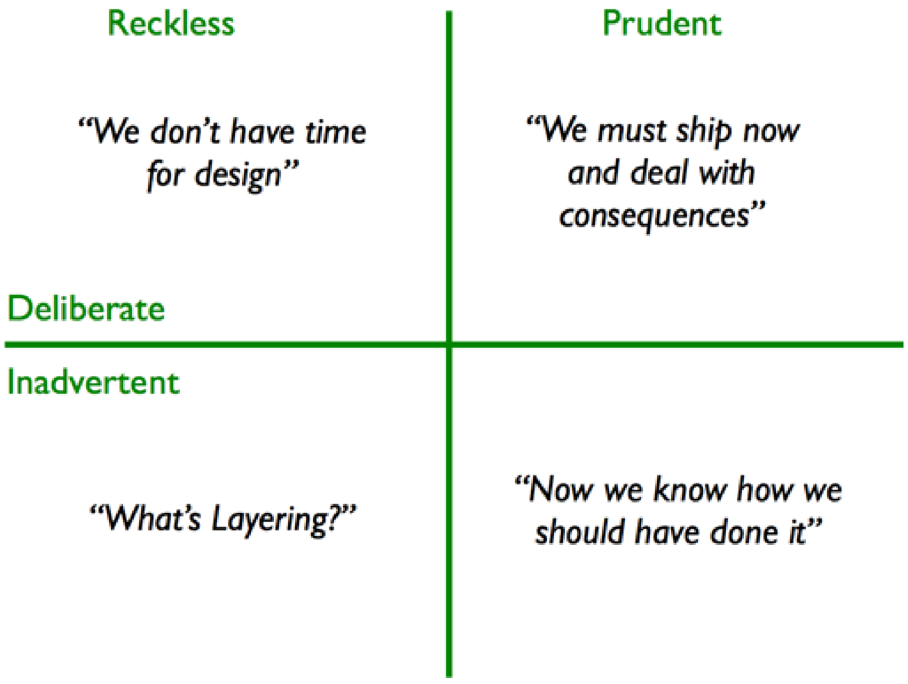
Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 2.2: Technical Debt Quadrant

development project.

While financial debt occus from deliberate actions of taking a loan, technical debt may be caused by several factors. Antares categories these factors into four groups:

- Work processes: Software development methology, can some tasks be automized (with a deploy script), is tests written after bug fixing, do you map and document shortcuts you take, is there any plans for techincal debt management later, is it important to implement new functionality or to make sure that the existing ones work propertly?

- People (knowledge and capacity): Do you need some individuals to finish a task, Do you have the right people for the job, is enough training given to new people, what happens if you need someone who's on vacation/change project/is sick or something. You should keep this in mind and make a plan on how knowledge is transferred. Techcnical debt can be the reason for poor motivation and productivity which causes you to work poorly.

- Technology: Is solutions hard to integrate with other solutions, is all the systems out there compatible with newer technology, is there any outdated or duplicated code in the system, is all the systems secure, is the solutions old, or user friendly, is there some code which is hard to maintencance.

- Collaboration in the organization: Commuinication between developers and requirement people. You often get a list with requirements, is the list understandable? Do we work with a backlog with tasks that should have been solved long time ago, but not which is not ”actual” now?

Developers might not care about the product because they don't feel that they ”own” what's being made. They get told what do to, but not more than that. Can't make requirements.

Operating technical debt might be to maintenance and manage existing code rather than implementing new functionality. It is important to keep track of the technical debt, and incur interest payments, before it makes troubles for you. Do do that, you could for example set up a plan for repayment which tells you something about how the debt shall be repayed. Scrum can be used to do this for example, where you split the repayment plan to smaller parts where you estimate and prioritize tasks. It is important to remember that taking too much loan might cause problems. As mentioned ealier, technical debt can be seen as taking a financial loan according to Cunningham. The loan has to be repaid with interests. Technical debt uses time and effort as repayment. It is acceptable to take a loan, but it should be controlled. Do not take loan than what you are able to handle. Think with your head.

Main developers behind a software project aren't the one who usually maintain the code. Companies often has policies where they transfer a project to new developers after the development phase is over. The reasons could be to save money, or that the new developers might work more. These maintenance people often haas to repay the debt. The main developers gets awarded for their implementation speed rather than thinking on maintanence and evolution. They can often be placed on new projects before the debt has to be paid back, making them unavaiable for a period. Too few systems also has TODO or FIXME comments in their source code.

2.1.4 Measuring technical debt

Define technical debt in your backlog. Use tools like Sonar, SQALE.

Technical debt isn't the problem, but it can be seen as a symptom of a potential problem. THings might go wrong with the organization which is causing techncial debt. Business processes, operational, business methods.

2.1.5 Technical debt in Industry

Technical debt today is connected with many differnet aspects in the software development process, like documentation debt, requirements debt, architecture debt etc [9].

All development results in some amount of technical debt. The challenge is to manage it, reduce it and develop practices to keep it at a level that does not impact performance and availability of your critical business services.

Using sequential design processes in software development processes to build complex, intensive systems is often a failure. Requirements are specified at the beginning of the software development process, and the remaining software development activities has to follow the initiral requirements. This kind of model is not appropiate to use for big softwares where technology and business requirements always change. This is why agile methods was made, where change and feedback is important. One of the benefits is the ability to quickly release new functionality. However, one of the problems with agile methods is that developers often wants to focus on implementing new fucntionality, which results in poor focus on design, code quality, testing, which again leads to technical debt.

Klinger 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 was taken and the extend to which the debt provided leverage [15]. What they found out was 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.

Codabux carried out an industrial case study where the topic was agile development focusing on techincal debt [7]. They wanted to gain insights on agile adoption and how techincal debt affects the development processes. This industiral case study happened in form of an interview. After conducting the research, they got two definitions of technical debt, further known as infrastructure and automation debt. Infrastrucutre is the work that improves the teams processes and ability to produce a quality procudt with refactoring, repackaging, developing unit tests. Automation isthe work relatedd to supporting continious integration and faster development cycles. The term technical debt was mostly related to design, testing and defect debts according to the participants. One of the engineers mentioned that when they were working, they didn't know the "balacne of the credit card. They kept charging it". Management decides when enough debt has incurred, and is influenced by the customer needs.

2.1.6 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 [6]. 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 [6].

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

2.2 Embedded Systems

With the rapid evolution of electrical and software based systems, known as embedded systems, we see that most of the future compuing systems will be embedded systems [24]. However, as the complexity of embedded systems increases, maintaining the quality of such systems becomes more difficult. Higher functionality is provided when multiple components are combined together with embedde systems. This type of combination leads to higher cost of verifying additional software, which makes many fail to test the product properly and deliver reliable products. Companies must often recall their products, and catching these software defects earlier in the system design process saves a lot of money. The ability to identify these kind of problems earlier is still something many companies has troubles with. Embedded systems has also long lifetime and it's important to find out how to make decisions so future maintenance and operation has low cost as possible. Technical debt is a big factor in embedded systems as developers might not be available years after implementaion.

Since embedded systems are some specialized hardware,

It is fairly safe to assume that a bsuiness in the coming years will be much more connected to the outside worl than it is now. This connectivity might come through mobile applications, social networks or the clouds.

If legacy software can be maintaned to an acceptable level

2.2.1 Embedded system software

Embedded software is a computer software for embedded systems. It is specialized for a type of hardware which it lays and runs on. Therefore, embedded software might have multiple constraints related to run-time, memory usage, processing power etc.

Embedded software had an important role today with the rapid evolution of ES. Escpecially with the Internet of Things trend.

However, there are some challenges with embedde system software. These type of software usually has long lifetime. Old systems are usually hard to maintain compared to new ones. Companies must maintain many different configurations, and maintaining systems are challenging due to time. It is important to take the right choices when designing such systems. Abstract and high level design, architecture. People tend to deliver something in time rather than making something good.

2.2.2 Security

2.2.3 Virtualization of embedded systems

2.3 Configuration management

Dart defines configuration management as a dicipline for controlling the evolution of software systems (siter dart). This includes content, changes and status in a shared project. Both processes and technical solutions to handle changes and the projects integrity. Example, if a new version of the product is released, everything related to this projects needs to be up-to-date. Like documentation. Configuration management identifies every component in a project and has an overview of every suggestions and changes from day 1 to the end of the product. Some examples on SCM is Git-SCM, SVN, RCS, Adele, ClearCase. Version control is the key behind SCM. IEEE standard 729-1983 highlights the following operational aspects of CM:

- **Idenfitication**: The products structure. Identifies every component in the products, making them unique and accessible in some form.
- **Control**: Controls every reelase and changes of a product throghout the life-cycles by having some controls in place that ensure consistent software via the creation of ab aseline product.
- **Status accounting**: Records and reports the status of components and change requests. It is also possible to review statistics about components in the product.
- **Audit and review**: Validates the product and maintaining the consistency between components by ensuring that the product is a well-defined collection of compontens.

In addition to those aspects, we can extend the definition of CM by three more aspects

- **Manufacture**: Maintaining the construction, and build the product in an optimal way.
- **Process management**: Take care of the companies policies, processes and lifecycle model.
- **Teamwork**: Review the work and make sure that the collaboration is good. Keep an eye on the interaction between multiple users and the product.

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:

- **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.

2.4 DevOps

Continious Integration, Continious Delivery, Automation, Deployment

2.5 Software Architecture

Bass, Klements and Kazman [4] defines software architecture as following:

> The software architecture of a system is teh 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 [4, 16]. Architectural design decisions that are made during the design phase affects 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 [22], activities that consumes a big part of the systems lifespan [23].

Architectural drift - the architectural documentation is updated according to implementation, but ends up as an architecture without vision and direction. Architectural erosion - the implementation drifts away from the planned architecture so they are no longer consistent.

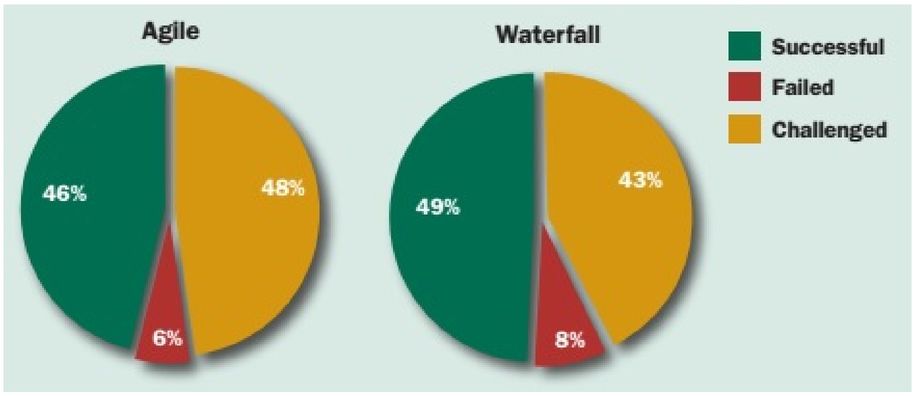

Figure 2.3: The CHAOS Manifesto, The Standish Group 2012

2.6 Software development methods

2.6.1 Waterfall

2.6.2 Agile

2.6.3 V model

2.6.4 Prototyping

2.6.5 Test-driven development

2.7 Software maintenance and evolution

Increasingly, more and more software developers are employed to maintain and evolve existing systems instead of developing new systems from scratch "trenger sitat". Software evolution is a process that usually takes place when the initial development of a software project is done and was successful [5]. The goal of software evolution is to incorporate new user requirements in the application and adapt it to the existing application. This phase is important beacuse it takes a lage part of the overall lifecycle costs. It is also important because these days technology tend to change rapidly, and not following these trend means loosing business oppertunities.

IEE 1219 defines software maintenance as follows [1]:

> 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 [1, 5].

- 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 corrective maintenance [23]. 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 [23]. This hasn't changed since the 1980s when Lientz and and Swanson conducted a study on software maintenance [19]. Their study found out that most severe maintenance problems was caused by poor documentation, demand from users for changes, difficulty meeting schedulment, and problems training new hires. Some other problem areas was lack of user understand and user training, the customers didnt understand how system works. Programmers had low productivity, skill level and motivation. System was badly designed leading to low quality.

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

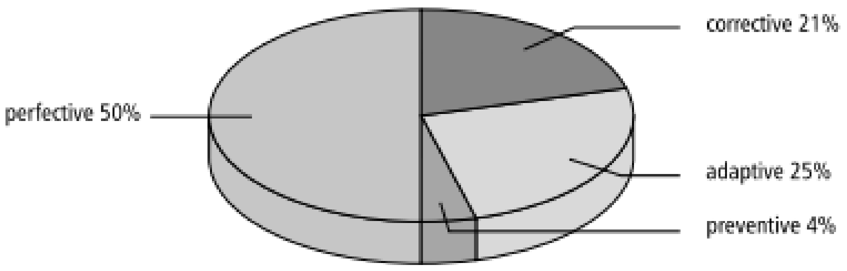

Figure 2.4: Distribution of maintenance activities [23]

of software architecture over time [4].

2.8 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 [11].

2.9 Refactoring

Design debt, a specific type of technical debt, accumulates as you write code [25]. This type of debt can be reduced when you refactor. Fowler defines refactoring as means of adjusting the design and architecture towards new requirements without changing the externial behaviour of a program in order to improve the quality of the system [2]. It is an act of improving the design of an existing system [23]. 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 [22]. It is believed that refactoring is one of the key methods to reduce technical debt in a system (”siter”).

CHAPTER 3 RESEARCH METHOD

3.1 Choice of methods

Qualitative methods were chosen. To answer our questions, an interview were executed.

3.2 Participants

To analyze our questions, several companies were interviewed.

3.3 Data collection

This study was performed by collecting data through a series of interviews with companies in the software business.

3.3.1 Interviews

The interviews were conducted at many companies. The interview question were focused around how the companies encountered technical debt, what they did to manage it initially, how they managed their technical debt problems over time.

The interviews were mainly conducted in Trondheim.

CHAPTER 4 RESULTS

This chapter presents the findings from the case that was performed.

4.1 Section Title

4.1.1 SubSection Title

4.2 Section Title

4.2.1 SubSection Title

CHAPTER 5 DISCUSSION

This chapter discusses the results.

5.1 Yee

CHAPTER 6 CONCLUSION

This is a conclusion. Nice.

6.1 Future work

Make an app for managing technical debt? Make an app for crawling through your source code, visualize all modules and their dependices etc?

BIBLIOGRAPHY

[1] Ieee standard for software maintenance. *IEEE Std 1219-1998*, pages i–, 1998.

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

[3] Eric Allman. Managing technical debt. *Commun. ACM*, 55(5):50–55, May 2012.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, 3rd edition, 2012.

[5] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[6] Steve Blank. Organizational debt is like technical debt - but worse, 2015.

[7] Zadia Codabux and Byron Williams. Managing technical debt: An industrial case study. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, MTD '13, pages 8–15, Piscataway, NJ, USA, 2013. IEEE Press.

[8] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

[9] Davide Falessi and Philippe 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, pages 28:1–28:4, New York, NY, USA, 2015. ACM.

[10] Martin Fowler. Technicaldebtquadrant, 2009.

[11] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

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

[13] Yuepu Guo and Carolyn Seaman. A portfolio approach to technical debt management. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 31–34, New York, NY, USA, 2011. ACM.

[14] Jim Highsmith. The financial implications of technical debt, 2010.

[15] Tim Klinger, Peri Tarr, Patrick Wagstrom, and Clay Williams. An enterprise perspective on technical debt. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 35–38, New York, NY, USA, 2011. ACM.

[16] Jens Knodel, Mikael Lindvall, Dirk Muthig, and Matthias Naab. Static evaluation of software architectures. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10–pp. IEEE, 2006.

[17] Philippe Kruchten, R.L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *Software, IEEE*, 29(6):18–21, Nov 2012.

[18] Andrew Kyte. Gartner estimates global it debt to be 500 billion dollars this year, with potential to grow to 1 trillion dollars by 2015, 2010.

[19] Bennet P Lientz and E Burton Swanson. Software maintenance management: a study of the maintenance of computer application software in 487 data pro-

cessing organizations. 1980.

[20] Steve McConnell. Technical debt, 2007.

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

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

[23] Hans van Vliet. *Software Engineering: Principles and Practice.* Wiley Publishing, 3rd edition, 2008.

[24] W. Wolf and J. Madsen. Embedded systems education for the future. *Proceedings of the IEEE*, 88(1):23–30, Jan 2000.

[25] Nico Zazworka, Carolyn Seaman, and Forrest Shull. Prioritizing design debt investment opportunities. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 39–42, New York, NY, USA, 2011. ACM.