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.

Acknowledgement

I want to thank...

CONTENTS

I Introduction **2**

1 Introduction **3**

1.1 Motivation and goals 3

1.1.1 SubSection Title 3

1.2 Problem outline 3

1.3 Project structure 3

II Teoretical background **4**

2 Technical Debt **5**

2.1 What is technical debt 6

2.1.1 Comparation with financial debt 6

2.1.2 Types of technical debt 7

2.1.3 Organizational debt 7

2.2 Why does it occur 8

2.3 Technical debt in Industry 9

3 Embedded Systems **10**

3.1 Introduction 10

3.2 Embedded system software111 10

3.3 Virtualization of embedded systems . . . 11
3.4 Configuration management . . . 11
3.4.1 Software reuse . . . 14
3.4.2 Software development life cycles . . . 14
3.4.3 Software evolution . . . 14
3.4.4 Deployment . . . 14

4 Research method **15**
4.1 Section Title . . . 15

III Fourth part **16**

5 Results **17**
5.1 Section Title . . . 17
5.1.1 SubSection Title . . . 17
5.2 Section Title . . . 17
5.2.1 SubSection Title . . . 17

IV Discussion part **18**

6 Discussion **19**
6.1 Yee . . . 19

V Last part **20**

7 Conclusion **21**

VI Appendix **22**

Part I

Introduction

CHAPTER 1

INTRODUCTION

1.1 Motivation and goals

1.1.1 SubSection Title

1.2 Problem outline

1.3 Project structure

Part II

Teoretical background

CHAPTER 2

TECHNICAL DEBT

Du har to uker igjen av til prosjektet skal leveres inn. Du sitter og jobber med en funksnonalitet som er veldig krevende. Du har kommet opp med to ulike løsninger. Dene ene er kjapp og stygg, men sørger for at prosjektet blir levert i tiden. Den andre vil kreve mer tid og du vil ende opp med å utsette levering, men på bekostning av god design.

You're currently working on a project. It's short time for delivery, and yet many functionality which needs to be implemented. You have two choices when implementing a functionality. You can deliver a less good solution (quick and dirty) where you'll deliver the functionality in time, but we trade off design and quality, which makes operation, management and maintenance hard to do. The other way is a much cleaner result, but in exhange of more time to put the functionality in place. This is what technical debt is, when shortcuts are taken to solve problems trading off quality and design. Technical debt is a big problem today. In 2010, the total debt was around 500 millions dollar, while it's estimated that in 2015 the technical debt will be around 1 billion dollar.

In order to understand and investigate the consequences of technical debt, one needs to know what technical debt is, the reasons for it, and how it can be eliminated.

2.1 What is technical debt

The concept of technical debt was first introduced by Ward Cunningham in 1992[1]. He defines the term in terms of bad code . Like financial debt, the technical debt incurs interest payments, which comes in the form of extra time it takes to redesign or refactor the code, or otherwise fix the problem[**allmann**]. This connects the term technical debt to the problem of "deciding when and how to refactor a system to improve its structure as a basis for future evolution".

2.1.1 Comparation with financial debt

Financial debt can be seen as the following: - You take a loan, and you have to repay it eventually. - You usually pay back with interest - If you can't pay back, a very high cost will follow. For example, you'll loose your house.

Technical debt has some of the following characteristics as financial debt. Debt has to be payed back with interests. Failure to fix the problem may lead to software project failure - the customer gives up and goes somewhere else, the system has to be rewritten, or collapse of the company.

However, there are some differences as well. The debt has to be repaid eventually, but not on any fixed schedule. 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 necessaryly 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. However, technical debt is not about bad code design. In practice, it's much more than that. Example on interests might be lowr 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.

2.1.2 Types of technical debt

McConnels defines two categories based on how they are incurred, intentionally or unintentionally. 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 formal explanation of how techincal debt can occur. He categories technical debt into different debt types, in which he calls "Technical Debt Quadrant". As seen in the figure, the debt is grouped into four categories: reckless deliberate/inadvertent and prudent deliberate/inadvertent. Reckless-deliberate is descibed as "we dont have time for design", while reckless-inadveretnt is described as "waht's layering?". Prudent-deliberate is described as "we must ship now and deal with the consequences" and prudent-inadvertent is described as "now we know what we should have done".

Krutchen divides technical debt into two categories. 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 Organizational debt

While technical debt is known problem, startups usually accrue one more type of debt, called organizational debt. 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. While technical debt is issues within the software which hampers its maintenance, organizational debt is issues preventing a company from running smoothly.

2.2 Why does it occur

When you Developers often wants to deliver something in time, and shortcuts is often used. However, technical debt might occur already in the requirement specification phase in software development project.

Technical debt may be caused by several factors: - 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.

2.3 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.

Using sequential design processes in software development processes to develop big software is often a failure. Requirements are specified in the beginning, and rest of the steps in this model has to follow these requirements which can't be changed. There's no benefits using this model for big softwares as technology and business requirements always change.

This is why agile methods was made, where change and feedback is important. However, one big problem with agile methods is that developets often wants to focus on implementing new fucntionality, and have very poor focus on design, code quality, testing, which again leads to technical debt.

There are multiple classification on techincal debt, which can occur everywhere in the software development life cycle. Like documentation debt, design debt, code, testing etc.

Architecture violations might hiundter future development as it might be hard to extend.

Klinger

Codabux

CHAPTER 3

EMBEDDED SYSTEMS

3.1 Introduction

Embedded systems is a combination of hardware and software to perform a specific task. Embedded systems has a big role today as

3.2 Embedded system software111

Embedded software er en slags programvare for innebygde systemer. Disse programvarene er spesialisert for en type hardware (som den ligger og kjører på). Disse programvarene har derfor spesifikke begrensninger når det kommer til run-time, som minnebruk, prosesseringskraft osv.

ES har en store rolle idag siden embedded systems utvikles stadig, spesielt med IoT som en trend nå.

Problem: Har en lang livsstid som gjør det utfordrenede å vedlikeholde gamle systemer kontra utviling av nye. Bedrifter må derfor vedlikeholde mange ulike konfigurasjoner, og vedlikeholde systemer er noe av det mest utfordrende siden det krever så mye tid. Derfor er det viktig å se på løsninger som tar til hensyn til dette. Utvikle abstrakte, high level design tme quality software. Viktig med arkitektur.

Problem er at de fleste foretrekker å levere noe i tide enn å lage noe bra.

Trenger en platform for å kunne vedlikeholde TG kontinuerlig, veilede, prioritere, håndtere, refactor.

.....

Most of the future computing systems will be embedde systems. Such systems will integrate both hardware and software components[6]. Embedded software are specialized hardware it runs on, which gives us some constraints when it comes to run-time, like memory usage, processessing power etc. With Internet of Things as a big trend now, these types of software has a big role today.

One of the problems ES faces is the

3.3 Virtualization of embedded systems

3.4 Configuration management

Configuration management er et disiplin for styring av av innhold, endringer, status på delt informasjon i et prosjekt. Det omfatter både prosesser og tekniske løsninger for å håndtere endringer og integriteten til prosjektet.F.eks hvis man utgir ny versjon av et produkt, må dokumentasjon også oppdateres. Config Management identifiserer hver komponent, og holder rede på alt som har blitt foreslått og godkjent endring fra dag 1 til slutt.

Software CM er en disiplin for kontrollering av programvaresystemer. Altså kontrollere utviklingen av store og komplekse programvarer. Noen eksempler på SCM er Git-Scm, SVN, RCS, Adele, ClearCase osv. Versjonskontroll er nøkkelen bak SCM. Følgende aspekter er med å definere CM ifølge IEEE: - Identification: Struktur av produktet, identifiserer komponenter og deres typer, gjør den unik og tilgjengelig på en måte. - Control: Kontrollerer release og og endringer av et produkt i løpet av produktets livsyklus ved å ha diverse kontroller/sjekk som sørger for konsistent produkt via "creation of a baseline product" - Status Accounting: Tar opp og rapporterer status til komponenter og evt endringer (forespørsler). Får også

statistikk om produktet. - Audit and Review: Validerer det komplette produktet og vedlikeholder konsistensen mellom komponenter ved å sørge for at produktet er en vel-definert kolleksjon av komponenter.

Kan utvides med disse tre definisjonene å: - Manufacture: Vedlikeholde konstruksjon og bygge produktet på en optimert måte. - Process management: Passe på organisasjonens personvern, prosedyrer og livssyklus modell. - Teamwork: Sjekke arbeidet og passe på et godt samarbeid, og passe på interaksjonen mellom flere brukere og produktet.

Når skal CM brukes? Det varierer. Noen velger å bruke CM system når produktet har gått gjennom utviklingsfasen og er klar for lansering/shipment. Andre velger å putte alt i CM ved oppstart av prosjektet. Begge har sine overheads. Man kan ta et valgt basert på overheads ved en endring. Er det mye manuell arbeid som å fylle ut diverse former, søke om godkjennelse osv vil man ofte plassere programverer under CM etter utvikling. Men hvis en forespørslen om en endring bruker lite tid og innsats fra utviklere, kan man velge å implementere tidlig. I teorien kan CM implementeres i alle stadiger i produktets livssyklus som opprettelse, utvikling, release, levering til kunde, bruk av produkt osv. Men ideelt sett bør et CM ha lite overhead som mulig, slik at software til CM implementeres så tidlig som mulig. Eksisterende CM systemer fokusterer dessverre på en viss fase i livsfasen, så brukere er begrenset av funksjonaliteten.

Ved å velge en robust SCM system gjør det oss mulige til å håndtere store og komplekse filmengder, støtter distribuert utvikling. En riktig kombinasjon av SCM system og "best practices" gjør det mulig for embedded development projects i å progressere raskt og effektivt.

Noen utfordringer med utvikling av embedded systemer er følgende: - Complex file sets o En embedded system består av flere komponenter, både hardware og software. Dette gjør systemet komplekst siden et slikt system kan ha mange varierte komponenter. Systemer kan også ha ulike varianter av komponenter til en spesifikk platform slik at man kan selge t produkt ved å endre ltit på krav. Å håndtere disse variantene er en stor utfordring. En annen utfordring er at produkter krever en kor-

rekt versjon av en komponent. Å sørge for at korrelasjonen mellom hver komponent og deres avhengige filer er vedlikeholdt er en utfordring det å. - Distributed teams o Komponenter kan utvikles i ulike steder i vår verden. Samtidig kan to teams to forskjjelige steder jobbe med samme komponent, spesielt når noe outsources. Slikt samarbeid krever at utviklere har adgang til hver andres arbeid. Utfordringen er at utviklere som jobber i hvert sitt sted (geografisk) holder seg synkronisert. - Management and versioning of intellectual property o Siden embedded systemer ofte tar I bruk tredje-parts teknologier, er det viktig at de utviklerne bak disse teknologiene oppdater og vedlikeholder arbeidet sitt. Disse oppdateringene må også være sporbare slik at prosjektet inneholder riktig, kompatibel og stabil versjon av teknologien. Utfordringen er å tillate disse utviklerne å sjekke inn contributions og spore endringer i det man har kontribuert. Velge man noe open source ervel dette ikke et problem?

Når det kommer til teknisk gjeld er det ikke alltid personen som har utviklet noe som tar ansvar, men kan en annen kan ta seg av den gjelda. Mange utviklere vedlikeholder ikke sin egen kode. Mange selskaper har også regler om at når et software er ferdig utviklet av de "beste" til å bli vedlikeholdt av de nest beste, som ofte kan få mindre betalt men har mye mer arbeid å gjøre. Ingen i din organisasjonen viser interesse for det, er brukerne som må betale for gjelda. Utviklere er belønnet for hvor raskt de implementerer enn langsiktig vedlikehold og kan ha fått seg et nytt prosjekt før gjelda er betalt. Få systemer har TODO eller FIXME kommentarer i kildekoden.

Til forskjell fra finansiell gjeld kan teknisk gjeld aldri betales tilbake i sitt fulle. Å betale TG kommer i en form av hvor lang tid det tar å fikse koden/problemet. Men det er ikke lett å vise hvilke gjeld som har høyest kost. Er interessen lavere enn hva det koster, er det ikke vits i å betale tilbake. Eksempel: Man har et system som trenger en oppgradering som kan koste 1 million. Man tar valget i å ikke oppgradere, og satse på at systemet fungerer. Det gjør det ikke, systemet går ned og firmaet taper felre millioner på å reparere systemet. Her kunne man spart penger på å oppgradere.

3.4.1 Software reuse

Software reuse is about 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.

3.4.2 Software development life cycles

3.4.3 Software evolution

3.4.4 Deployment

CHAPTER 4

RESEARCH METHOD

4.1 Section Title

Part III

Fourth part

CHAPTER 5

RESULTS

5.1 Section Title

5.1.1 SubSection Title

5.2 Section Title

5.2.1 SubSection Title

Part IV

Discussion part

CHAPTER 6

DISCUSSION

6.1 Yee

Part V

Last part

CHAPTER 7

CONCLUSION

Part VI

Appendix

Stimulate savant beef noodles corporation Legba realism convenience store table human wristwatch Kowloon sprawl futurity math. Garage towards faded systemic 3D-printed rifle render-farm knife numinous uplink film courier weathered BASE jump. Cartel voodoo god 3D-printed jeans euro-pop footage bomb beef noodles lights Tokyo crypto-futurity rifle bridge vinyl. Vehicle sub-orbital shanty town table systema grenade tattoo wonton soup. Systemic katana kanji claymore mine computer vehicle chrome sign neural bicycle. Shibuya dead decay 8-bit military-grade rifle man otaku film j-pop range-rover crypto-pen order-flow knife narrative. Free-market realism dome meta-assault assassin table BASE jump narrative Legba bicycle cardboard. Sunglasses smart-sign neon vehicle rebar carbon ablative city engine. Towards shanty town digital soul-delay RAF cardboard nano.

BIBLIOGRAPHY

[1] Ward Cunningham. "The WyCash Portfolio Management System". In: *SIGPLAN OOPS Mess.* 4.2 (Dec. 1992), pp. 29–30. ISSN: 1055-6400. DOI: 10.1145/157710.157715. URL: http://doi.acm.org/10.1145/157710.157715.

[2] Albert Einstein. "Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]". In: *Annalen der Physik* 322.10 (1905), pp. 891–921. DOI: http://dx.doi.org/10.1002/andp.19053221004.

[3] Israel Gat and John D. Heintz. "From Assessment to Reduction: How Cutter Consortium Helps Rein in Millions of Dollars in Technical Debt". In: *Proceedings of the 2Nd Workshop on Managing Technical Debt*. MTD '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 24–26. ISBN: 978-1-4503-0586-0. DOI: 10.1145/1985362.1985368. URL: http://doi.acm.org/10.1145/1985362.1985368.

[4] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Reading, Massachusetts: Addison-Wesley, 1993.

[5] Donald Knuth. *Knuth: Computers and Typesetting*. 1984. URL: http://www-cs-faculty.stanford.edu/~uno/abcde.html.

[6] W. Wolf and J. Madsen. "Embedded systems education for the future". In: *Proceedings of the IEEE* 88.1 (2000), pp. 23–30. ISSN: 0018-9219. DOI: 10.1109/5.811598.