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.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 Motivation and goals

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. We can see that most future computing systems will be embedded systems [3]. 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 [2]. 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. Companies must often recall their products. If they could catch these software defects earlier in teh 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 big factor in embedded systems as developers might not be available years after implementation.

Technical debt is a rising problem. It is estimated that the total debt in 2015 will be around 1 billion dollars. That is the double of the amount technical debt in 2010.

1.2 Research questions

The main question for this project is:

- Hva er teknisk gjeld? Klarer vi å lage en definisjon til metaforen?
- Hva blir teknisk gjeld tolget som?
- Hva slags synonymer finnes til TG?
- Hva slags typer TG finnes?
- Hvordan skiller teknisk gjeld seg ut fra finansiell gjeld?
- Hva slags teknikker finnes til håndtering av TG, og hvordan blir disse brukt?
- Hvordan er TG relatert til evolusjon og maintenance?
- Hvordan prioriteres hva som skal nedbetales først?
- Current state of TD in the systems?
- How much do you focus on technical debt? I hvilken grad fokuserer virksomheten på dette?
- How much time is spent on reducing technical debt?
- Pådra gjeld med vilje? Fikses det?
- How much focus there is on maintenance and evolution?
- When a debt should be refactored?

- Which debts are easy to fix which gives software quality boost?
- Which debts are most difficult to address, which debts are hard to fix which doesnt give much software quality boost?
- What kind of methologies are used to track/prioritize technical debt

1.3 Research method

This research will consist of a sequence of activities. I've chosen to follow the model of the research process defined in the book "Researching Information Systems Something". This model says that for each research question, one strategy is needed with one or more data generation methods. A dataanalysis will be done after generation in form of quantitative or qualitative in order to draw a conclusion. This is to answer the research questions.

First of all, a litterature review will be carried out in order to get familiar with the area of study. Following the litterature review, one or more research questions will be defined. A strategy will be choosed, in this case a case study/survey . I will use the interview form as data generation

1.4 Project structure

The rest of this report will be structured as followed:

- Chapter 2: State-of-art of technical debt, embedded systems, software evolution, software maintenance, software development life cycles, software reuse, configuration management, security.

- Chapter 3: This chapter will present our reseach context and research method.

- Chapter 4: Will present the result of this project.

- Chapter 5: Evaluate and discusses contributions and the results to this project

- Chapter 6: Concludes the report and provides points to future work.

CHAPTER 2 STATE-OF-THE-ART TECHNICAL DEBT

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 trading of design and quality in order to solve a problem in time. 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 [?]. 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, 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 [?].

Technical debt is issues within the software which hampers its maintenance, which leads to that organizational debt is issues preventing a company from running smoothly. 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.

Reasons for technical debt: - 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.

Good building, furniture, and compensation for executive staff isnt 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.

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 Why does it occur

Many thinks that technical debt is mostly related to code decide. 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.

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.

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 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. 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. 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.

CHAPTER 3

EMBEDDED SYSTEMS

3.1 Introduction

With the rapid evolution of electrical and software based systems, known as embedded systems, we see that most of the future computing systems will be embedded systems [3]. 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,

3.2 Embedded system software

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.

.... 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 korrekt 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?

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 maintenance and evolution

http://www.kantega.no/livslop/ http://swreflections.blogspot.no/2011/04/lientz-and-swanson-on-software.html

After the software is transferred to the users, software will enter maintenance mode. Software maintenance according to IEEE 1219 is defined as "modification of a software after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." however, the definition is poor defined term and little used in the industry. Most research today has been on the inital development phase. Configuration management has made contributions to software maintenance.

The term maintenance can also be appliec to the problem of keeping web pages and software applications up to date and consistent.

Lientz and Swanson did a survey back in the 1970s with 487. The survey where they categorized maintenance into four categories: - Adaptive: Changes in the software environment - Perfective: New requirements - Corrective: Fixing errors - Preventive: Preventing future problems According to van Vliet, the real maintenance activity corrective maintenance. However, the survey showed that nearly 75 % of the attendees spent their time on category 1 and 2. Nearly 22 percent on category 3 and 3 percent on category 4. Why didn't they spend more time on fixing bugs and preventing future erros? Most severe maintenance problems was caused by poor documentation, demand from users for changes, difficulty meeting schedulment, 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. How much has this changed since then? it looks like that new user requirements is the main problem for software evolution and maintenance.

Change: According to Viet, perfective consumes 50%, adaptive consumes %25, corrective 21 and 4 on

Software maintenance is important because it consumes a larg part of the life-cycle costs and that if not able to change software quickly to adapt to software environment means that business opportunities are lost.

A model called stage model basded on empirical observation was defined by Bennet and Rajlich. This model helps us analysing maintenance and evolution of software development. Maintenance was seen as a single activity after initial development, however, a staged model was introduced which split maintenance into five distinc stages. The reason is to separate maintenance into an evolution stage followd by servicing and phase out stages. To make this possible, both software architecture and software team knowledge needs to be well made. Architecture needs to be well defines as it will be used for the rest of the life of the program. Knowledge is abhout the knowledge og the app domain, requirements, algorithms, data formats, strength/weaknesses of program/architecture/environment.

Servicing stage is about small changes, usually patches and code changes. The program is not evolvable here.

Software evolution is a process that usually takes place when the initial development of a software project is done and was successfull. 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.

Final stages are phase out and close-down. System might be in production, but no service is undertaken. During close-down, software is disconnected.

An amplified of this staged model was defined as well, versioned stage model. The difference between this model and the simple stage model is that during evolution, versions of software are being made. All new functionality or changes are implemented in the future versions. If a version becomes outdated, it is replaced with a new version.

CHAPTER 4 RESEARCH METHOD

4.1 Section Title

Part I

Fourth part

CHAPTER 5
RESULTS

5.1 Section Title

5.1.1 SubSection Title

5.2 Section Title

5.2.1 SubSection Title

Part II

Discussion part

CHAPTER 6 DISCUSSION

6.1 Yee

Part III

Last part

CHAPTER 7
CONCLUSION

Part IV

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. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

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

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