CHALMERS

How to Manage Technical Debt in a Lean Startup

Master of Science Thesis in the Programme Software Engineering

HAMPUS NILSSON
LINUS PETERSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2013

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

How to Manage Technical Debt in a Lean Startup.

HAMPUS. NILSSON,
LINUS. PETERSSON,

© HAMPUS. NILSSON, October 2013.
© LINUS. PETERSSON, October 2013.

Examiner: MICHEL. CHAUDRON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2013

Abstract

Startups are becoming ever more prominent in today's world and the lean startup movement has shown a method to reach success through rapid development and prototyping. The effects this has on software quality and how it should be handled is a novel subject, the concept of technical debt has been known within the field Software Engineering for over a decade but there is very little research on how it is, or should be handled in startup contexts.

This study aims to fill this void through interviewing nine startups companies about their technical debt issues. Simultaneously the researchers developed an internet startup project and evaluated the effectiveness of methods and software tools that can be used to manage technical debt.

To address difficulties of discussing technical debt a new model for classifying debt named the Technical Debt Quadrant is presented. To solve the issue of managing technical debt in startups a list of concrete tips for managing technical debt and a matrix appellated the Debt Strategy Matrix that can be consulted in the different phases of a startup's life was developed. The validity of these new solutions will need to be further evaluated in future studies to assess their usefulness.

The new terms for referring to technical debt will be of use for both researchers and practitioners in the field of Software Engineering in the future. The strategies for managing technical debt can be used without the overhead associated with previous solutions by any startup to avoid long-term technical issues.

Acknowledgements

We would like to thank the following people for their support during the work with our master thesis.

Elma Delic, Henrik Lång and David Svensson at Alice.

Our supervisor Jan Bosch.

Our examiner Michel Chaudron.

Contents

Glossary **iv**

1 Introduction **1**

2 Background **2**
2.1 Lean Startup Methodology 2
2.1.1 Build-Measure-Learn 2
2.1.2 Minimum Viable Product 2
2.1.3 Metrics 3
2.1.4 Pivot 4
2.2 Technical Debt 4
2.2.1 Types of Debt 4
2.2.2 Identifying Debt 6
2.3 Tools and Methods 6
2.3.1 SQALE 6
2.3.2 Sonar 7
2.3.3 CAST AIP 7
2.3.4 Code Climate 8
2.4 Alice 8

3 Purpose **10**

4 Method **11**
4.1 Data Collection 11
4.1.1 Primary Case 11
4.1.2 Secondary Cases 11
4.1.3 Software Evaluation 13
4.2 Validity Threats 13

5 Result **15**
5.1 Evaluated Software 15
5.1.1 Sonar 15
5.1.2 CAST AIP 16
5.1.3 Code Climate 17
5.2 Primary Case 18
5.2.1 Process 18
5.2.2 Technical Debt 18
5.3 Interview Data 19
5.3.1 Jan Salvador van der Ven 19
5.3.2 Appello 20

5.3.3 Burt . . . 21
5.3.4 Duego . . . 23
5.3.5 NetClean . . . 24
5.3.6 PugglePay . . . 25
5.3.7 Recorded Future . . . 26
5.3.8 Shpare . . . 28
5.3.9 Trimbia . . . 29
5.4 Interviews Summary . . . 30

6 Discussion **35**
6.1 Analysis . . . 35
6.1.1 Technical debt problems identified . . . 35
6.1.2 Debt generating activities . . . 37
6.1.3 Methods in use . . . 38
6.2 Solution . . . 40
6.2.1 The Debt Quadrant . . . 40
6.2.2 Strategy . . . 43
6.2.3 Debt Strategy Matrix . . . 45
6.3 Validation . . . 47

7 Conclusion **50**

8 Future Work **51**

References **52**

Glossary

Code Smell	An issue in the code, such as excessive complexity, duplication, poor commenting or obfuscated code etc.
Cowboy Programming	Developers work without any formal process and assign their own tasks without much intervention from business developers.
Kanban	An agile software development methodology focused on just-in-time delivery.
Pivotal Tracker	An online task management board made for Scrum development.
SaaS	Software as a Service, a solution hosted online and delivered to users over the web rather than being installed on their own computers.
Scrum	An agile software development methodology.
Startup	In this thesis a startup is defined as "A startup is a human institution designed to create a new product or service under conditions of extreme uncertainty." (Ries, 2011, p. 27).
Trello	An online task management board.

1 Introduction

When beginning work on a new software based product, following best practices and spending time planning the architecture and design of the product may be time well spent, given there are resources available and a set of semi-final requirements. However, in a world where the model of the lean startup is becoming more common and the focus of the product can change many times, both late and early in the development process, time spent planning can be time wasted.

Eric Ries introduced the lean methodology in 2008 (Ries, 2008). Lean focuses on performing the minimal amount of work on the product in order to sell, and design after the customer's wishes instead of spending time implementing features that you don't know are relevant. This is a valuable proposition for smaller development houses and startup companies where the economic resources are limited (Moogk, 2012; Ries, 2008).

Merging these two concepts it's obvious that forgoing planning, meticulous testing and adherence of coding standards etc. can be a reality when developing using lean. This will lead to an amassment of technical debt that will grow more unwieldy with time (Poort, 2011). This report attempts to propose a solution to this problem by offering guidelines on how to manage the debt early on and avoid ending up in a situation where you need to put the brakes on development in order to rectify earlier mistakes.

The thesis first addresses the background of the lean startup methodology, defines technical debt and presents previous work conducted within the domain. It also presents a few prominent software solutions and methods that can be used to manage technical debt. A brief evaluation of the software solutions is presented in a subsequent chapter. A case study conducted at a company following the lean startup methodology is presented containing experiences about how technical debt behaves in that type of setting. The subject is further explored using interviews conducted at various startup companies to investigate how it is currently handled in practice. Further, a new model of how to characterize technical debt is presented which aids in communicating about technical debt by defining a common terminology. Finally, a general strategy guide for how to manage technical debt is presented followed by a more specific model where the management strategies in the different phases of a startup's life are defined and categorized.

2 Background

This chapter details what technical debt and the lean startup method is and a number of different methods available for managing technical debt and their strengths and weaknesses.

2.1 Lean Startup Methodology

The Lean Startup methodology was first introduced by Eric Ries in a blog post (Ries, 2008) which later lead to the book *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses* (Ries, 2011). The methodology emerged from Eric's experience working with successful and failing startups. The methodology is a mix of agile software development, customer development and lean practices using frequent customer interaction and short iterations. It focuses on how to utilize the time as efficiently as possible and how to quickly learn about the customers and their needs.

2.1.1 Build-Measure-Learn

The process is focused around validated learning and what Ries calls the *Build-Measure-Learn* feedback loop, visible in figure 2.1, to conduct experiments (Ries, 2011). It starts with the *Build* stage where some kind of product is built. It may be as a working product, an interactive mockup, a paper mockup or whatever fits the bill. This item is referred to as the artifact, and is handed off to a customer and the *Measure* step of the feedback loop starts. The response from the customer is measured using both quantitative and qualitative techniques. The data from the measuring is used in the *Learn* step of the feedback loop where the idea(s) can be validated or discarded. The process is then restarted with new ideas and the *Build-Measure-Learn* loop continues in an iterative fashion all through the project.

2.1.2 Minimum Viable Product

A Minimum Viable Product (MVP) is a very early version of the product that is built for the purpose of learning about potential customers and validate the business idea. It is built with as little effort as possible just to be put in the hands of customers so the team can begin to receive feedback. It doesn't have to be a fully functional product that does everything in a perfect manner. (Moogk, 2012; Ries, 2011)

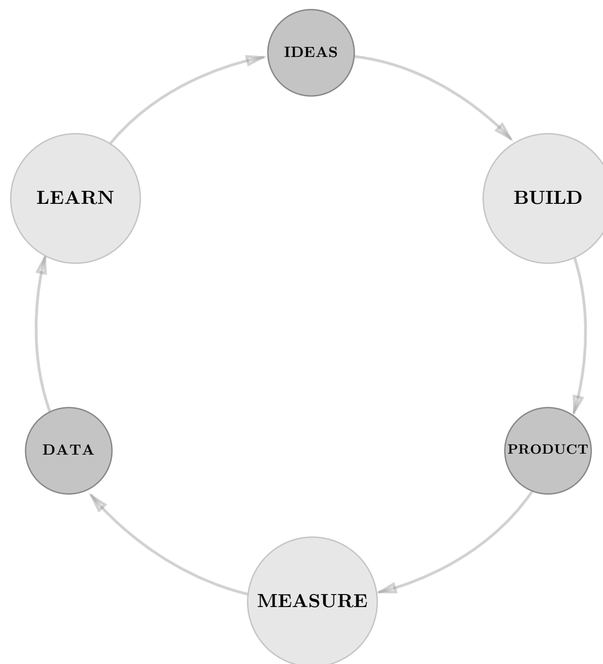

Figure 2.1: Build-Measure-Learn feedback loop.

For instance, when the online shoe store *Zappo* started, its founder Nick Swinmurn built a very simple online web shop and when a customer ordered a pair of shoes, he went to the local shoe store, bought the shoes and sent them to his customer (Ries, 2011). Instead of spending a lot of time and money to build a fully functional product he created this simple MVP with very little effort. Using this he could, with a very low risk, validate his business idea in the real world before taking the next step. That's mainly what the MVP is about.

2.1.3 Metrics

To know whether the product development is actually leading to a real progress appropriate measuring needs to be conducted. It is important to select the right metrics and stay away from, so called, *vanity metrics*. Vanity metrics are metrics that "give the rosiest possible picture" (Ries, 2011, p.182) and therefore may be very misleading. These metrics differs between companies depending on their type of business. Instead, *actionable metrics* are more appropriate. An example of an actionable metric is *A/B testing* (also known as split testing). When using A/B testing a new feature is rolled out to a group of customers and their behavior is measured (Burke, 2005). The data generated from the test can then be used to make a decision about what action to take next and help you confirm or refute your hypothesis. Of course, the A/B testing is only one of many actionable metrics a startup may use. (Kohavi et al., 2007; Ries, 2011)

2.1.4 Pivot

During the project, as the team learns more about the customers' needs and in what direction the project should be heading the product is continually being evolved and optimized. In some cases the project may be on the wrong track and needs to make a slight (or sharp) turn and change its strategy. This change of strategy is called a *Pivot*. (Ries, 2011)

Pivots are a polarizing moment for a startup, pivoting allows you to tune the product to what the customers actually wanted and rid yourself of the parts of your initial idea that turned out to be wrong. The difficulty of pivoting is to decide when it is appropriate and doing it at the right time to make use of your strategic resources in the best possible way. (Ries, 2011; Bingham et al., 2011)

2.2 Technical Debt

The concept of technical debt was introduced by Ward Cunningham back in 1992, and refers the tendency for code that continue to be worked on to become more and more ill-fitted for the new requirements it's made to fit (Cunningham, 1992). As more and more functionality is bolted on to the system, metaphorical "debt" is collected in the form of increasing complexity. This debt needs to be repaid later on by taking the time to rewrite the original system to better fit with the current demands, in the process that is now known as refactoring.

Small amounts of technical debt can lead to flexibility in decisions, to quote Cunningham (1992) "A little debt speeds development so long as it is paid back promptly with a rewrite.". It is however important to still keep track of the debt, as it will incur overhead on development over time on development (Curtis et al., 2012; Kruchten et al., 2012).

2.2.1 Types of Debt

There are many definitions of the types of technical debt that exists. Kruchten et al. (2012) divides the debt in two prominent categories, either visible - which is debt that can be identified by people who are not software developers - or invisible - which is debt that can only be identified by software developers.

Examples of visible debt is new features and defects, which are visible to managerial staff who can easily see these objects in the backlog or similar tools. In essence this encompasses the external quality attributes initially defined in ISO 9126 (2001) and later supplemented by ISO 25010 (2011).

Invisible debt can be due to many factors and it is very difficult to put exact measures on the size of it. This type of debt includes things like bad architecture, code complexity,

bad technology choices and code hygiene issues like lack of coding standards. Many of the concepts here can be linked to internal quality metrics defined in ISO 25010 (2011), that is, things that are not visible at runtime but only when looking at the source and architecture of the application. Some external qualities can be linked here as well however, like efficiency issues.

McConnel (2007) talks about technical debt from another aspect. Instead of dividing the debt based on its characteristics he defines two basic categories based on how they are incurred. The two groups includes debt that is incurred intentionally or unintentionally. The unintentional group includes debt that comes from doing a poor job, for instance when a junior programmer writes bad code due to lack of knowledge. Another example is when a company acquires another company and technical debt is found after the acquisition. Intentional debt is debt taken on consciously for strategic reasons. For instance when a company must release a product before the debt can be cleaned up.

McConnel further divides intentional debt into short-term and long-term debt. Short-term debt is incurred reactively and paid off frequently while long-term debt is incurred proactively and paid off over the course of several years (McConnel, 2007).

Fowler (2009) has yet another categorization of the different debt types, in what he calls the “Technical Debt Quadrant”. As seen in figure 2.2, Fowler groups the debt into four categories: reckless deliberate/inadvertent debt and prudent deliberate/inadvertent debt. The reckless debt includes debt which is taken on due to ignorance or lack of knowledge of good design and best practices in a deliberate or inadvertent way. The prudent deliberate debt on the other hand, is when you take on debt consciously for a good reason. Fowler (2009) points out that the prudent inadvertent debt is a bit odd but argues that it is inevitable for teams that are excellent designers.

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

Figure 2.2: Fowler’s Technical Debt Quadrant.

2.2.2 Identifying Debt

One way of keeping track of the debt is by making use of code analysis tools. These can identify consistency errors, lack of commenting and documentation and other minor transgressions. Curtis et al. (2012) introduces a very mechanical way of identifying debt by using the CAST software suite, described in detail in section 2.3.3. This is very similar to the concept of invisible debt that Kruchten et al. (2012) distinguished and the goal is to make it understandable to non-software developers.

There is however no code analysis tool that can identify architectural missteps that can require redesigning major parts of the system, or a choice to rely on a technology that over time becomes obsolete or marginalized due to changes in the environment (Kruchten et al., 2012). Identifying these threats relies on the expertise of the developers and their ability to convey the risks of choices to the stakeholders. With experience all developers get an understanding of what code is good code and what code is bad code and learns to approach development in a way as to avoid much of it (Hunt and Thomas, 1999).

2.3 Tools and Methods

2.3.1 SQALE

SQALE (Software Quality Assessment based on Lifecycle Expectations) is a method of assessing the technical debt in a software project. It is based on tools that analyze the source code of the project, looking at different types of errors such as mismatched indentation, different naming conventions and more. Each of the errors is assigned a score based on how much work it would be to fix that error. The analysis then gives a total sum of technical debt for the entire project. (SonarSource, 2013*b*)

Much of the SQALE method is grounded in visualizing the amount of debt present. As such the analysis should be conducted on a daily basis if not more often, and illustrating it on a dashboard or similar is another boon to show engineers what effect the code they are writing is having on the shared codebase. (Letouzey and Ilkiewicz, 2012)

SQALE was designed to be generic and applicable to any programming language or development methodology. The method itself is published under an open source license and is entirely royalty free. An open source framework implementing it named Sonar is available, which also exists in commercialized versions. This framework is an official implementation of the SQALE methodology aimed to ease the communication of the importance of technical debt between managers and programmers. (Freddy Mallet, 2010)

2.3.2 Sonar

Sonar is an open source platform that inspects and analyzes your code quality. It is owned and maintained by SonarSource S.A in Switzerland. The software is cross platform and written in Java. Out of the box Sonar supports code inspection for Java (SonarSource, 2013*b*) but support for other languages can easily be added through plugins. There are plugins available for a number of popular languages such as COBOL, C, C#, C++, PHP, Python, VB.NET and more. Some are free community supported plugins and some are commercial.

Sonar includes the ability to create dashboards that can be customized for different uses. The dashboards are configurable using widgets that can be rearranged through drag and drop. The widgets can contain anything from pure text to advanced and interactive charts. Sonar comes with a number of default widgets and new widgets can be added by plugins that need some kind of specialized visualizations.

The software comes in three different flavors; Open source, Professional and Enterprise. The open source version is free and doesn't include the commercial plugins or support. The main differences between the other two packages are the number of included plugins, level of support and training. The professional version starts at €12 500 and the enterprise version starts at €50 000 (SonarSource, 2013*a*).

Sonar has a free plugin that calculates the technical debt in your codebase (SonarSource, 2012). However, it is very limited. SonarSource has instead created a commercial plugin that is much more advanced and feature complete (SonarSource, 2013*c*). The plugin is a full implementation of the SQALE methodology (SQALE.org, 2013) and it uses the ISO 9126 quality model with characteristics and sub-characteristics. It can show all kinds of metrics such as total remediation cost, remediation cost per file, remediation cost per characteristic, remediation cost grouped by severity, overall SQALE rating, SQALE rating per component and more. It presents all the data visually on a dashboard using configurable widgets. The commercial plugins comes with a price tag of €2 700 per year.

2.3.3 CAST AIP

The CAST Application Intelligence Platform (AIP) is a complete solution for analyzing and measure software quality. The application is very comprehensive and focuses mainly on complex enterprise systems. It helps companies to identify and mitigate IT-risks, analyze the architecture and code quality, monitor team performance, reduce technical debt and a lot more. It has support for a lot of different programming languages, databases and frameworks such as J2EE, Cobol and .NET.

The tools included in the CAST AIP can quantify the technical debt that has incurred in the project and help the team to proactively manage and reduce it over time. CAST AIP measures five different types of technical debt, or health factors as they call it. The

health factors are: Changeability, Transferability, Security, Performance and Robustness (Cast Software, 2013). CAST AIP can also benchmark and compare the quality and performance with other applications through the appmarq repository provided by CAST. This repository contains statistical data, trends and best practices from thousands of other applications. The data in the repository has been extracted from applications in different industries.

CAST AIP is focused around transparency and visualization of quality, trends, issues etc. This visualization is done via web based dashboards that can be configured to show relevant information based on its intended audience. These dashboards gives the managers and developers a common place to follow the application's evolution over time.

2.3.4 Code Climate

Code Climate is a SaaS solution for monitoring technical debt that is made specifically for monitoring the code quality of Ruby applications. The service makes use of multiple analysis tools available for Ruby, like metric_fu and Flog and composes them to a single output view for the developer.

The software works from a website and only requires checkout rights to a git repository to function. It monitors the codebase continuously, assigning graded score in the range from F (worst) to A (best) to classes and methods across the project, pointing out "Code Smells" such as cyclomatic complexity concerns, code duplication and there is also the option of finding security related issues. Developers are notified by email weekly with new issues arising in the code, or things that were improved. This provides continues feedback on the status of health of the product and the intent is to keep developers motivated to work with code quality issues.

2.4 Alice

Alice is a small startup located in Gothenburg, Sweden, that develops software used for managing health and safety in companies. The company begun its software development in 2013 in a business incubator setting at Chalmers University of Technology.

Alice's first product was an incident management system where employees can report incidents that happen at the workplace, such as accidents, deficiencies in the work environment etc. The reported events are then tracked and handled by the responsible manager(s) through a web interface. The company follows the Lean Startup methodology created by Eric Ries and their first MVP was developed during the spring of 2013.

Alice was studied for a few months during the development of their first MVP. Since it is a startup that follows the lean startup methodology it fits well with the research focus of

this thesis. The product rapidly changed during development and changed direction and target customers towards the end of the research project.

3 Purpose

The purpose of this study is to explore and better understand the concept of technical debt in a lean startup context. In general technical debt is seen as a detriment to the software development process, but this may not be the case for a startup using the lean startup methodology as quick solutions can be a resource to be harnessed (Ries, 2009).

The problem with lean startups is that the rapid prototyping and development used quickly generates a high level of debt (Poort, 2011), perhaps this debt can be used as a leverage though, and what methods would be appropriate to manage it?

By reviewing literature in the area of startups and technical debt and joining it with input from industry practitioners we identified some key areas that are lacking in the handling of technical debt:

1. **There is no good way of categorizing/classifying technical debt** – Interviews and literature offers little in the way of classifying technical debt on a level between breaking it into its smallest constituent parts and the types introduced by Fowler (2004) that is aimed at debt acquisition, not inventory.
2. **Methods for handling tech debt are focused on mature projects** – The methodologies available for handling technical debt are all aimed at projects that either use more traditional management styles and planning than startups do. There is a noticeable lack of how technical debt should be handled and reasoned about in a lean startup.
3. **Existing methods does not give clear guidance about when and how to handle different types of debt** – Finally the methods that are available are all on either of a very mechanical focus, centered around code analysis or defined through fuzzy, general guidelines on how experienced programmers work with architecture. Our aim is to define more concrete guidelines and methods on how debt should be handled in startups.

The goal of this study is to explore these areas in depth and their place and relationship in the lean startup model. Possible solutions will be developed and presented but not validated through practice due to the limited scope of the project.

4 Method

4.1 Data Collection

The study was performed by collecting data both through a primary case as well as from a series of interviews with startup companies in the software business. In the primary case a new software product was developed from the prototype stage to launch and a short period afterwards and data was sampled using a soft systems research approach (N. Denzin, 2000), were the researchers participated in the development.

Based on literature, experience gained from the primary case and the real life knowledge collected from the interviews a model and a set of guidelines were defined.

4.1.1 Primary Case

The primary case was conducted at the startup company Alice in Gothenburg. The authors participated in the development of a new software product for the business to business market. The product development applied the lean methodology, developing an MVP in tandem with trying to sell the product to customers and integrating feedback from customers very early in the development process.

The primary case was used to get a first-hand experience of how a lean startup works. A number of issues were investigated such as what challenges arise due to technical debt, when in the development process do you have to worry about the debt etc. The primary case was also used to evaluate a number of different software solutions that were available to manage technical debt, revealing their strengths and weaknesses in relation to the context of a startup. This was done by looking at factors such as setup overhead, how easy it is to use the data the tool provides, how relevant the data is and how difficult it is to integrate the feedback the tool provides into the product.

4.1.2 Secondary Cases

The secondary cases were used as complements to the primary case to support the findings. The interviews focused around questions about how the company encountered technical debt, what they did to handle it initially and how they are handling it ongoing throughout development, and also to collect data on how older startups have handled their technical debt problems over time.

Interviews

The companies to interview were picked from a list of startups in Gothenburg. The reason for them all being in Gothenburg was so they could easily be interviewed in person rather than by phone or some other form of less direct communication. From the list, the companies that best seemed to fit the study were selected.

The interviews were conducted in a semistructured fashion. A semistructured approach has the benefit of enabling the interviewer to improvise and probe deeper into a subject during the interview (N. Denzin, 2000; Hove and Anda, 2005). Most of the interview questions posed were formulated in an open fashion, this allows the interviewee to express his view on the subject in depth (Wohlin et al., 2012; Hove and Anda, 2005). Before the interviews were conducted it was not clear how the companies worked with technical debt, how they defined it, or if they even knew what it was at all. The semistructured approach made it possible for the interviewers to clarify and answer any questions the interviewees had and avoid misunderstandings and misinterpretations. Also, interviews are possible to change along the way as the knowledge about the subject increases. This is much harder when you, for instance, use surveys (N. Denzin, 2000). The interviewees were chosen by the companies but the type of person needed for the interview were given by the interviewers. This was important since the people that were interviewed had to have both a certain level of technical knowledge and been with the company since its conception or close to it.

Before any interviews were conducted a number of questions were created and validated by performing a mock interview to find weaknesses in phrasing and structure. Most of the questions were used in all interviews, but a couple of questions were added or refined for the last few interviews. When conducting an interview the questions were used as guidance and appropriate follow-up questions were asked to probe deeper into interesting subjects. A few example questions and answers can be found in table 4.1. The same two interviewers were present at all interviews, both asking questions and taking notes. One interviewer took more detailed notes while the other noted the most important things and focused on listening. This made it easier to avoid misunderstandings and loss of important data, and has been shown to make the interviewee more talkative (Hove and Anda, 2005). After all the relevant questions had been asked and if there was time the interviewee was also indulged on the results of the study so far and asked for their opinion on the conclusions reached at that point.

After the interviews had been conducted, the collected data was analyzed using tabulation (Wohlin et al., 2012). The data was arranged in tables containing the most relevant characteristics, see table 5.1 and table 5.2, and conclusions were drawn from analyzing it.

Table 4.1: Examples of interview questions and answers.

Question	Answer
Do you think about technical debt?	Yes, but it is not discussed in those terms.
Is technical debt a problem for you?	Yes, there are issues with the reliability due to missing tests.
How do you deal with technical debt over time?	We continuously refactor code around the points where we work to keep the codebase in good shape.
Do you think the technical debt will become a problem in the future?	Yes, especially if developers leave the company.

4.1.3 Software Evaluation

During the project several different software tools for monitoring technical debt and other software metrics were evaluated. As the primary case development was only done using Ruby on Rails, it was not possible to evaluate tools based on other languages in the primary case. For these solutions, open-source projects were downloaded and analyzed instead.

For the different products, the following aspects were in focus:

- The price of the solution, would it be feasible for a small development shop to spend the money required to make use of the product.
- The scope of the solution, how much setup was required to begin using the software. Excessive setup requirements would seriously impair a lean startup on focusing on the product, which is orthogonal to the lean process.
- The maintenance requirements, when using the solution did it require continuous manual labor to be useful and if so how much.
- Was the feedback the tool gave useful to the developers, did it give excessive amounts of information, or too little. Were false positives common and could they easily be avoided etc.

4.2 Validity Threats

This section considers the threats to the validity of the study performed according to the four aspects of validity threats as established by Wohlin et al. (2012).

Construct Validity. Construct validity is the threat of the researchers view of the subject being not properly investigated by the tools selected for the study (Wohlin et al., 2012). As the sampling tool used in this study was interviews, the concept of technical debt was discussed prior to the interview to make sure the interviewee had the

same understanding of the subject. Also open-ended questions were an important part of making sure that the subject was allowed to express tangential information on the subject and explain their viewpoints thoroughly (Hove and Anda, 2005).

Internal Validity. Internal validity is concerned with how the treatment is linked to the outcome and if the researcher has missed any other factors besides the investigated when analyzing the results. The greatest internal threat is selection bias, which is always a threat when subjects are not chosen at random.

Threats related to instrumentation was avoided by using a proven interview technique (N. Denzin, 2000; Hove and Anda, 2005) and allowing the companies to select the person to be interviewed (increasing the randomness of the subjects).

In the primary case action research was used as a research method. Since the researchers participated in, and influenced, the process and studied it at the same time, there are risks of data collector bias and data collector characteristics (Onwuegbuzie, 2000). Data collector bias is when the researcher(s) favor a group or result and therefore, consciously or unconsciously, skews the data in that direction. Data collector characteristics concerns the characteristics of the people conducting the data collection. Characteristics such as age, gender, culture etc. may influence the type of data that is collected (Onwuegbuzie, 2000).

External Validity. The external validity is the ability to generalize the findings to the general population. In this report this would be if the findings are valid for startups beyond the investigated. As this is a qualitative study it cannot be identically replicated at a later date, instead the intention is to study and explain the underlying phenomena and patterns found. The greatest threat to the generalizability of the study is the fact that all the companies interviewed where located in the Gothenburg area in Sweden, and as such the Swedish work culture may influence the findings. This is mitigated in part by also interviewing a startup coach in the Netherlands.

The results from the primary case may have a low generalizability due to the research method used. Qualitative research is known to have a low generalizability, mainly because of the low sample sizes which yields a low statistical significance. This is mitigated by the findings in the secondary cases, i.e. the interviews.

Reliability. Reliability concerns the dependency between the data and the researchers conducting the study (Wohlin et al., 2012). E.g. would the study yield the same results if other researchers tried to replicate it.

Since the interviews were conducted in a semi-structured fashion when interviewing the companies, questions and discussions that were not planned beforehand came up. All discussions and interview questions may not be reflected in the report. If other researchers would conduct the interviews at other companies, it is probable that the same questions would not come up, yielding slightly differing results.

5 Result

The following chapter presents the results of the study, including evaluated software, a summary of the primary case and company interviews.

5.1 Evaluated Software

During the case study, a multitude of software for analyzing code quality and assisting with software development were evaluated.

5.1.1 Sonar

Sonar is a very comprehensive software solution with a lot of features, settings and customizations. Unfortunately, it is relatively expensive if you require the commercial plugins. For instance, if your project is written in VB.NET you need the commercial plugins that cost about €6 000 per year. This may not be a huge cost for a big enterprise company but for a startup with very scarce resources it's very expensive. On the other hand, if you can get by with the free plugins and the free version of Sonar this is not an issue. However, the team still needs to put in time to install, configure and integrate the software into the workflow to make full use of it. This might not take a long time for an experienced user but for someone who has never used Sonar it may be a bit overwhelming and hard to do properly.

In the primary case for this study the Ruby language was used. Sonar does not have official support for Ruby so it could not be used directly in this project. There is an open source plugin developed by PICA Group (2013) for the Ruby programming language, however it is inadequate by supporting only a tiny subset of the metrics required for a useful evaluation of code quality.

To evaluate Sonar a project written in a language supported by Sonar was analyzed instead. The Storm project (Storm, 2013), written mostly in Java, was chosen. The installation includes downloading the source code, configuring the database and launching Sonar from the command line. To run the analysis a number of clients are available. For this evaluation the default Sonar Runner was used, but there are clients available for integration with CI systems such as Jenkins, Hudson, Bamboo and more.

Sonar runs as a web server and most of the configuration can be done directly in the web interface. There are a huge amount of options where you can configure quality profiles, encryption settings, dashboards, users and more. The configuration section is a bit overwhelming and was left as default for this evaluation. The SQALE v.1.7 plugin (SonarSource, 2013*c*) was installed for technical debt analysis.

There were a few issues where the analysis tool could not parse the source code so some files had to be excluded manually from the analysis. When the analysis completed a dashboard with a few widgets showing information about technical debt, violations of coding standards, SQALE rating and more was configured, as shown in figure 5.1. The dashboard shows high level information but you can drill down to see exactly what code is causing the issues.

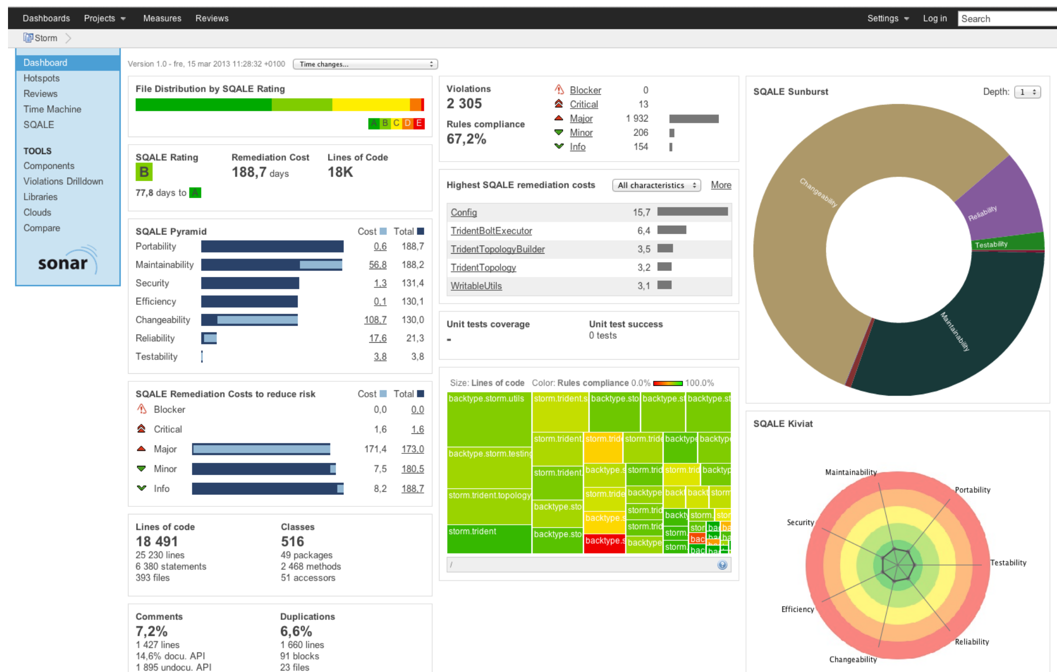

Figure 5.1: Dashboard in Sonar.

When everything is configured and integrated into your process Sonar works quite well. However, it is important to configure the dashboards in a way so it does not get messy and overloaded with information. The cost of setting up Sonar and integrate it into your workflow will differ a lot depending on previous experience and what plugins you need. If you are a startup don't have any prior experience with the software and need expensive commercial plugins to make use of the tool, it is probably not worth it.

5.1.2 CAST AIP

CAST Software are mostly focused on big enterprises and corporations. For this study the authors were not given any access to Cast AIP to make a more thorough evaluation.

5.1.3 Code Climate

Code Climate is much simpler than its competitors. However, it is much easier to grasp and quicker to get started with. It does not require any setup more than checkout rights to your git repository. The pricing is low and starts at $34USD/month for 2 users (excluding the security monitor), $74USD/month for 8 users (including the security monitor) and up to $399USD for 32 users (Code Climate, 2013). Since it is so easy to get started with, integrate into your process in conjunction with its low pricing it fits great in a startup.

The interface is clean and simple. It has a dashboard with a feed where all changes are listed chronologically, it displays hotspots and a chart of your classes ratings, seen in figure 5.2. It is easy to navigate through the different areas and find where the issues are in your project. Also, each week an email is sent out with a summary of the past week's changes. The summary includes both new issues that has cropped up during the week as well as improvements. This summary makes sure that you don't forget about the system and brings awareness to the team about the health of your application.

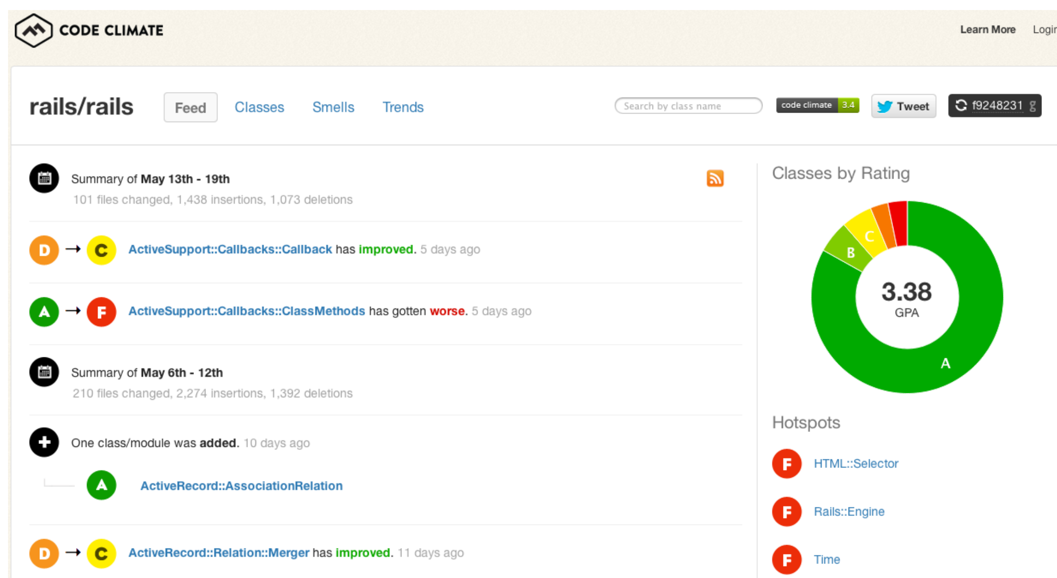

Figure 5.2: Dashboard in Code Climate.

However, Code Climate lacks in many other areas. It does not support any other languages than Ruby and it only performs analysis on pure ruby code files (not templates). It can not analyze test coverage and the issues it brings up, while most often valid, can be hard to pinpoint as the tool does not always tell exactly what or where the issues are. For instance, it may state that a specific class definition is too complex, but not exactly what part of it or how to mitigate it.

In the security monitor it is possible to mark an issue as a "false positive" which ignores it in the future. This is not possible in the code quality section which means that if you don't agree with Code Climate about an issue you cannot manually ignore it. If you have a lot of disagreement the issues you care about may get lost among the ones you don't care about.

5.2 Primary Case

Alice is a startup based in Gothenburg, Sweden, which develops software that companies can use to manage health and safety. The team consists of five people; three business developers and two software developers. The company started out as a master thesis project at Chalmers University of Technology by the business developers and after the product was decided, the developers were brought in to create an MVP. More information about Alice can be found in section 2.4.

5.2.1 Process

Alice works in an agile fashion and follows the Lean Startup methodology. Since the team is quite small and everyone is sitting close together it makes communication a breeze and formal meetings mostly superfluous. The company used paper mockups as well as HTML mockups before the development of the MVP started. To keep track of the features to be implemented the simple online task board Trello is used. Test-driven development is not used but tests are written for most features, especially for security-related and business critical areas of the application. Alice exercised pair programming in the beginning but now it's used rarely since there are only two developers and one focuses on the web application while the other focuses on the mobile application.

5.2.2 Technical Debt

They do not believe that the technical debt is a problem yet since their MVP is still being developed. They do believe it may become a problem in the future though, especially if the company decides to pivot or the developers are replaced. Alice believes that the most important debt they have is in lack of tests and documentation, architecturally the design is sound so far.

Alice does not have any formal coding standards defined but simply follows Ruby's best practices. They do have some informal guidelines though, regarding method complexity, commenting etc. Manual code reviews are not used in any formal way so far. The codebase is still quite small so the developers review each others' contributions continually as work is done. Also, they have not felt it necessary to create any external documentation

for their application. Instead they try to write clear and self-documenting code, add comments where appropriate and use the commit logs.

The team use Code Climate (read more about Code Climate in section 5.1.3) to keep track of the code quality, though not focusing on optimizing it yet. They check it continually to make sure the quality does not get out of hand but plan to use it more when their MVP has stabilized.

5.3 Interview Data

5.3.1 Jan Salvador van der Ven

Jan Salvador van der Ven is a PhD student at the University of Groningen and an agile enthusiast. He has an academic background but also experience from working in the industry. He has experience from working as team lead, Scrum Master and Agile Coach. Currently he works as an Agile Coach at the companies Factlink and Crop-R.

Process

In the companies where Jan has been involved the amount of planning varied a lot. Some companies planned for several months before starting the development and others spent a week planning. However, some level of planning was always present.

Most of the companies had some kind of beta or pilot version to test their idea on potential customers. They used short iterations and actually built a working prototype. The prototypes are very often reused and evolved to become the end product. Paper mockups was rarely used other than as a design tool.

The companies were all using agile development processes involving short sprints in small teams. They all write some tests, though the amount varies between different companies. A commonality were that they all wrote unit tests but had issues with integration tests.

Pair programming was used occasionally but only when the developers decided to pair up on a complex problem. From his experience a lot of agile propagators wants to see pair programming used all the time but in reality that does not always work.

Technical Debt

In the companies that Jan were involved some used software to keep track of the code quality. However, no actions were actually based on this data. Instead a senior developer reviews the code that is added by looking through commit logs. An issue with this is

when the senior developer is too busy all code will not be reviewed. Through these reviews the senior developer can hopefully discover when technical debt is added or where there are risks for it to arise.

Debt was most often generated through quick fixes such as skipped tests or changing the product to fit a particular customer's demand, something that later had to be generalized to be useful to the customer base or re-evaluated. These sorts of quick fixes he finds are are more commonly the source of bugs, while architectural issues slows down development.

Jan argued that if you avoid all the debt and do everything correctly from the beginning you will become slow and lose some of your agility. On the other hand, as the debt grows you will also loose speed and agility so a trade-off has to be made. In a startup with scarce resources you often cannot afford to do everything right from the beginning. You need to put the product on the market quick and later on, when you have paying customers, you can be more relaxed and implement things properly.

5.3.2 Appello

Appello is a company working on developing navigational mapping solutions targeted towards mobile devices. Their main customers are telephone operators who license their maps and brand and market them themselves. The initial prototype begun development in 2004 in the developers spare-time and they launched in 2006 and has since gradually expanded their business.

Process

Appello's technical team consists of 8 developers, they use Scrum with three week iterations, and have one rogue developer assigned to maintenance that uses a Kanban process for bug fixing etc. They feel this division is a great way of managing maintenance. They have also discovered with time that to use Scrum effectively the entire organization, not just the development part, needs to be tuned and educated of the process, for example so that the sales personnel don't sell features that won't fit into the current sprint.

As Appello is one of the older companies interviewed, the difference in how they worked initially versus later on in the development cycle is an important subject. For the first two years they used a cowboy programming style where there was no defined process, but when Scrum started becoming popular in 2008 they switched to it and have stuck with it since.

They do have some tests on the server-side of the software but don't find them very useful. They result in many false-positives and require lots of effort to maintain and keep up to date as the product evolves. While there are no formal demands for documentation they try to keep all code documented.

They also make use of a wiki to document tweaks made for different customers, this is very useful when the system breaks in the middle of the night and only the on-call developer is available as he or she can easily overlook what is different from the standard solution in the system.

Technical Debt

Because it is very difficult if not impossible to update deployed versions of the software installed on cellphones, and there are still devices with the 2006 and 2007 versions of the software in use. This means that they need to retain backwards compatibility for a very long time, forcing them to keep around lots of old code for communicating using the old protocols. They do not however spend any time maintaining this so it does not introduce significant overhead even if it does constitute lots of technical debt. In contrast, for the server side Appello does have a simple deployment environment, here they find that the ability to quickly update the backend and resolve issues lessens the burden of any technical debt and quick fixes as you can quickly patch them.

They are often forced to compromise on solutions to meet customer deadlines, something they find always gives backlash later in the process. They find that often when you are constrained on time and make quick fixes, you do not have time to document them either, and when they are later encountered after a few months or a few years it's harder to fix because of this.

They do make use of static analysis tools to keep errors down, this was initially done due to customer demand to get a report of the codebase error rate. But they have found the tools very useful in catching latent bugs in the codebase and make regular use of it since its introduction.

If they had the opportunity to start over, they would be much more adherent to standards for protocols. Using standardized communication makes it easier to plug different systems together in ways that was not conceived of initially (for example, for integration testing). They would also like to have made the system more modular from the beginning, for the same reason.

5.3.3 Burt

Burt is a company active in the marketing business, delivering detailed statistics and analysis to online ad publicists on a massive scale. Their core business is joining analytics from websites (page views, activity, ...) with information from the business systems such as income. They were initially targeted towards advertising agencies, and spent 9 months developing the initial product in close communication with their partners. Their initial hope was to help the advertising agencies make better ads, but after a year they realized their product was better suited to help the ad publisher's needs instead, and shifted focus through a minor pivot.

Process

Burt works without any formal process but still in an Agile way, labeled by them as "Ad-hoc Agile". They describe it as "Kanban"-like. They have tried to do Scrum at times but find the process is not compatible with their lean development methodology. Lean requires you to present alternatives quickly and iterate only on what works, which makes it difficult to have any meaningful length iterations with set goals in mind. They believe the reason they have the ability to work without any formal process is thanks to their small team size of about 10 developers.

They have the ambition to test all their code, but all team members do not see it as a necessary aspect of development. They don't feel that forcing these team members to write tests is productive but rather have it that each developer realizes the benefits on their own. As for documentation, there is a difference of opinion in the team but the interviewed persons viewpoint was that code comments are a sign that the code is not understandable enough. Comments are also easily forgotten while the code changes and end up being out-of-date over time, making them a obstacle rather than an aid.

They do not make use of any static analysis tools, and believe these are too easily circumvented by smart and lazy programmers to be of any real use. They may have been used as a tool to promote personal growth of the skillset by single developers.

Technical Debt

During development the concept of technical debt is often on their minds, although they do not speak of it by name, instead they all have an awareness of what parts of the code are lacking in quality.

Their method of dealing with avoiding new technical debt is heavily oriented towards lean. They prefer making prototypes of new functionality they can show for the customer without writing any real code, and if it is deemed useful they can develop it later. Any time spent on writing quality code for the system that will later be thrown away is considered a great waste of time.

The interviewee makes a distinction between what he calls "incidental technical debt" and regular technical debt. Incidental debt is what is created without an awareness from the team, when corners are cut to meet a deadline, while the regular kind of technical debt is created with discussion and an awareness by the team members. The incidental kind of debt being much more dangerous as it is not known how much of it exists, and it lessens the team's ability to be agile with decisions in the future.

5.3.4 Duego

Duego is a social network primarily aimed at the Brazilian market. The company was founded in 2010 and gained investment capital to fund development in 2011.

The product was developed for 6 months during 2011 before launching. As the launch was accompanied with marketing efforts they felt it necessary to have a reliable and functional product from the start. The founders did several mockups to visually present how they imagined the service to work to the developers and to define the product's functionality.

The service was first implemented in PHP as that was what the initial developers were comfortable with. After a pivot were they refocused to have several front-ends in addition to their website, they rewrote the entire system in Python Flask, making the web and mobile equal citizens to their API.

Process

Duego works with a Scrum-based process. They initially used two week iterations but have moved over to one week to better cope with rapid changes in requirements. They view lack of process as a sign of a prioritization problem in the company, if you cannot commit to leaving developers to do their thing for 1-2 week durations you need to reconsider what you are doing.

They have a strong commitment to writing tests, always including test for the features they have added in commits. Occasionally they write tests before they write the code but this varies per developer and what is to be implemented. Their documentation is entirely in the code except for the API they expose to their customers. The examples in the API documentation is also tested along with running the regular tests in their build system. This ensures that the examples works as advertised when tested against production.

Technical Debt

They accrued a lot of technical debt as the product evolved the first two years as their entire frontend and backend was unified in one giant PHP application. Eventually they decided to rewrite it all when they were pivoting as repurposing the PHP application for the new vision would be more work than doing it again.

They have a strong committal to tests and code quality, which is enforced by code reviews before going into production and they don't let anyone get away with skipping writing tests. However, they find it's rare that they get time to do major refactorizations of areas of the code that are less than ideal, but they are aware of where the problems lie and do not consider it an imminent issue.

As for future issues, they find their current method sustainable. The culture at the company ensures that new code is up to their standards and it is very rare that they need to make compromises to meet launch dates or other demands.

5.3.5 NetClean

NetClean is an older company than most of the interviewed as it was founded ten years ago, in 2003. The interviewed person was hired as a developer in 2005 and has been with the company since. NetClean produces software that can detect child pornography on computers and in networked systems. They sell the software to businesses and aim to make it a hygiene factor at any workplace to have their software installed.

The idea has been the same all along but the product portfolio has been expanded with additional products, such as mail server integration and network deep packet inspection.

Process

Traditionally NetClean has been operating under a cowboy programming style, with developers picking tasks to complete themselves. Lately however they have been moving towards Scrum as the team is growing to the size were methodology becomes important. The main issue they had with the old style was that as the team size grew, it was more difficult to keep track of what others were doing, and it sometimes happened that people picked up the same user story by mistake or other similar clashes.

They have difficulties in prototyping their software, both because it is deployed on customer's computers which makes it more difficult to push out updates frequently, and because it is important that the system works properly.

Testing the software in a production environment is difficult due to the nature of the product. Historically they have not been very focused on testing but they are moving towards more tests now, many parts of the old code is unsuitable to testing and is being replaced as it is being encountered. They do not have any guidelines for either testing or documentation practices, rather it is to each developers own digression. They do have up-to-date user manuals for their customers though.

Technical Debt

NetClean has been experiencing the pains of growing technical debt for a long time as developers have moved on to different parts of the business and the product has grown. They manage the debt continuously through gradual refactoring. As parts of the initial product were coded in Visual Basic and the team has since moved to coding primarily in C# older parts are easily identified. A rule of thumb is to not update Visual Basic code

but rather replace the module with a C# based one, this means that refactoring is an integral part of the development effort.

They have never felt the need to stop feature implementation entirely to work on improving the existing codebase. However, as they have grown they have more flexibility to do improvements to the code as they have a more diverse customer set and don't need to listen to every whim and abide every request. Instead they can spend more time on doing things right rather than compromising. Recently they have also started using static analysis tools to quantify the improvements that are done to the codebase. They find the tool useful to make sure there are improvements, but the large amount of existing errors makes it too daunting a task to try to remove all the technical issues.

They do not have many tests for the client side code, and are not directly trying to backfill it either for existing code as it is in many cases difficult to test code that was not written with testing in mind. Instead most tests are performed manually and they are currently trying to establish a dedicated Q&A process.

They rarely need to do major changes to existing code, but the litter of small problems becomes an issue when that is the case. Doing huge refactoring efforts is much more difficult when the issues are everywhere, when you start refactoring the amount of things that needs to be changed keeps growing even if you are only trying to improve on a small part of the product.

5.3.6 PugglePay

PugglePay is a startup company in Gothenburg that specializes on payments and invoicing for services over the web. The company was founded in 2011 and have three full time developers today, and had four during the initial development stages. When the first developers were brought in the founders had already planned a lot by creating user stories etc. The founders had previous experience from this type of business so they knew a lot about who the customers were and what they wanted. This led them on the right track from the beginning so they could get their first customer only four months after the project started.

Process

The company tries to have a pragmatic approach to how they work. They work in an agile way but don't use any formal processes but instead pick different parts from different processes, such as Scrum, and uses what works good in their team. In their process they use Pivotal Tracker for user story management and prioritization.

Since they work in the financial domain they believe it is important to have a well-tested system of high quality. Due to this they test most of their code, sometimes before the implementation is written but more often afterwards. Also, to increase the quality of the

code, they make use of code reviews. If a developer has written a feature alone another developer reviews it before it is merged with the rest of the code. If a feature is developed in a pair programming session they don't feel the need for a separate review, unless it is a complex and important feature.

They have an API documentation but do not document the code itself in any strict way. They make use of continuous deployment and push their changes to production often, feeling confident in that their tests will catch the issues before they go live.

Technical Debt

PugglePay thinks technical debt is an important topic in the domain they work in. The philosophy they have is that it does not become a big problem as long as you are aware of it. When you take on some debt you buy agility for the moment but sacrifice future agility. When they make changes to the code they try to refactor the code around it. By doing this they increase their code quality and avoid some technical debt.

The company believes that most of their debt has come from big changes in their API. In the beginning they over-engineered parts of the system and the API weren't as flexible as it had to be. The API has been re-worked a few times and now they have three different API versions to manage and have now started working on a fourth version.

PugglePay does not use code comments in their regular codebase as this is believed to be a sign that the code is too complex. However, they use comments to document the API functions. These comments are used to generate a separate website with documentation of the API for their customers. Also, the comments include code examples which are extracted and run as tests.

5.3.7 Recorded Future

Recorded Future was founded in 2009 in Gothenburg, Sweden. Today they have a headquarter in Cambridge and offices in both Gothenburg and Arlington, VA. The founders worked on the project for about 12-18 months as a side project before they got in contact with investors and could work with it full time. In the beginning they focused mainly on financial businesses but after a while they shifted focus to business intelligence as it was a better fit for their product. Today their customers are mostly big corporations and national organizations.

Process

Recorded Future uses a variant of Scrum as a development process. They work in small teams of 4-5 people in two week iterations. In the beginning the company did not use any formal methods, but as the company has grown more formal methods has been added.

The developers don't use test-driven development but they do write unit tests for the parts of the software where they think it is appropriate. They feel that this is an area where they need to improve and plans to do so in future projects.

Code reviews are not part of Recorded Future's regular work process. Occasionally they do perform some reviews but only if a developer actively requests it. Pair programming is also something that is used at times but not in any formal way and only when a developer feels the need for it.

Recorded Future uses Amazon's cloud services and often deploys new code several times per day. They use Chef recipes to manage their servers and can easily push different code to different nodes that have specific roles.

Technical Debt

Recorded Future don't think that the debt they have is a big problem. It helps that their main system is global which makes it easier to maintain and update without legacy issues. They do have some customers with local installations though which are a bit more troublesome. Most of the system is configurable so when they test a new feature they can test both the new and old code at the same time. An issue with this is that a lot of old legacy code is left in the codebase and not cleaned up when an old feature is discontinued. They don't believe it is a problem though as the code is not in use. They think it is better to prioritize the development of new features than to clean up the old code.

Recorded Future has done some technology changes since the start. In the beginning they used MySQL as a database but later they realized that this was not a good fit for their product and migrated over to Sphinx. Eventually they also left Sphinx in favor of Elastic Search. They believe that these changes are part of the natural evolution of their product and that the debt acquired will not become an issue any time soon. They try to act on resolving actual issues before they occur arises and constantly have some architectural changes they plan to implement.

The company believes that they chose a good path from the beginning. Based on previous experiences they knew they wanted to separate the API and not couple it with the backend technology. This made it easy for them to, for instance, switch the backend database technology when issues arose.

Recorded Future have some documentation in the code but no external documentation other than some design documents. They don't have any formal coding standards since they believe that forcing their developers into a specific standard is bad. The developers often converges over time to a common coding style. They don't believe that the code quality or documentation are any issues yet but they need to become better at writing tests.

The company does not use any tools to analyze their codebase and they believe that it would not be very beneficial in their current situation.

5.3.8 Shpare

Shpare is a newly founded company that targets conferences and tries to make it easier for people attending the conference to find who is interesting to them and their interests and book a meeting to get acquainted.

Shpare was initially developed in the spare time of a single developer, who simultaneously learned Ruby on Rails as it was developed. They have had more developers in the time at times.

Process

They operate with a Lean agile approach, as the software is mainly used during conferences efforts are usually concentrated over a few days. A Trello board is used to manage user stories, initially a more strict Scrum-based solution was tried but this proved to be too much overhead to be useful and locked them out of doing things the way they wanted to.

They do test but not rigorously, tests are mainly of the backend systems and not of the frontend. Self-documenting code is the goal with a few comments to clarify complex logic, something that could be done better. The API for the service is exhaustively documented however, as it is business critical.

Technical Debt

As Shpare was initially developed as the developer got acquainted with Ruby on Rails, many mistakes were made. However, the initial version of the product ought to be considered an advanced prototype, and Shpare found the feedback on this version a huge aid in further development and all of the debt associated with it was dissolved. After another year of development the service was heavily refactored once again to a better architecture.

Additional integration testing would be a good idea as there are many issues with functionality breaking and nobody noticing it. The fact that their business is dormant and being used in bursts makes this a particularly pronounced problem. They also believe that adding new functionality before you know that the current functionality is working properly and being used by your customers is a bad idea. Keeping unused functionality around is a waste of maintenance resources.

They find that continuous integration is a great aid in managing debt. As issues can be pushed instantly if you have a powerful deployment environment most visible debt issues will not affect you for any longer period. They find architectural debt a much larger problem than code and tests issues as it must be fixed eventually, and doing so takes a considerable time and effort.

5.3.9 Trimbia

Trimbia is a startup company in Gothenburg developing a business to consumer solution for managing finances. Their focus is on being a lightweight alternative to many of the professional software suites, both in cost and time commitment. They begun developing their prototype in 2012. They first did it as a pure HTML application with some JavaScript to simulate interaction, after validating the concept they set out to build the MVP in Ruby on Rails.

Process

Trimbia tried to work with a loosely defined Scrum method in the beginning. They had weekly iterations with accompanying planning meetings, a synchronized backlog with use cases. They found that it was a poor choice for the prototyping stage as a sprint is too long a duration to set goals for. Also, as the team is quite small the process introduced more overhead than benefit.

The two developers are both themselves the product owners and inquire the business developers for additional requirements as they move along. They also conducted market surveys and talked to customers to get a better understanding of what they would later build.

For code quality, they have few guidelines, they find it unnecessary for teams as small as theirs. They unceremoniously agreed to the Ruby standard guidelines, and all their documentation is in the form of source code comments. They focus on documenting complex data structures and let the rest of the code document itself.

They don't strive to do test-driven development for the entire product, but rather for a small subset. Mainly the financial parts which they feel must work properly for the product to be viable, and also because writing the tests helps define all the special cases.

Technical Debt

So far they have not had many issues with technical debt. Their product is still young and following the same architecture and using the same technology stack they envisioned

from the start. The slow user-pickup rate also means that they have ample time to implement features the “correct” way.

They think the threshold of when technical debt becomes a problem is when the programmer feels reluctant to start working on a piece of code due to the problems present in it, either with the general quality or the complexity of it being a significant hindrance to modifying it. They feel they are far from this pain point as of now but think it might become a greater problem if and when they pivot and greater changes are forced on the code.

Overall their approach has been to keep the server-side components more modular, while the client-side is less refined. This is because they believe the client side will be subject to more change from customer feedback, while server-side change will be motivated by decisions made by themselves.

5.4 Interviews Summary

Looking at commonalities between the different companies, there are many recurring patterns that can be established.

The table 5.1 shows the similarities and differences of the companies and their processes. Most of the companies have been founded in the last 5 years, with Appello and NetClean being significantly older. The most common platform used was Ruby on Rails, which is a popular platform in the startup world in general (Morini, 2011). The older companies used Java and C#, this can be traced to the fact that the Ruby on Rails and Python Flask frameworks were not mature enough at the time of the companies being founded.

Most of the companies produced web solutions, and they all had server-side components that they themselves were responsible for managing and updating. The companies that deployed client-side applications or programs found technical debt issues more concerning as bugs and issues could not be solved immediately, instead the software was on a fixed schedule meaning quality control was of more importance. For the internet based components they all found the ability to rapidly update the live version of their website or service very useful.

Many companies initially tried to use Scrum or a similar process to boost development, but few succeeded in finding it useful, instead it turned out to be more overhead on development efforts. Many described the iterations as being unnecessarily constricting when working directly with customers and rapidly changing requirements. Instead they settled for a cowboy programming style with a task board used to keep track of user stories, with the small team size allowing everybody to be appraised of prioritization and what different people are working on.

Looking at the process and debt issues in more detail in table 5.2, common factors can be found as well. All the companies made use of testing to some degree, mainly for backend

systems and deployments. Many found test-driven development to be too strict, both in up-front effort with writing the tests and also with lessened agility as tests needed to be updated continuously with new functionality. Many expressed regret for not having enough tests, but found enforcing it to be an ineffective tool due to both the time spent and the fact that experienced developers who are opposed to the methodology can work around it.

Documentation was viewed as unnecessary by all but Appello. The view expressed was that it quickly grew outdated, and another sentiment being for self-documenting code, some mentioned the book “The Pragmatic Programmer” (Hunt and Thomas, 1999) as a source of inspiration. Hunt and Thomas (1999) argues that comments in the code indicates that the code is bad and that you should strive to write code that is so clear and easy to grasp that low-level comments become unnecessary. If used at all they should be reserved for high-level explanations.

All the companies that had an API service as part of their business offering were rigorous with keeping it accurate, seeing it as a critical part of their service. Some companies also published manuals, mainly because their business was geared towards other businesses and doing so reduced support requests.

Code guidelines were not in use by most companies. Many said that the guidelines are not constructive and can easily lead to arguments between developers, finding it more useful to focus the energy on developing the product.

Prototypes were used differently between B2B and B2C companies. The business targeted companies had a very close relationship with some customers and felt comfortable in showing them extremely simple drafts of new features and inquiring about their usefulness. The ones that did not, found their product too sensitive to present incomplete versions of or simply found it of limited use. The consumer targeted on the other hand used prototypes coupled with metrics to judge if a feature was being used and if so how, optimizing the service as it was being developed. This is close to the lean philosophy of prototyping.

All companies felt the need to refactor parts of the software as new features were developed, developers often doing so at their own disgression as they encountered it. Some expressed the benefits of not having strict iteration deadlines allowing developers to spend the time necessary to improve a certain system. Many of the interviewees addressed the fact that for developers in small teams it is much easier to keep track of where the debt lies and what needs to be done to address it. Because of this acute awareness they felt more bold in taking on additional debt and stressed the importance of discussing it within the team before committing to suboptimal solutions.

When time was short and features were necessary in a very short time, all compromised on tests and documentation (if used) first and foremost. If that was not enough to save time they simply skipped refactoring ugly parts of the code and kept on patching the old solutions.

The companies that did perform major rewrites either did so because of lack of experience with the environment early on, or as part of a major pivot, where the opportunity for a rewrite meshed with the new business direction. This gave them both the time and motivation to perform the changes necessary. The companies that did not undergo a major pivot still often had a minor pivot where they changed the target customer, in these cases the product was mostly left unchanged.

The one area that companies regretted compromising on, or felt particularly proud of spending time on doing the right way was data structures and protocols. If they had a suboptimal database design initially they found it affected development for a very long time after its conception, and wished they had done it a better way initially. The companies that did spend time on designing the data structure layers better found that even if they needed to change direction or technology it was relatively easy thanks to the well-designed already existing solution.

Only a few of the companies had any knowledge about the Lean Startup methodology and even fewer actually used it. Shpare and Trimbia were the only companies to actively employ the methodology. They started selling the product before it was developed, made use of prototyping and so on. Other companies, such as Burt, were "Lean" in some areas but it had mostly grown into the team as they found appropriate ways of working, not actively researched and employed from the start, in a formal Lean Startup manner.

Table 5.1: Company Characteristics

Company	Type	Founded	Team Size[1]	Process	Platform[2]	Product Type
Appello	B2B	2004	8	Scrum	Java	Map solutions for mobile
Burt	B2B	2009	~10	Kanban-like	Ruby	Web Service, Advertisment Tracking
Duego	B2C	2010	~15	Scrum	Python	Website, Social Networking
NetClean	B2B	2003	9	Scrum	C#	Software, Child Porngraphy Detection
PugglePay	B2B	2011	3	Cowboy	Ruby	SaaS, Invoicing
Recorded Future	B2B	2009	~20	Scrum	Java/Scala	SaaS & Licensing[3], Information agreggation
Shpare	B2C	2010	1	Cowboy	Ruby	Web Service, Social Networking
Trimbia	B2C	2012	2	Cowboy	Ruby	Web Service, Cash flow analysis

[1] Only technical developers are included in this number.
[2] Main implementation language, ancillary parts like mobile applications are not considered here.
[3] Allowing customers to deploy their own installations of the software for private use.

Table 5.2: Company Debt Characteristics

Company	Testing	Documentation	Prototyping[4]	Refactoring Problems	Tech Changes
Appello	Limited[5]	Yes	No	Continuous[6]	No
Burt	Volontary[7], common	No	Yes	Continuous	No
Duego	Mandatory	API	Sometimes	Major Rewrite	PHP to Python Flask
NetClean	Volontary	Manuals	No	Continuous	VB to C#
PugglePay	Mandatory	API	No	Pre-emptive[8]	No
Recorded Future	Limited	API, Manuals	No	Continuous	Database layer twice
Shpare	Limited	API	Yes	Major Rewrite twice	No
Trimbia	Limited	No	Sometimes	Continuous	No

[4] Is prototyping used continuously during the process, not only when developing the first version.
[5] Testing of certain subsystems. Not integrated into the process.
[6] Problems are fixed when they interfere with implementing new features.
[7] Some developers do it because they find it a useful tool. Not integrated into the process.
[8] Refactor bad code even if it is not necessary for implementing a new feature.

6 Discussion

This chapter is a reflective account of the experiences gained while working in a lean startup fashion, the usage of the static analysis tools and a discussion of the meaning and similarities between companies interviewed. From these sources and literature a new way of modeling technical debt is presented. Then using this model we introduce a set of recommendations which details how a new startup should manage their debt and process.

6.1 Analysis

6.1.1 Technical debt problems identified

This details some of the reasons that startups accrued technical debt and the reasons thereof in an attempt to delimit the problem scope and identify the underlying issues. The problems are enumerated and will be referred to later as different solutions are presented.

I It is difficult to talk about technical debt.

The first issue that was identified while talking to industry practitioners is that while technical debt is a well-known concept, there is very little terminology to discuss the details of it. During all interviews the interviewees had to start from the beginning in describing different aspects of their debt.

For example, many companies described that they were not very worried about debt that was visibly present in the codebase, like lack of tests or documentation. While they found debt issues with technologies and processes much more damaging. The communication was however very long-winded, and this is a problem. The models introduced previously by Kruchten et al. (2012) and Fowler (2009) are neither well-known nor an appropriate tool for saying exactly what type of debt a company has. Instead it is always necessary to describe it in detail, which is a waste of time and could be simplified by more powerful terminology when it comes to classifying technical debt.

II The debt you don't know about is dangerous.

Many of the interviewed companies expressed that they were not worried about their respective mountains of technical debt in the lack of tests, documentation and hacks because they were aware of the presence of it.

What they instead were afraid of was the debt that they did not know about, examples of this were weaknesses of their initial design that they had not found yet, lacking adherence

to standards that they were not aware of or how pivoting might affect the aptness of their design.

III Test-driven development is too strict but some testing is good.

All of the interviewed companies and our own work in the primary case utilized testing for development. But it was very rarely done in a systematic way as it was found to be too time-consuming. Instead tests were kept on a low level and focused on the critical parts of the system that could not be allowed to fail.

An issue with not always writing test is that code that had been written without testing was found by practitioners to end up with dependencies which make unit testing difficult or even impossible. So at a later stage when you want to backfill on tests you find yourself with a codebase that is very difficult to write sane tests for due to its complexity. In contrast if you have the mindset of always writing tests for your code you naturally end up with modularized code with these dependencies (Janzen and Saiedian, 2008).

IV Complexity and difficulty in deploying your product makes handling debt more difficult.

Many companies mentioned the relationship between their release cycle and their ability to handle debt. If your product is a client side application, it is more important to ensure its quality than if it is a web site or service. The reason for this being that a website can be fixed almost instantly if you have a quick deployment architecture. An application installed on a user's machine or phone on the other hand you can at most update every week, however this might annoy users and you should try to be more conservative in updates (even if the friction in issuing them is decreasing steadily as app stores become more ubiquitous).

For web services, many felt as if they could afford many dirty hacks in order to make it function, because resolving them when they break is a simple matter. They weighed this against the loss of agility of architectural issues in the project, with the deployment flexibility being a counterweight to handling more technical debt.

V Prototyping is difficult to apply to most problem domains but valueable when possible.

The practice of prototyping and testing out new features using mockups, which is an important part of lean has shown to be a difficult thing to do. Many companies want to make use of prototypes more but find it hard due to the nature of their product, either it being a sensitive thing to test or it only being possible in real production environments.

However, the companies that were able to make use of prototyping on a regular basis felt that it was very valuable, showing that it still is a very useful tool. The question is how can prototyping be done easily with lean development while not spending excessive amounts of time on doing them properly.

VI Static analysis is not a helpful tool early on in software development.

Our experience in the primary case and the sentiments of the interviewed companies is that the available code analysis tools are inappropriate in a startup environment. The tools place unnecessary restriction on implementation when you want to be as flexible as possible. The warnings they produce are often false positives, this sets a bad precedent for developers to ignore warnings. The issue is further compounded by experienced programmers being able to identify the patterns the tool identifies and only work to avoid warnings produced by the tool, and not actually resolving the issues.

This setting changes with time however, and older companies seem to start to benefit from tools like this as the scope of the project grows beyond the ability for every programmer to have a clear view of it. This complexity becomes a source of errors and misinformation and then the analysis tools start becoming useful because developers can spend the time required to configure them properly and interpret the results.

6.1.2 Debt generating activities

This section describes what companies did when they generated debt, sometimes creating debt is unavoidable, sometimes it's required for business and sometimes it is due to process errors, the latter being the thing we attempt to identify.

The obvious villains in the technical debt arena are deadlines. All expect for one startup found that the stress of deadlines was the primary source of technical debt issues, something that was present even in Cunningham's (1992) original article. Deadlines also affected development differently depending on the stability of the startup, companies with a set customer baase had more security in denying customer requests and controlling their own fate than newer startups without any loyal customer base. These instead had to resort to produce results quickly, even if it meant figuratively tossing quality out the window.

This leads to debt creation through lack of communication. This can occur through secrecy (a developer hiding debt from others) due to shame for not implementing something properly, due to stress or lack of knowledge. This debt will eventually surface as the project moves along, and when it does it will suddenly need to be repaid, requiring allocating further effort to the implementation. This problem was continuous at many of the companies and was often not with conscious intent, rather being a byproduct of any technical development. The epitome of this is problem II.

Many startups went through at least a minor pivot. This has shown to be, in some cases, an opportunity to shed much of the debt you have already gathered. For example, when doing a technology pivot you may replace all or part of your technology stack or how data is stored. However, this requires you to have the funds and time to redo your product. Some pivots were motivated by necessity and the companies had to make do with the minimal possible changes to fit to the new requirements. If resources are not

available, this can be a source of major future debt since the architecture required by the new direction may not match with the old, and it will be significantly more difficult to change it later on in the future. In addition, spending time trying to perfect your architecture, for example by through test-driven development is wasted as the goals of the project change, see problem III.

Another source of debt was personnel changes. When developers leave a startup they often take a considerable amount of the knowledge in the project with them due to the small team sizes and abundance of tacit documentation. When they leave the other members then lose the knowledge of the debt they created. This generally became a problem after a few years when technical staff either moved to more business oriented positions or changed companies altogether. This issue was seen in the primary case and was also brought up by the companies in the interviews.

Related to this is the issue of consultants developing software. Appello for example hired consultants to create some of their ancillary products and later ended up with having to redo large parts of it to avoid being reliant on an external stakeholder forever, as the consulting team did not communicate with their own developers very well.

Poor technology choices can be considered a debt generating activity. However, it is very rare for a company to consciously select a poor choice for implementation in order to get short term benefits. Instead, poor technology choices surfaces with time as the system grows and its bottlenecks are discovered. Resulting in a form of unknown debt that requires time and effort to fix once it is discovered and cannot be planned for.

Also, when choosing technology it is important that the developers have the experience and competence to use it. Otherwise you will incur a lot of technical debt as the developers don't know exactly how to work in the best way possible. We saw this issue where the developer learned the framework while building the product which led to major rewrites later on as it was discovered that the framework was not harnessed as it ought to be.

6.1.3 Methods in use

This section describes what methods the interviewed companies used to manage their debt, why they selected this solution and their relative usefulness

The most common solution to handling debt issues that the interviewed companies expressed was that technical debt was less of an issue as long as all developers were aware of it, counteracting problem I. If all programmers in the company knows that tests are missing, or that part of the program has architectural or coupling issues, the debt is less significant than it would be if they were oblivious of this. When developers are aware of where there are debt issues and how serious they are, they can be taken into account when planning feature implementation, which lessens the impact of the debt.

Most companies tried to write modular code with low coupling. By keeping the code modular it is easy to switch out or change parts of the system without it affecting other parts. This approach was applied from the beginning at Recorded Future which enabled them to easily change database system when they outgrew the old solutions thanks to the upfront investment in flexibility. The foresight and planning required however does not mesh well with lean methodologies.

Another method for more conscientious programming was the use of test-driven development. The common use case being complicated code that dealt with data manipulation (not UI etc.) since it aided in explaining the data flows to the developer as well as documenting it. Trimbia among others used this method. Practicing restraint in what should be tested is a way of dealing with problem III.

If prototyping is an alternative for your product and does not require excessive investment to attempt it remains an incredibly useful tool however and can be invaluable in testing new ideas, iterating on problem V. The requirements to make use of it properly are difficult to fulfill in many situations, having access to first-adopters who are willing to sacrifice reliability to try out new things is much more common when working towards consumers and more difficult in business-to-business setups where the stakeholders are betting much more on the system's functionality and reliability.

A common approach, among the interviewed companies, to keep the debt from growing over time is to conduct refactoring. Most of the companies did not do this in a formal way but instead the developers refactored code as they encountered it while working. For instance, PugglePay had this notion of always leaving the code a bit better than when you started working on it, refactoring ugly code around the point where you are working. This keeps the codebase from deteriorating and mitigates technical debt issues.

We also found that it is important to adhere to standards when developing your data structures and protocols. By keeping them standardized it will make it easier in the future when you, for example, want to integrate your product with other software solutions. This issue was voiced by Appello who had issues with supporting and testing older versions of their products that used a non-standardized solutions. Making use of standardized components also gives you greater interoperability and thus flexibility, aiding with problem IV.

Many companies noted that you often have deadlines that you need to meet to make sure your customers are as happy as possible. This is where they incurred a lot of the debt they had. However, PugglePay did not have this issue as they believe that meeting deadlines are not as important as most business developers believe and therefore prioritized the quality of their product higher. We believe that this depends on what kind of product you are developing, what kind of customers you have, in what stage your product is in and how accepting and flexible your customers are in accepting less than ideal software.

This approach would not work very well for companies with problem V, such as Shpare

whose product is mostly used in bursts of a few days during conferences so postponing releases may mean that they miss the entire opportunity to market and test their product. NetClean also suffers from this issue but for them it is due to their product (child pornography detection) and big business customers not being very open to having prototype software running.

As for problem VI, static analysis tools was not used by the majority of the interviewed companies. However, the two oldest companies did find it useful. They did not use it from the start but felt that as the company and their products grew larger the benefits became increasingly more viable. The younger, and smaller, companies felt that it was easy for the developers to keep track of the changes made to the codebase and where technical debt may be present, without any dedicated software.

When it comes to picking suitable technologies for implementation, companies generally picked the solution they were most comfortable with. For many this resulted in later having to change parts of the stack. There is no good way to avoid this zugzwang in dealing with technology or architecture debt. The only escape is through developer experience and selecting solutions that are known to work in the problem space your are targeting. However this is not possible when the problem domain is new and existing, tried solutions do not already exist, which is often the case for startups.

6.2 Solution

6.2.1 The Debt Quadrant

Problem I that we found in our research is that of subdivision and communication of technical debt. Our research found that the existing subdivision of technical debt by Kruchten et al. (2012) is too heavily geared towards quantization. This is a valid approach for enterprise applications where measurement is a very useful tool to communicate to management the hidden overhead of lacking code quality. This model however is difficult to apply to startups as they do not have the time available to perform precise metrics and even if they did, errors discoverable by metrics are unlikely to represent the problems startups face.

McConnel (2007) proposes another division, he splits debt into unintentional and intentional debt, with intentional debt having further subdivision into short and long term. The problem with this classification is it only addresses how debt is incurred, not the disposition of the debt you have. The ideas he brings forward are useful for communicating the consequences of technical debt to non-software developers but is less useful for thinking about what debt is acceptable and what is not. His view is also like Kruchten's in that it is geared towards the more mature organization were issues of code standards and documentation are more significant.

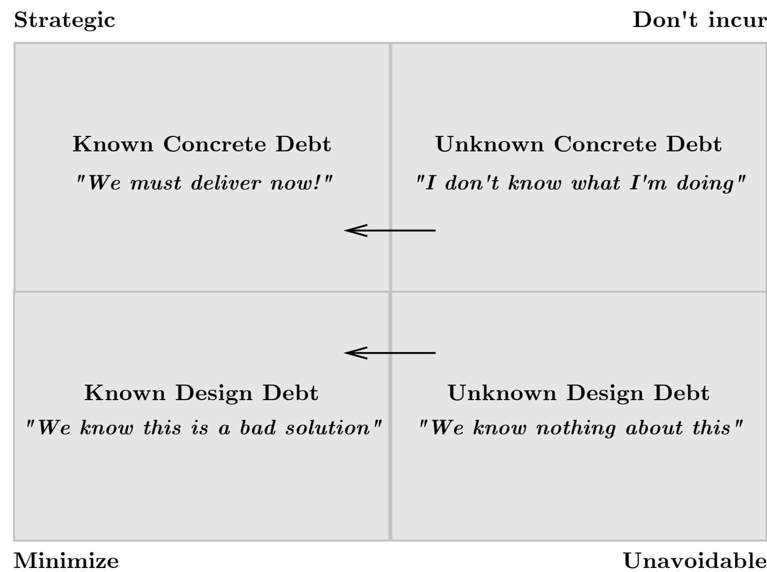

Figure 6.1: Technical Debt Quadrant.

Fowler (2009) suggested categorizing technical debt into a quadrant with four areas of debt, separating deliberate versus inadvertent debt, and reckless versus prudent debt. This is a more astute way of dissecting technical debt, but has the same issue again. The debt is categorized by how it is acquired, not by its characteristics.

The debt quadrant we propose is presented in figure 6.1 mitigates problem I by introducing a new categorization of technical debt that has a different goal from the ones previously proposed, here we group the debt into *known* and *unknown* debt and if it's *concrete* (dealing with lack of documentation, tests, code quality issues, repetition etc.) or *design* (technology choices, architectural issues such as coupling and cohesion) debt.

This is based on the fact that the companies when thinking about their own debt have expressed the difference between the debt you know is there, constantly looming over your work, lessening your agility and requiring maintenance and the unknown debt created through incompetence or inadvertently during development, or incidentally as it's impossible to predict what the proper course of action is before one has been attempted.

It is the debt created carelessly that is a threat to most companies. An example of an *unknown concrete debt* would be a developer, being stressed and not communicating a quick fix he did on a deployment server to solve an issue. This fix then further down the road causes other issues, and the fact that the fix exists at all is only discovered after developers spend time delving and debugging the code. If the developer had instead communicated that he did perform a dirty hack here (and even better documented what it did) it would be less of an issue as at least there would be an awareness of something being less than ideal.

For *unknown design debt* the metaphor of debt is less apt. This type of debt is created naturally while working on the project as the architecture and technology you selected initially turns out to be the wrong choice for your product. This is materialized as problem II, and awareness of this type of debt mitigates it. An example is that you may run into scaling issues with your database, or discover that the way you structured your data makes it difficult to search properly. When this is discovered, this unknown design debt turns into known design debt, which is easier to handle.

Known design debt is issues that developers are aware of exists and affects the system. Generally debt of this type will be prioritized to be resolved quickly because the system can not cope with the new requirements until it is addressed. It can be of a less serious nature where it is only a drag on implementing new features, this can be considered the interest of the technical debt.

Finally, there is the *known concrete debt*. This type can be created in two ways. Either as a conscious decision, with the team deciding that there is not enough time to finish writing the tests or that copying a class and changing it slightly will be much faster than making the original class more flexible. This type of debt is often the source of bugs, although these bugs are often more trivial to fix than the ones caused by design debt, which may require considerable rewrites to resolve. This type of debt can also be created when an unknown concrete debt is discovered, often surfacing as a bug and making it known to the team.

Then the question is, how should debt be managed. Our view is that the four different kinds of debt all should be handled in different ways.

First off, creation of unknown concrete debt should be avoided as much as possible. This debt is dangerous in its invisibility and will result in continuous minor bugs cropping up, requiring time spent investigating the source. A startup should strive to have open communication between developers so that all new debt creation is brought to attention rather than hidden. It can also be avoided by having skilled workers who create less debt incidentally as they are working.

Known design debt should be kept to a minimum. This type of debt has the highest "interest", any architectural issues that crop up as you are developing needs to be resolved sooner rather than later so that you can remain agile in addressing new requirements further down the road. Keeping major design debt around will make new features more and more difficult as you need to find more creative ways to work around the limitations of the system rather than remaking it to fit your vision properly.

Known concrete debt is the type of debt that is your strategic capital. This is where you should create debt when you need to meet deadlines or in order to try out a new feature quickly. While it can result in bugs these bugs are often easily resolved through minor code changes and don't require refactoring. This type of debt is discarded if the feature is thrown out, this can also be said to be true of design debt to some extent, but not as heavily because relics of your design modification will remain even after the

feature is thrown out. An example would be changing a one-to-many relationship to a many-to-many to accomodate a new feature. This change will not be reverted even if the feature that required it is removed.

The unknown design debt is essentially unavoidable. However once discovered, it should be addressed so it does not slow down future development in the same way known design debt would. The only way to avoid it appears to be either through selecting mature technologies, where all the kinks have already been sorted out or through developer experience. Expert programmers are more adept at choosing the correct architecture from the start and predicting the future issues, while novices will not find the same patterns and commit more mistakes.

In conclusion this new definition of technical debt in conjunction with the methods for managing each type helps to mitigate problems I and II that was posed earlier.

6.2.2 Strategy

In order to address the issues and methods in technical debt handling we discovered in startups and through lean development in general, we propose a strategy guide on how to tackle technical debt. This section will detail the overall approach a startup should have with debt, while the action scheme presented in 6.2.3 is a more targeted view of the problem focused around the different stages of a startup's life.

Always discuss decisions that will result in new debt being created – This is motivated by avoidance of unknown concrete debt, illustrated in the lower right quadrant. Unknown debt is very hurtful to your organization, since you cannot take it into account when planning. Instead you should practice discussing decisions related to debt which will result in little being created. This should be done by having an open atmosphere where developers are not afraid to surface issues, either through daily meetings or by having an open floor plan were people can speak freely. It can be further enhanced adding `TODO` and `FIXME` style comments to the code as shortcuts are taken, ideally with attached contact information so you can find the person who added it easily. This turns the debt into known concrete debt instead and helps to mitigate problem II.

The last area in which technical debt should be accumulated is in protocols and data structures – Compromising on as fundamental part of the software as data structures and protocols is generally a very bad decision. This is because while architecture can be changed, code refactored or in the worst case even rewritten, data structures and protocols will persist for a very long time. This will be a continuous burden on development, possibly for the lifetime of the product. Any decision to do quick hacks in communication protocols between clients, or a sub-optimal database design should be taken with care. Having temperance in decisions will ease problem IV primarily through keeping your code standardized. Collecting data in the wrong format can be a huge issue to convert later in development and will require considerable efforts to fix. Companies that carefully picked their initial design were however able to retain their data

definitions over a very long time and through pivots even when the system architecture around the definitions changed.

A quick deployment environment raises your debt tolerance – The ability to deploy your product to customers quickly and efficiently, for example it being a web service in contrast to a desktop application, allows you to tolerate more technical debt, see problem IV. This does not mean that your agility in implementing features or fix bugs will not be lessened by greater debt. Rather it means that you can maintain a high debt and the external quality of your product will not suffer considerable as issues will often be resolved before they have even been discovered by any significant part of your user base. This is very similar to lean's lauded MVP implementation strategy where you only implement features you find are being used or requested, with the parallel being you only fix bugs that cause issues and do so quickly.

Prototyping can be used to manage debt through validation – Prototyping and lean feature implementation remains an excellent tool for validating ideas you have about your product just as Ries (2008) touts. It is also a very useful debt management tool, since there is no debt in a prototype that you later discard. It can also be used for envisioning requirements and thus orchestrating a better architecture initially, rather than diving in into implementing something that is less than ideal, thus avoiding introducing unknown design debt into your product. The main issue with this is the difficulty in applying it to certain types of project, see problem V.

You can take parts of a prototype and turn it into parts of your system, as long as care is taken. If your product domain and customers allow you to make use of prototypes or A/B testing you ought to do so. Reasons for it not being so may be that your system is of vital importance, your customer base fickle or bureaucratic and anything that is not functioning properly is a major detriment.

Static Analysis only becomes a useful tool late in a startups life – Finally, the way of dealing with problem VI. Using tools to quantize your debt becomes useful only once you have a stable customer base. This is when you should delve into your concrete debt mountain and start considering what parts needs to be improved. Static analysis tools then becomes an aid in that they can show your progress in repaying debt clearly, setting up a tool that shows how every commit will affect your debt in a very efficient tool visualizing the problem. Introducing this too early however can be a detriment as clever programmers will work around the warnings produced, and it will affect your flexibility in growing your product if you are shackled by debt constraints too early.

Taking on the right kind of technical debt is acceptable when attracting your customers – Almost all companies we interviewed felt that compromising on some aspects of your development was a necessity. What should be kept in mind is to primarily compromise in the concrete debt realm of your product. Code duplication, lack of tests and documentation and waning code standard adherence should be sacrificed first. While introducing code smells through strong coupling and dirty hacks should at least be documented with a `FIXME`, following the rule stated above. Selecting sub-par architecture

and technological solutions should be vehemently avoided but can be a last resort. This does not directly relate to a problem statement but is rather a general observation of how debt should be approached.

Applying these rules and keep debt in mind during even initial development should result in a more sanguine atmosphere in startups. It should be mentioned however that startups often find themselves in dire situations constrained of resources, and compromises will need to be made that will reflect poorly on technical debt issues for a long time. Options should be considered carefully and the rules still be kept in mind, even if not applied, for the duration of the project. On the other hand, as your product grows more stable with time and you diversify your customer base you can be more picky with what you choose to implement and no longer need to follow your customers every whim. This allows you to slow down debt accumulation by implementing things properly and retain your ability to be agile. When constrained for time, the next section will detail what decisions you should make for your stage in the project.

6.2.3 Debt Strategy Matrix

The previous section focused on general tips for avoiding and handling technical debt, this section attempts to complement the general guide by zeroing in on different phases of a project's lifetime and offering solutions to each.

A division of a startup's debt strategy into different periods of their development is a logical step. We settled on four phases: Pre-Deployment, First Customer, Growth and Adolescence.

Pre-Deployment is the period from the initial idea for the startup to when the first customer is recruited. The length of this phase varies widely depending on the relative complexity of the product. A simple web service can set up a landing page and start recruiting customers immediately while something like Appello's mapping service will require a considerable time investment before the product can start being used. While it is certainly possible to recruit interested customers before the product is usable, as per the lean suggestion, it might be difficult if you target organizations.

Once the First Customers have been recruited it becomes very important to retain these. This makes it possible to start prototyping new features (depending on how fickle they are) and receiving feedback on different parts of the product. Once this starts happening there is no longer time to linger and avoid technical debt as was possible before, and the startup will need to compromise to show their customers that they care about them and develop the product to their needs.

After enough customers have been recruited for the product to have taken off (either through being self-sustaining economically or achieving an exponential growth which proves the product's potential) the Growth phase is entered. This is where the system is likely to show its weaknesses and where quick solutions to keep it operational becomes a

reality. To ensure quality and stability testing becomes more important and tackling the difficulties of problem III becomes relevant. Also as the team size grows documentation and process become more vital as communication becomes more difficult. The important thing to start keeping in mind here is to document the solutions you implement as per the strategy so that they can be identified later when they start becoming the bottlenecks themselves.

Finally a startup enters the adolescence phase. Here the product has found a stable customer base and is in no immediate danger of failing. The safety of a less precarious position allows you to start decreasing your technical debt that was accumulated during the First Customer and Growth phases and start with systematic approaches to handle it. This means that problem VI is worth handling and more traditional software engineering practices and methods are become relevant to structure both your team and your software and ensure its persistent quality over time.

With this temporal division in hand, and joining it with the different kinds of technical debt that Kruchten et al. (2012) mentions and which are related to the quality metrics defined in ISO 25010 (2011) it is possible to make a matrix showing what course of action should be taken in each phase of a startup's life and each type of debt. This matrix can be seen in table 6.1. The first two types of debt listed are design debt while the following four are different forms of concrete debt, the final two are a mix of concrete and design debt and depend on the exact details.

Table 6.1: Debt Strategy Matrix

	Pre-Deployment	First Customer	Growth	Adolescence
Architectural	strong ↓	weak ↑	weak ↓	↓
Structural	strong ↓	weak ↑	weak ↓	↓
Test	↑	weak ↓	↓	↓
Documentation	strong ↑	strong ↑	weak ↓	↓
Code Complexity	weak ↑	↑	–	↓
Coding Style Violations	↑	strong ↑	strong ↑	↓
Low Internal Quality	weak ↑	↑	↑	↓
Code Smells	weak ↑	↑	weak ↑	↓

The matrix shows the different kinds of debt and how they should be approached when the necessity arises. The upward arrows ↑ means that this type of debt can be gathered in this phase of the project while the downward arrow ↓ means that type of debt should be avoided, and possibly be paid back if the opportunity arises. A dash means that type of debt should be contained on the same level. The strong and weak distinctions refer to how pronounced the handling should be. A weak avoidance would mean that

it is acceptable to take on this sort of debt if the medium or strong options have been exhausted.

Some of the tips warrant additional explanation. Tests become more relevant to write with time as after acquiring customers the importance of the system not breaking unexpectedly grows. Documentation starts becoming relevant in the growth phase as your organization grows and the same level of communication may not be possible.

Coding Style and Complexity we have found is not of importance to any of the interviewed companies as the small team sizes lead to consensus with time. This avoids the egregious code and if it ever becomes an issue it is in general easy to fix.

Code Smells and Internal Quality are more important than the different forms of concrete debt, but are still not as important as architectural decisions. These types of debt can be accrued without worry when the pressure of a deadline is looming, but they result in more persistent problems, and can be a requirement to fix later on when testing or documentation becomes a priority since coupling and understandability are prerequisites.

Depending on your product different types of debts ought to be avoided with more vigor. A startup dealing with financial data for example may want to focus more on testing even initially to guarantee that the system does not result in erroneous transactions, similarly a company with their API as an important part of their product does not benefit from forgoing documenting it to save time. Discussing decisions regarding technical debt with your business goals in mind will be helpful in all cases.

Once adolescence is achieved, technical debt issues moves into the realm of more traditional management methods. This means you should reduce the debt you have accrued (hopefully mainly in the form of concrete debt) in general and find a stable level where the debt does not constrain you moving forward but avoid the situation of excessive process in keeping your code base clean shackling you instead.

6.3 Validation

Earlier we introduced some of the issues facing startups regarding technical debt and its treatment. In this section we will go through each of the problems and validate how they are solved by the models and strategies presented earlier.

Current models of classifying technical debt are inadequate.

The terminology for talking about technical debt was found inadequate during the interviews conducted. While the concept of technical debt is well-known, most software engineers have a poor understanding of the borders of its definition and what it encompasses. Several interviewees described different types of debt they had during the

interview, but the communication was long-winded as they did not have the proper terminology to pinpoint what they meant quickly.

Examples of this is how Burt described issues with their technology choices ending up inadequate being a source of debt, which is what we later defined as the unknown design debt. Appello described their initial development of several clients being haphazard, and they knew this would result in what we later named known design debt.

Everybody described the lack of tests and documentation et. al. as a major constituent of their debt that they knew of, which is what we named known concrete debt. Some had also encountered problems with undocumented code left by contractors or older programmers being unfathomable when it was reviewed later in development, which is a topical description of unknown concrete debt.

The debt quadrant we introduced categorizes the debt that companies have and talk about in a succinct way. While it challenges previous technical debt models it does not have the same scope, while previous models focused around the acquisition of technical debt or the absolute quantization of it our models aim is to define apt terminology to discuss technical debt between practitioners and researchers.

Methods and tools for handling technical debt are focused on mature projects.

While researching the existing methods and tools for managing technical debt such as SQALE (section 2.3.1), CAST AIP (section 2.3.3) and Sonar (section 5.1.1) etc. we found them to be focused on more mature projects and not suitable for startups. This finding became even more evident when we interviewed the startup companies. Among the interviewed companies, the younger did not feel the need for any static analysis tools but the older ones did. This approach of not bothering with any dedicated analysis tools in the beginning worked well for the companies and we had the same experience in the primary case. The tools that had been tried by many both as a past-time and as a serious utility for managing code quality were found to introduce more overhead than benefits, the loss of flexibility was a serious concern, as was the difficulty of enforcing the practices on the programmers. This sentiment is included in our strategy through the static analysis guideline.

Regarding general methods for managing technical debt, several of the companies stated that the impact of the debt is reduced if there is knowledge about it, because then it is possible to plan with the debt in mind and avoid putting yourself in a situation where the debt will cripple you by surprise. While the debt was still a factor in slowing development it did not eclipse it, we have included this practice as a general guideline in the technical debt strategy as it seems to mitigate many of the damaging aspects of technical debt. This way of working does however not scale to larger organizations as we found companies experienced issues with it no longer being possible for every developer to be aware of everything when complexity reached a certain level. It is also the case that in larger companies there are often more people involved in the projects, and the personnel rotation is higher which further complicates communication.

Existing methods do not give clear guidance about when and how to handle different types of debt.

In the debt matrix presented in section 6.2.3 we present clear and concrete guidelines for how and when a startup should handle different types of technical debt. A startup can easily consult the matrix and use it as support when making decisions. This is something that was found lacking in the existing tools and methods while researching the primary case, as they were heavily focused on communicating debt to managerial staff and not with small teams where technical knowledge was ample.

The need for the division into different phases is apparent when looking at the interviewed companies. Their issues differed depending on what phase they were in. Older companies found that they could spend the time and resources to backfill issues, very new companies similarly had time but for different reasons, in that their product still had such a small customer base that the pressure was more focused around recruiting customers than fixing bugs and quick additions to meet an external requirement. Companies that had just started selling their product had issues with meeting the demands from the customers as they were not in a position of power to negotiate and often had to compromise with differing levels of success. Finally the companies that had assembled a customer base and the popularity of their offers rapidly growing had many issues with problems surfacing.

Also, in our strategy guide, we have several points which gives guidance on how to act with specific types of debt in a concrete way. For instance, Appello incurred some technical design debt when they did not use standardized protocols when developing in their early phases, which has given them issues later on. Issues like these are mitigated by the rule of always retaining standardized data structures and protocols. Burt is an example of this who were able to retain their initial data definition through their pivot and scaling phases, and the savings in not migrating and changing the data is worth the upfront investment.

Many customers exclaimed that debt issues most often arose due to strict deadlines, something echoed in the literature. Under pressure most resorted to skip writing tests, in extreme cases they would resort to architectural compromises and releasing potentially incomplete solutions if the market pressure was too high. Doing so resulted in unmaintainable code that required several programmer's effort to even modify later on as the consequences of changes were unpredictable. Appello's issues with their debit system were a prime example of this. This is cemented by the debt matrix that was developed.

7 Conclusion

In this thesis a model for how we believe technical debt should be categorized is presented. We found that a few researchers have tried to create such models but found them to be less than ideal as a guide to communicate technical debt issues. The prior models focused on describing how the debt is incurred or how it should be viewed by management rather than what type of debt it is in a way that is relatable to developers. The model instead focuses on aiding communication among developers and researchers and aiding in how you should act on it practically.

We also present a general strategy that guides startups in how to manage their debt. These more general guidelines are accompanied by the debt strategy matrix which is a more concrete guide on how to manage technical debt and how to act on it. It includes guidelines regarding what kind of debt to incur or avoid, depending on in what phase the company currently is. Technical debt can, and should, not be avoided at all times as it can be used as leverage for strategic reasons. It makes you more agile for a period of time but lessens your future agility as the “interest” is paid off.

Debt that concerns the architecture level has been found to generally be considered more dangerous to companies to incur. This is because it is often hard to fix these issues in later stages of software evolution. Concrete debt on the other hand is something that can be incurred when appropriate as long as it is managed and kept under control.

What was also found in the study is that the awareness is an important factor. When the developers are aware of the technical debt it is less dangerous as it can be accounted for when planning for the future. When the debt is unknown and hidden it is more dangerous as it can crop up at unexpected times and be hard to mitigate properly.

8 Future Work

When conducting this study we found some interesting subjects that ought to be looked into further, but unfortunately are out of scope for this study.

It seems that the programming language used is correlated with the way debt is approached in the company. This can be due to the strong philosophies of how things are done within the respective communities. For instance, in the Ruby community there is a lot of focus on writing clear, self-documenting code with a strong test suite. As the code is self-documenting the need for external documentation or lots of code comments is regarded unnecessary. The companies using another language, like Java, seemed to be more in favor of documentation, both in the code as comments as well as external. A study focusing around on how the programming language effects the occurrence of different coding practices like testing, documentation etc. would reveal more about the correlation.

Since the lean startup movement is relatively new, there are few companies that have applied it and survived for enough years for technical debt problems to become a considerable issue. A possible future approach is to conduct a longitudinal study of the debt approach for companies using a lean startup methodology to see how they handle it when they grow more mature.

This study is affected by survival bias as only successful companies were interviewed. Looking at failed startups, with technical debt problems, and investigate how they approached their debt would be another subject to explore. Did they think about it at all? Did they accumulate so much debt that the company couldn't survive? This would be an interesting subject to investigate further.

Finally, the rules that were developed and presented in section 6.2.2 and the debt matrix in section 6.2.3 ought to be validated through application in a lean startup. A study in which the rules are followed in some projects and not in others (possibly evaluating several MVPs) will give greater credence to their applicability. It was not possible to perform this in this project due to the constraint on time and resources present.

Bibliography

Bingham, C. B., Eisenhardt, K. M. and Furr, N. R. (2011), 'Which strategy when?', *MIT Sloan Management Review* **53**(1), 71–78.

Burke, K. (2005), 'Online a/b testing', *Target Marketing* **28**(5), 37–38,42.

Cast Software (2013), 'Measuring and Managing Technical Debt with CAST AIP'. (2013-04-02).
URL: *http://www.castsoftware.com/resources/document/brochures/measuring-and-managing-technical-debt-with-cast-aip*

Code Climate (2013), 'Code Climate Plans Pricing'. (2013-05-20).
URL: *https://codeclimate.com/pricing?v=str*

Cunningham, W. (1992), The wycash portfolio management system, *in* 'Addendum to the proceedings on Object-oriented programming systems, languages, and applications', ACM Press, Vancouver, pp. 29–30.

Curtis, B., Sappidi, J. and Szynkarski, A. (2012), 'Estimating the principal of an application's technical debt', IEEE Computer Society.

Fowler, M. (2004), 'Technical debt', Martin Fowler. Accessed: 2013-01-09.
URL: *http://martinfowler.com/bliki/TechnicalDebt.html*

Fowler, M. (2009), 'Technical debt quadrant', Martin Fowler. Accessed: 2013-04-20.
URL: *http://martinfowler.com/bliki/TechnicalDebtQuadrant.html*

Freddy Mallet (2010), 'SQALE, the ultimate Quality Model to assess Technical Debt'. (2013-03-03).
URL: *http://www.sonarsource.org/sqale-the-ultimate-quality-model-to-assess-technical-debt/*

Hove, S. E. and Anda, B. (2005), Experiences from conducting semi-structured interviews in empirical software engineering research, *in* 'Software Metrics, 2005. 11th IEEE International Symposium', IEEE, pp. 10–pp.

Hunt, A. and Thomas, D. (1999), *The Pragmatic Programmer: From Journeyman to Master*, Pearson Education.
URL: *http://books.google.se/books?id=5wBQEp6ruIAC*

ISO 25010 (2011), Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models, ISO 25010, International Organization for Standardization, Geneva, Switzerland.

ISO 9126 (2001), Software engineering —- product quality, ISO 9126-1 to 9126-4, International Organization for Standardization, Geneva, Switzerland.

Janzen, D. S. and Saiedian, H. (2008), 'Does test-driven development really improve software design quality?', *Software, IEEE* **25**(2), 77–84.

Kohavi, R., Henne, R. M. and Sommerfield, D. (2007), Practical guide to controlled experiments on the web: listen to your customers not to the hippo, *in* 'Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining', ACM, pp. 959–967.

Kruchten, Philippe, Nord, L., Robert and Ozkaya, I. (2012), 'Technical debt: From metaphor to theory and practice', IEEE Computer Society.

Letouzey, J. and Ilkiewicz, M. (2012), 'Managing Technical Debt with the SQALE Method', IEEE Computer Society.

McConnel, S. (2007), 'Technical debt'. Accessed: 2013-04-20.
URL: *http://construx.com/10x_Software_Development/Technical_Debt*

Moogk, D. R. (2012), 'Minimum viable product and the importance of experimentation in technology startups', *Technology Innovation Management Review* (March 2012: Technology Entrepreneurship).

Morini, T. (2011), 'Here's why ruby on rails is hot'. (2013-05-15).
URL: *http://www.businessinsider.com/heres-why-ruby-on-rails-is-hot-2011-5*

N. Denzin, Y. L. (2000), *The Handbook of Qualitative Research*, Sage Publications.

Onwuegbuzie, A. J. (2000), 'Expanding the framework of internal and external validity in quantitative research.'.

PICA Group (2013), 'Ruby Sonar Plugin'. (2013-03-02).
URL: *https://github.com/pica/ruby-sonar-plugin*

Poort, J. (2011), 'Creating mountains of technical debt in lean startups'. Accessed: 2013-05-20.
URL: *http://launchingtechventures.blogspot.se/2011/03/creating-mountains-of-technical-debt-in.html*

Ries, E. (2008), 'The lean startup.', Startup Lessons Learned. Accessed: 2013-01-09.
URL: *http://www.startuplessonslearned.com/2008/09/lean-startup.html*

Ries, E. (2009), 'Embrace technical debt.', Startup Lessons Learned. Accessed: 2013-03-12.
URL: *http://www.startuplessonslearned.com/2009/07/embrace-technical-debt.html*

Ries, E. (2011), *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*, Crown Business.

SonarSource (2012), 'Sonar Technical Debt Plugin'. (2013-03-26).
URL: *http://docs.codehaus.org/display/SONAR/Technical+Debt+Plugin*

SonarSource (2013*a*), 'Sonar Comparison'. (2013-03-26).
URL: *http://www.sonarsource.com/products/software/comparison/*

SonarSource (2013*b*), 'Sonar Features'. (2013-03-26).
URL: *http://www.sonarsource.com/products/features/multi-language-support/*

SonarSource (2013*c*), 'Sonar Technical Debt (SQALE)'. (2013-03-26).
URL: *http://www.sonarsource.com/products/plugins/governance/sqale/*

SQALE.org (2013), 'SQALE.org'. (2013-03-26).
URL: *http://www.sqale.org/*

Storm (2013), 'Storm'. (2013-05-20).
URL: *https://github.com/nathanmarz/storm*

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. and Wesslén, A. (2012), *Experimentation in Software Engineering*, Kluwer Academic.