Practices to minimize technical debt in agile software development: A practical student project case study

Fredrik Bjärås, mas13fbj@student.lu.se
Pontus Ericsson, dat14pe1@student.lu.se

Abstract—Technical debt is an area of great importance in software development and a failure to detect and manage it can lead to such high interest payments in the form of extra effort for every new feature, that the project comes to a standstill. Technical debt refers to the notion of higher development costs later in a project because of earlier design decisions and poor implementation. This paper explores the possibility of mitigating technical debt with the help of three practices that were tested on a team of 10 engineering students taking the course "Software-development in groups" at Lunds Tekniska Högskola. At the end of the course a survey was sent out to all students taking the course and the results from our team was compared to the other teams. The results show that practices for active mitigation of technical debt can result in a positive effect. This by focusing on where the buildup is most likely and letting the whole team evolve the practices throughout the project. The results shows that our team had better collective code ownership and a greater sense of good code quality. However the data was not conclusive enough to answer if our team had more or less technical debt than the others.

1. INTRODUCTION

In recent years the process of software development has been undergoing a change into a more diverse and streamlined process. This process has created a whole plethora of practices and theories which aim to focus on specific aspects on software development. In 1992, W. Cunningham introduced the term of technical debt to describe the occurrence of long term negative effects that the code build up during development. These factors require to be "paid back" in some way during the development process and in turn they take resources and time from the process itself.

This paper is created in conjunction with the courses "software-development in group" (PVG) and "Coaching of programming teams"(Coaching) at Lunds Tekniska Högskola. PVG is a project course which puts about 10 second year students in an agile software team that has the goal to develop a timekeeping program for enduro races. The project consists of 6 days of programming and 6 planning sessions during 6 weeks. The project uses the XP method and focuses on learning to work in a team and handling all the problems that can occur. The coaching course is an advanced course that focuses on the coaching in software projects. The coaches from this course practice as coaches for the project in PVG. During this project the coaches also perform an in depth study, and this is that study.

The focus of this study is to analyze the subject of technical debt, and how it can be managed and mitigated in practice. We also aim to explore the definition of technical debt and how it builds up in the software.

Questions we have and aim to answer are:
RQ1: What is Technical debt?
RQ2: What factors cause it in software?
RQ3: What can be done to mitigate the occurrence of technical debt?
RQ4: Do these solutions present other challenges and problems to the development?
RQ5: How well does these solutions perform in relation to a project without them?

2. THEORETICAL BACKGROUND

Before our analysis and discussion about the effect of technical debt, this section will provide the necessary background information. Firstly by defining the term technical debt and describe how it forms. Then the concept of agile development will be described.

2.1. Defining technical debt

The term technical debt has been redefined multiple times where it has been more and more generalized from the original definition coined by Ward Cunningham[1]. The original term refers to the debt that builds up towards the customer when release deadlines forces developers to make adaptations and concessions in the product. According to Holvitie et al.[1], Cunningham noted that the debt most often required payback and failure to do so could
eventually lead to a stand still in development as the interest on the debt becomes unmanageable.

A more recent definition comes from Steve McConnell that separates technical debt into two categories, intentional and unintentional. Unintentional technical debt is the debt that comes from simply bad code or from poor design decisions whereas intentional technical debt is debt that is strategically incurred to make a deadline[1]. This definition is, according to Holvitie et al.[1], widely adopted by academia.

Holvitie et al.[1] also brings up the definition by Nanette Brown which is interesting as this definition relates to financial debt which could make it easier to understand for people without much of a technical background. The definition says that like financial debt, technical debt incurs interest payments that presents themselves as increased development costs for future stories. The cost comes from previous quick and poor implementation and design choices. Much like financial debt, technical debt can be necessary and the interest can be continuously paid or the debt can be paid off by a large refactoring and redesigning[1].

This final definition presented in this section is the one we used for our study and it comes from a seminar on technical debt[2]. The definition is referred to as the 16162 definition of technical debt, and the attendees of the seminar agreed on the following definition:

In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.

2.2. How technical debt forms

The main cause for technical debt in a project is the pressure of deadlines that forces the poor decisions leading to the rest of the reasons mentioned here[3]. Deadlines are also the reason for intentional technical debt, mentioned previously. Intentional technical debt comes in the form of simpler implementations with more bad smells and this can then create the other types of technical debt[1].

Another big cause of technical debt is architectural decisions concerning different conflicts and architectural inadequacies[4]. Bad decisions in regards to architecture are also very costly to correct and can require large refactorings and major tasks of redesigning[4]. This reinforces the importance of architectural decisions.

One more cause of technical debt is non-unified use of patterns and policies, e.g. violation of naming conventions or design patterns. As well as bad implementation in the form of duplicated code and overly complex code[4].

Testing a program is increasingly more important as it grows and a failure to establish a proper system for tests, including automated testing, will quickly create a lot of debt[3].

We can summarize these factors into more simplified forms: Deadlines, design choices and architecture, a mutual understanding and collective code ownership, simplicity and proper testing.

2.3. Handling Technical debt in Agile development

Agile development is a popular concept in the industry but it has a few critical points that have to be gotten right if a project using it shall succeed. The project that this paper is based on is a project using the agile method of Extreme programming (XP). It was first proposed by Kent Beck in the late 90s and has been spreading rapidly since. If we look at XP from the perspective of the factors that cause technical debt, how does XP’s practices help in preventing technical debt?

Dealing with the architecture is a main component in mitigating technical debt and XP handles it through a few mechanisms: spikes, metaphor, first iteration, small releases, pair programming, coding standards, and an open workspace[5]. Spikes let the team delegate an exploration into possible solutions to a problem and its architectural impact[5]. The use of a metaphor lets the team and the customer come up with a mutual understanding of the whole project and therefore visualize an architecture around it[5]. The first iteration will decide the foundation of the whole project, so it is vital to pick stories that force the team to structure up the whole system[5]. This will force out a architecture for the whole system as well. Small releases will make sure that the architectural structure is created for each release. The last three points involve the interactions between the individual developers and how they discuss and relate to the
code. All these three factors help the developers know what the others think and therefore their knowledge and understanding of the architecture is shared and mutual.

Non-unified use of patterns and policies is mainly a question of the developers not understanding the component they are working on but can also show the lack of structured standards for the project[6]. To mitigate this the developers need to detect these factors, refactor, hold each other accountable for these missteps, and structure up unified ways of dealing with these problems. Some of this is done due to the use of pair-programming. Some through steps of code review, not directly specified by XP. Others is done through discussion between developers in different formats.

3. Method

The aim of this study is to be able to give answers to the research questions that were specified in the introduction and this requires a few approaches. The first one is the theoretical part that aims to answer RQ1 and RQ2, and the theoretical components of the other research questions by reviewing literature regarding the topic. The second part is the practical study. This part focuses on applying a few practices to the project group and analyze the effects this has on the technical debt. It also involves reviewing the buildup of technical debt during the project as whole and analyzing all practices the team uses during the project. This also involves analyzing the choices made by us coaches and the team, and how they effected the technical debt. The results from this will mainly give a background to the answers for research questions RQ3 and RQ4. The third part involves analyzing the differences these factors had on our team compared to other teams. This will give the background for RQ5.

3.1. Practices

The aim of the introduced practices is to strengthen up the critical sections where the buildup of technical debt is most likely. The key areas we have chosen to focus on are: a mutual understanding and collective code ownership, and design choices and architecture.

The first practice is the use of more coordinated discussions between the pairs. Since XP does not directly supply practical methods for facilitating how the team shall communicate, this is a vital part to structure up. The use of a structured way of communicating is vital when you aim to maintain a mutual understanding, collective code ownership and a coherent design and architecture. Our way of structuring up the communication started with the use of 10-15 minute discussions between pairs with interconnected stories. During the project we implemented more structured pair switches between conflicting stories, and a progress report at each switch.

The second practice is the use of a unified code-review system. This comes from our own experience with XP and TDD in the PVG course, where we felt that the review process wasn’t really done. Therefore we introduced a practice and added focus on a review process following each implemented story. After a story was considered done by the implementing developers on that story, it was put up for review and waited for another pair to review the implementation. The review consists of making sure that the code is understandable, i.e. it follows the XP value of simplicity, as well as looking for bad code smells. A story in the project was not considered really done before it passed a review. To make sure that the review process was properly unified we set up a checklist to go through when doing the review.

Finally the third practice is an individual short review of the code after each programming session. In the course between each programming session each week there is a scheduled spike time for the students where they explore different areas and tools that could benefit the project. This practice meant that a small part of that time was allocated for the student to review the code that had been implemented the previous session, mainly parts that they themselves hadn’t work on. The purpose of this practice is to achieve greater collective code ownership, a key rule in XP.

3.2. Team progress

The team started out with a basic understanding of the XP process through a theoretical part of the course they took. This gave them some understanding of the XP practices. During the project we held a team retrospective session where we discussed how the team was progressing. During this discussion we aimed to come up with practical solutions to the problems we faced. This meant that the group introduced new practices during the projects duration.

3.3. Measuring Technical Debt

Measuring technical debt is a difficult task and a good method was not defined in the literature we reviewed. But there are a number of factors that all can be indicators of technical debt, as described in Besker et al[4]. These factors are: dependency violations, non-uniformity of patterns and policies, code issues, inter-dependent resources, and Lack
of mechanisms for addressing Non-Functional Requirements.

3.3.1. Survey: We decided to use a survey as a way to measure technical debt. This by asking questions about the places where technical debt can form, from the factors in section 2.2, and with the addition of questions that queried the number of large refactorings and the refactoring habits in the project. The first survey questions were:

- The source code of our project has good quality
- It is easy to add new features to the code
- The code has none or very few "bad smells"
- The code is simple and easy to understand
- I feel that I have good knowledge about all parts of the code
- I feel confident in working on any part of the code

All with the grades: Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree.

The refactoring questions were:

- Did you manage to deliver the same amount of points to the customer each week?
- How many times during the project did you require large refactorings? Large meaning more than 2 hours
- Did the large refactorings block progress for other stories?

4. RESULT

This first section will present the results from the survey we made. All results are divided into answers from the other teams and answers from our team, team 9. This is done to provide a clearer picture of what differences there are between a team with our practices and a team without.

4.1. Survey results

The survey was sent out to all ten teams that participated in the PVG course. The survey was answered by 45 people, 10 of them being from our team. Meaning about half of all students answered our survey.

Firstly the results from the questions that were asked as statements to either agree or disagree with, are shown in figure 1 and 2. These results show a higher uniformity of the answers in our own team. It also shows a slightly higher positive (agree and S-agree) answer trend from our team.

Figure 3 shows the responses from our question about if the team delivered the same amount of story points each iteration to the customer. The figure presents the results in a pie chart, one for the other teams and one for our team, team 9.

The last two figures, figure 4 and figure 5, illustrates the occurrence and impact of large refactorings in the project. In our survey we decided to define large refactorings to be refactorings that take longer than 2 hours. Figure 4 charts the responses to the question about how many refactorings the team had, again divided between the other teams and our own. Finally figure 5 shows the frequency of answers to the question if large refactorings blocked progress in the project.

4.2. Observations of our group

This part will detail how the team used the different practices that involve technical debt. It also will note changes and new ideas that came up during the project.

Our initial aim was to focus on the practices that we introduced into the team, and let these support the mitigation of technical debt the most.

4.2.1. Spikes: Early on it became clear that the team was looking for ways to discuss the design and implementation of stories. But the time to do this during the planning sessions was limited. To facilitate this need, spikes were used to plan up critical stories. These spikes were documented and used by the team that implemented the stories. We discussed how to best make use of the spike material but did not come up with any more practices surrounding this.

4.2.2. Review: The review practice that we introduced was adopted and used almost always before setting a story to done. Later on the team expressed that the team needed clearer information on what had been done in each story, to be able to properly review it. This was never implemented but it was suggested to use pull-requests or some form of documentation for what was done in each story.

4.2.3. Communication: We have identified proper communication channels as an important way of mitigating technical debt. Our team used a few practices that helped them communicate and facilitate a mutual understanding. We used the XP practice of pair programming. The team concluded that when pairs switched partners there often arose new realizations in how different stories effected the design and architecture of the program. We also added on to this process during the project. One step was switching partners between stories that impacted each-other. Another step was to encourage discussions between these types of pairs, making it so that they structured up a mutual plan before beginning the implementation. This ties in to our practice of coordinated discussions, and we strongly emphasized this practice to the
team in the beginning. After implementing this, the pairs almost always had a discussion with some other pair before implementing their story.

4.2.4. Unified practices: To make sure that all developers had an agreement and mutual understanding of what practices and structures we used we made sure to document and list all decisions of this kind. The first iteration we gave the team a checklist for the review practice. The team later created another checklist for the process of implementing a story. We did also discuss the naming conventions of the packages and classes during multiple stand-up meetings, but did not document this. The team concluded the need for design meetings to structure up a more unified design, but this was never implemented due to the project ending.

5. DISCUSSION

The discussion will be based on the results from the survey and how this ties into the observations we have from our own teams progress. This will be discussed in the context of the research questions that were asked in the introduction.

5.1. Methods and effects

Technical debt is often an invisible danger in every software development project and it requires
some real effort and structure to be able to mitigate it. So what can actually be done to mitigate it? The first part is by looking at how it forms and what can be done to prevent this. In the theory part we came up with a few groups of factors where technical debt forms, and we tried to come up with and focus on practices that would strengthen these areas.

5.1.1. Architecture and design: The architecture and design parts of technical debt was addressed by a few different practices. The main part of analyzing the possible structure and impact of new stories was by spikes. During the planning meeting we tried to identify the stories that the team were the most unsure of and then give the planning of them as spikes. The spiking team came up with possible design-choices that would make the implementation as clean as possible. Then the implementing pair would review the spike and use this as an initial plan. One clear initial problem with this method was that the spiking pair would have trouble accurately planning and identifying the best way of implementing the story. Often the solution would not work when it actually came to it, or it was a flawed solution that created new debt by itself. One of the XP foundations is that the decisions are made by the ones most impacted by the decisions[7]. So by giving the responsibility of decisions that would impact them all, to just a few, made the process more vulnerable. The team expressed the desire for more time discussing and planning out the code, and these types of design-meetings would probably make sure that the best possible way forward would emerge. Perhaps spikes could be used for different pairs to analyze different stories, and then to discuss solutions that would combine them all as a group.

The review process is also a strong tool for dealing with design and architectural choices. The aim of our review process was to make sure that the group had an understanding of the code, and that bad decisions were found early. The review seemed to give the reviewers some insight into the code and removed some errors, but it did not seem to provide the strength of ensuring debt free code. One part of this was the lack of understanding of what actually was changed. As discussed in the results, the use of something like pull-requests would give the reviewers something to reference to. If we also had come up with a design during the proposed design-meetings, the reviewers would have an additional reference to what actually should be done.

5.1.2. Mutual understanding and collective code-ownership: The biggest part of ensuring a unified view of the software was to encourage communication and to get the group working on all parts of the program. As we found...
pair programming as a vital tool in spreading knowledge, it became a point of interest in how we could maximize its benefits. As we found in the results, we could use pair-programming to swap pairs between stories that effected each other in substantial ways. This made sure that both pairs had the full picture and were forced to come up with a common way of solving the problems. Another thing with pair programming we explored was the driver-passenger dynamic. This was not unanimous in the group, as some preferred to let the new one to the pair be the passenger, and others found this to be a problem. It seemed that this was rooted in the confidence and personality of the pair, and that some persons had trouble asking questions about the code. This could make it so that parts of the group could spend a lot more time trying to understand the part they are working with, when a short introduction and some questions would have given them this understanding. During one programming session we decided to force the practice of letting the new student to the story, be the driver. This to make sure that they had to explain and understand the code before starting. This gave mixed feedback, with most agreeing that this would have to be decided separately at each occasion.

Another part of the common understanding was the fact that we continuously created opportunities for the team to work with and review the code, be it through spikes and reviews, as discussed previously, or as the general code reviewing we made the team perform each week. All this made sure that the team had a somewhat good understanding of how the code was structured and worked. But nevertheless there were sometimes confusions and discrepancies in the understanding of the code. Sometimes we solved this through stand-up meetings, or a detailed discussion between some pairs. But to boost this understanding more, we would have had to ensure that there was a unified plan and design, so that all understanding actually aligned to the same goal.

5.2. Comparison of our team and others

This section will discuss the differences and similarities between our team and other teams based on the results from the survey.

The first question concerned code quality and our team scored slightly higher with 80% answering that they think the code quality was good compared to the other teams that had roughly 65%, shown in figure 1 and 2. This could come from our strong focus on reviewing all code before it was considered done. That code with inferior quality was caught earlier as well as poor design decisions. Another factor could also be the fact that we had very good collective code ownership in our team compared to the other teams. In figure 1 and 2 it is shown that all but one student in our team agreed that they had good understanding of all parts of the code. Compare this to the other teams where only about 65% agreed that they had good understanding. The question regarding whether they felt confident in working on any part of the code shows the same trend. Here every student in our team agree where as in the other teams again only about 65% agree. The fact that our team had good collective code ownership means that no one was afraid to change in code that someone else had written. This will make sure that all parts of the code evolves over time when necessary. The trend showing high understanding of all parts of the code also means that the team doesn’t write as much duplicated code. There is a higher chance that the developer will realize that the function they need is already implemented and they know how it works and can use it.

In figure 3 the variation of the story points during the project is shown and there are some differences between our team and the others. For the other teams 29% said that they delivered mostly the same amount of points each iteration compared to our team with 20% saying that. Our team also had a higher percentage answering that the points declined more and more as the project went on, 30% for our team and 11% for the other teams. This could be an indication that we had an increasing technical debt, and as described earlier this leads to increased cost in development. Another reason why our team delivered less points each iteration, more so than the other teams, could be that the review process took more and more time as the project went on. We know from the planning sessions we had, where we also took time for reflection, that the students felt that reviewing was harder in the later iterations because the code base was larger and it was harder to know what had changed. One more reason could also be that the implementations in the later iterations required more and more refactorings.

A big difference between our team and the other teams is the number of large refactorings in the project. We defined large refactorings to be ones that took more than 2 hours. The difference is shown in figure 4. A big majority of our team agreed that we had 0-1 large refactorings in the project, and two answered 2-3. The responses from the other teams shows that 60% said to
have 2-3, and 9% said to have 3 or more large refactorings. The reason behind the other teams having more large refactorings than our team could be that they had more technical debt and therefore needed large refactorings to handle it when it became too much. The fact that 31% of the responses from the other teams also said that the large refactorings blocked progress, shown in figure 5, further support this idea. If the refactoring is blocking other stories from being implemented this points to the design needing improvements, and as discussed previously this is a common cause of technical debt.

6. CONCLUSION

The conclusions of this study will be structured around our initial research questions and the answers will be summarized here.

6.1. RQ1: What is Technical debt?

Technical debt is the debt that a project builds up towards the customer during development. The debt comes in the form of poor design decisions or otherwise inadequate implementation of the code that slows down development for future implementations. Too much technical debt can lead to a complete stand still in progress when the interest cost is too much for new features.

6.2. RQ2: What factors cause it (technical debt) in software?

The root of the problem with technical debt is lack of time. Pressure of deadlines on the developers force the other factors to emerge more. These other factors are architectural decisions, non-unified use of patterns and policies, and improper use of systematic testing.

6.3. RQ3: What can be done to mitigate the occurrence of technical debt?

When discussing the causes of technical debt it is mentioned architectural decisions is an important aspect to consider and it is also very costly to fix. Therefore dealing with architecture is a main component in the mitigation effort, especially in the beginning of the project. There are some techniques to aid the design such as spikes, metaphors and first iteration stories. The conclusion of this report is that the design decisions need to be discussed, planned out, and unanimously agreed upon, to be able to give a debt free foundation for the software.

Another area where technical debt forms is non-unified use of patterns and policies. This occurs when the developers don’t understand what they are working on. These problems need to be detected and refactored, and this is where a review process is beneficial. Pair-programming is also a way to address this area.

6.4. RQ4: Do these solutions present other challenges and problems to the development?

The main challenge with applied methods of handling the architectural technical debt, lies in the fact that most of these put the decisions on a single pair. Many of these decisions need to be discussed with the whole team to be able to really give the best solutions. So it is important to not solely rely on individual pairs to handle decisions for the whole group, but use this as a tool to then discuss around.

The solutions to keeping a mutual understanding mostly seemed to present positive effects, with only some challenges to watch out for with the driver-passenger dynamic.

6.5. RQ5: How well does these solutions perform in relation to a project without them?

From the data we collected it is hard to provide a definitive answer. When it comes to the number of large refactorings the responses from the survey suggests that our team had less technical debt compared to others. However the data from how many story points were delivered each iteration, shows the opposite trend that our team could have more technical debt.

A conclusion that is easier to justify based on the survey responses is that our team had better collective code ownership and a higher sense of good code quality.

6.6. Threats to validity

A big threat to the validity of our results is the fact that we don’t really have a control group. The nature of the course means that while we are testing our theory with our practices, every other coach is doing the same for their own study on their team. This means that when we compare our team to another team in regards to the effect of our practices, the results are also affected by the type of study done by that teams coaches. For example the results can point to that our practices do help improve collective code ownership, but in reality it’s rather a practice or method used by the other team that decreases it. To minimize this threat we chose to compare our team to as many other teams as possible and not just one.

With this said it’s also important to note that every team is different from one another and a comparison between teams is not always fair. A better comparison would have been to compare results from our team before and after our practices where introduced. This however was not possible for this course mainly because of the limited time we had available. This is
the next aspect to consider in regards to the validity of our work, time. The course only has one programming session per week during six weeks, which is a relatively short period to test our practices.

A last thing to note is, that while this report explores aspects of some possible practices to mitigate technical debt, these practices are only a few of the possible solutions and factors of many other. Therefore different combinations and approaches to the practices may have varied results.

6.7. Future work

In the section Threats to validity we discuss the issues with our study of these practices we introduced and as such future work would be to test them in a more controlled environment. To test them over a longer period of time and with a proper control group.

Another possible area for future work in the field of technical debt is to find a consistent way to measure it. This proved to be a difficult part of our study and we did not find any journals with a detailed explanation of how to measure it.

7. Literature

References


