MSc thesis

Computer Science

Reversing Entropy in a Software Development Project: Technical Debt and AntiPatterns

Jacinto Ramirez Lahti

November 24, 2020

Faculty of Science
University of Helsinki

Supervisor(s)

Dr. A-P Tuovinen, Prof. T. Mikkonen

Examiner(s)

Dr. A-P Tuovinen, Prof. T. Mikkonen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Computer Science

Tekijä — Författare — Author

Jacinto Ramirez Lahti

Työn nimi — Arbetets titel — Title

Reversing Entropy in a Software Development Project: Technical Debt and AntiPatterns

Ohjaajat — Handledare — Supervisors

Dr. A-P Tuovinen, Prof. T. Mikkonen

Työn laji — Arbetets art — Level MSc thesis	Aika — Datum — Month and year November 24, 2020	Sivumäärä — Sidoantal — Number of pages 51 pages

Tiivistelmä — Referat — Abstract

Modern software development is faster than ever before with products needing to hit the markets in record time to be tested and modified to find a place in the market and start generating profit. This process often leads to an excessive amount of technical debt accrued even specially in the early experimental stages of the development of a new software product.

This accumulated technical debt must then be amortized or otherwise it will hinder the future development of the product. This can in many cases be difficult not only by the time pressure for new requirements but by the nature of the problems behind the technical debt. These problems might not be apparent and appear just as symptoms that might not directly indicate the real source.

In this thesis, an AntiPattern centric approach to the identification and fixing of the root causes of the technical debt was implemented in the context of a case study of the five-year-old codebase of a startup company. AntiPatterns were not only found and fixed from the codebase but from the Scrum methodologies used in the project and thus these were also analyzed and improved through AntiPattern analysis.

The case study showed promise in this approach, generating concrete plans and actions towards decreasing the technical debt in the project. Being limited to the context of this one company and project, more research should be done on a larger scale to be able to generalize the results.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management →
Software post-development issues → Maintaining software

Avainsanat — Nyckelord — Keywords

Software Development, Technical Debt, Legacy Code, Code Smells, AntiPatterns

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software Systems specialisation line

Contents

1 Introduction

The “Manifesto for Agile Software Development” [5] sprung lots of different interpretations and implementations that fit different programmers, managers and projects. It is very rare for any project or organization to use purely any of these methods, but a tailored set that fits best, with different degrees of success. Sometimes the approach chosen leads to forgetting some of the 12 principles behind the Agile Manifesto focusing singularly on just a few of them, which will likely lead to problems if not careful. For instance the first principle which talks about continuous delivery (CD) and delivering value to the customer as early as possible combined with the second principle which talks about the changing requirements during development, can be very dangerous if principles like the ninth principle which talks about the need to keep technical excellence and the good design of the product in mind, is forgotten. This could lead to thousands upon thousands of lines of code being delivered regularly but without any kind of structure, with abandoned features looming around. This will inevitably lead to the codebase being completely ungovernable which will in turn make future development that much more difficult.

The need for change is the main driver of software development, be it a bug that needs to be fixed with a change in the code or a new feature that needs to be implemented through a change in the code. Developing a software product can be roughly reduced to these two types of changes that will lead to a more reliable and feature rich product delivering value to the customer. Therefore, any time that the development process leads to a codebase that is not easy to change everything will slow down and in the worst of case grind the development process to a halt if something grave enough happens. The third and sometimes forgotten reason for change is just fixing this degenerative process by refactoring. And we would say forgotten, because refactoring should not be a thing that is done as a last measure to fix a problem that has been avoided for long but as preventive measure that will allow keeping the software in a state of technical excellence and good design, as the Agile Manifesto’s ninth principle says, regardless of the volume of features and bugs that have been implemented and fixed. As Martin Fowler puts it in his book “. . . first refactor the program to make it easy to add the feature, then add the feature.”[14]. Otherwise you will just make your life harder and delay the inevitable. The later the problem is dealt with the costlier it will be and the longer it will take.

In the current job market, where programmers change companies every couple of years [26], an added complexity arises, particularly in smaller projects that might have just a few or maybe a single programmer. If the programmers leave a project in the state previously described, the next person tasked to make a change will have a very hard time to make any needed change, be it urgent bug fix or some other supposedly mundane task [18]. In this case we are talking about legacy code within a legacy system, which traditionally denoted a codebase that was difficult to work with but was of vital importance and thus needed to be maintained [6]. In its literal definition legacy code is just a codebase that a programmer has inherited from another programmer that most likely is not available anymore for consulting on it. With the increased pace of development and job changing this phenomenon does not happen just when a gray bearded programmer retires and leaves behind his lifelong COBOL or Fortran projects which are the backbone of a humongous system, it happens in the smallest startups and projects when the initial push is made and as mentioned before, the essential care for good design and technical excellence is forgotten. This is the reason why the definition given by Michael Feathers fits better the new paradigm: “Code without tests”. [13] This definition not only shows the main flaw that most inherited projects that are immediately labelled as legacy code due to their complexity and difficulty to change without risking breaking something, is simply the lack of tests. Tests are a fundamental tool that do much more that “test”, but they document the code and the intentions of the original programmer and serve as a failsafe for changes that should not change any external behavior. Refactoring is also a near impossible task without having test that will make sure that the changes made don’t affect the functionality of the software, making them indispensable but sadly disregarded.

During the process of developing software easy and fast solutions are preferred over more arduous and time-consuming implementations that might be better on the long run. This process generates what is called as technical debt [33]. If this technical debt grows uncontrollably, what were just small shortcuts might become big obstacles for the further development of the software. The longer it is left unchecked the more difficult it is to get a handle of it and be able to amortize the accrued debt.

One concept that can help identify cases like what has been presented is the concept of AntiPatterns, as presented by the “Gang of Five”, which are common solutions to problems with negative consequences.[7] AntiPatterns are thus the opposite to Design Patterns which describe solutions with known positive consequences in the structure of software. AntiPatterns are usually the result of haste, apathy, narrow-mindedness, sloth, ignorance or pride, which sadly are often present in software development. A pertinent AntiPattern to the previous discussion is the Lava Flow AntiPattern, which comes out as code that no longer has any function in the program but it is not clear enough to be simply removed and usually due to it being an inherited codebase, the original programmer can't be asked about it. Thus, this dead code (the alternative name of this AntiPattern) just increases the complexity and becomes a piece of code that everybody just avoids, working around it, generating even more unnecessary complexity. This AntiPattern is solved by making sure that the architecture of the software enables identifying clearly dead code, which in turn enables its prompt removal. Other AntiPatterns that every programmer has probably seen but maybe not thought of them at that moment are: Spaghetti Code, Cut-and-Paste Programming, Vendor Lock-In and Reinventing the Wheel. All of them have their own solutions that should be known to enable moving towards a better architecture and design.

In this thesis we will be looking into how to reverse the complexity of a codebase, Robert C. Martin puts it very well in the preface of Working Effectively with Legacy Code "It's about reversing entropy".[13] The thermodynamic concept of entropy is a great analogy of how a codebase increases in complexity if not regulated from the outside. How can a case of ramping entropy like this be identified? How can it be controlled? And finally, how can it be reversed? The thesis also will be aided by a concrete case study of a software project in need of extensive refactoring, due to fast development focused on finding the right market niche, neglect of design patterns and testing. It is in this project that the Lava Flow and Spaghetti Code AntiPatterns can be clearly seen. The main tools used to solve this problem will be the previously mentioned refactoring and testing but additionally implementing proved effective design patterns and processes and finally thinking about the permanent cultural changes needed to avoid the repetition of the same mistake before too long.

The rest of this thesis is structured as follows. Chapter 2 goes through the background of the thesis covering what software development looks like today, how software can become hard to change and what tools we could use to help us get rid of problems that inhibit change. Chapter 3 sets up the research questions and methodologies used in this thesis. Chapter 4 contains the case study at the center of this thesis that analysis the problems in the codebase of a software startup with the use of code smells and AntiPatterns. Chapter 5 serves as the concluding remarks of this thesis.

2 Modern Software Development

This chapter introduces the main concepts and ideas that are the foundation of this thesis. Section 2.1 goes through the agile software development model with a couple of concrete agile frameworks as examples. This is followed by section 2.2 which explains the need for change that is a constant in software development. Then section 2.3 covers factors that can inhibit the change of software. After that, section 2.4 introduces the concept of code smells and static code analysis that can be used to find the code smells in a codebase. Finally, section 2.5 explains the concept of AntiPatterns which will be in the center of the methodology used for the analysis of the case study at the end of this thesis.

2.1 Agile Software Development

The way software has been developed since the 1970s has clearly evolved from the more heavyweight processes to more lightweight development methods that prioritize iteration and prototyping over heavy up-front planning. This change become very apparent during the 1990s when many software developers started publishing their own ”lightweight development” frameworks and methods in contrast to the heavyweightness of the processes they had seen in the industry until then. Scrum [29] and eXtreme programming (XP) [4] are examples of these software development methods created in the 1990s that still live strong. It was a group of these developers and thinkers that decided to gather and ended up writing the Agile Manifesto [5] which is contains the core values and principles that these new methodologies had in common.

The Agile Manifesto promoted a new approach to software development that focused on collaborative development where processes are secondary to the people and their interactions. At the same time, it promoted the minimization of unnecessary work like documentation for documentations sake and instead limit it to what is absolutely necessary, leaving more time to producing working software that can increase the value received by customers and stakeholders. The active participation of customers and stakeholders in an ever-ongoing collaborative effort through the development process is also on the core of all agile methodologies, avoiding too detailed planning upfront that will inevitably be obsolete as the product takes its form during development. All this means that the inherent uncertainty of software development projects is accepted and following agile principles and methodologies it is much easier to adapt and respond to the inevitable changes to the initial plan over time. In summary, agile methodologies focus on the ability to create value, responding to change and enabling strong collaboration with the customers and in the development team itself.

Agile has now grown immensely in its 20 years of life. The number of frameworks, methodologies and practices is staggering. This can be seen in Figure 2.1, which is Christopher Webb's (Deloitte) depiction of what the world of Agile looked like in 2016. From it we can discern that agile is not only used for software development teams but has been scaled up to the whole organization level or business model with frameworks like Scaled Agile Framework (SAFe) [11] and the Lean Startup methodology [27].

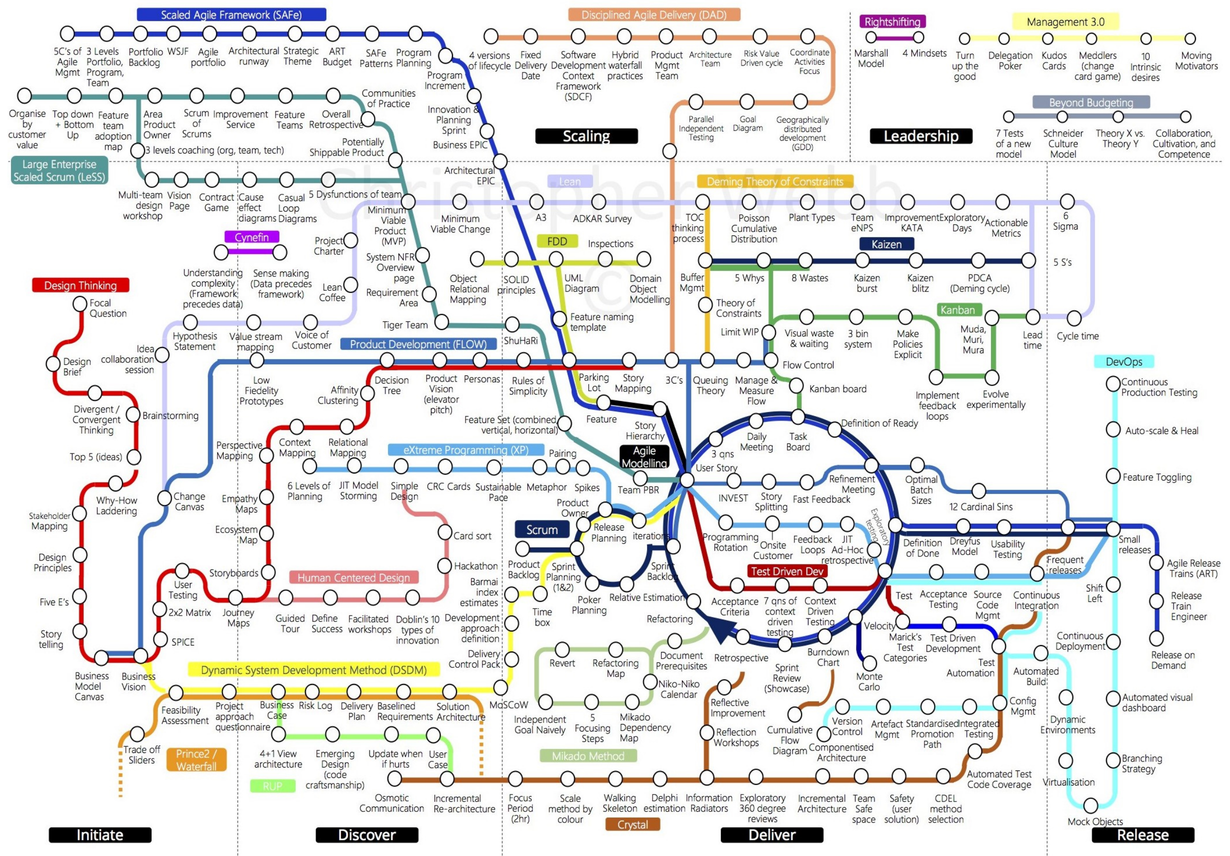


Figure 2.1: Christopher Webb's depiction of the Agile Landscape [34].

Another prevalent phenomenon today is the use of just certain practices that are part of the different frameworks and methodologies without adhering in full to any of them. Some examples of these kinds of practices are the ones derived from XP like pair programming, test driven development (TDD) or continuous integration (CI). You will rarely a development team using these practices talking about how they use XP, but rather talking just about agile instead. It is good to keep in mind that this tendency does not always lead to the desired results when decision is made through improvisation and without thinking how the changes and modifications might affect the bigger picture. In these cases, it is important to notice what is not working and be able to adapt and change in the spirit of the Agile Manifesto.

To understand how agile can is used in practice and how it has influenced other facets of software development the next two subsections will focus on the Scrum framework, the most prevalent agile framework, and the Lean Startup, a business development methodology with clear influence from agile principles and widely used in software startups.

2.1.1 Scrum

The Scrum Framework is the most popular Agile Frameworks used in software development projects with at least the 75% of the respondents of the 14th Annual State of Agile report using it or some kind of hybrid variation of it [32]. This prevalence is why some people even use interchangeably scrum and agile as synonyms which they are not.

Scrum heavily focuses in the iterative and incremental nature of the software development process with a cyclical workflow that repeats in each iteration, which is called a sprint. In its very core Scrum has the idea of a self-organizing development team that through tight collaboration develop the product itself. Scrum benefits of a development team that is small enough to be agile but not too small that progress would be slowed down by its small size. The coordination needed to manage a team larger than 9 members is too complex which becomes detrimental to the ability to follow the principles of scrum.

A scrum team includes two other important roles in addition to the development team. These roles are that of the scrum master and the product owner (PO). The scrum master is the person responsible for the team following the Scrum Framework and help resolve impediments which may appear. This impediment solving is probably one of the most difficult tasks of a scrum master, as in some cases the impediments might be very hard or even completely out of the hands of the scrum master to solve but it is an essential task for the team and PO to work efficiently. The PO is the person representing the client, knowing what the client wants from the product being developed. In many cases the PO is the client or from the client organization itself, which is preferable but it not absolutely necessary. The functionalities to be implemented are managed in the so-called product backlog, which is maintained by the PO. In essence, the product backlog is just a list of prioritized work to be done by the development team.

Figure 2.2 shows the development cycle presented by the Scrum Framework. The product backlog, which was previously mentioned, on the left is implemented in 2-4 weeklong periods called sprints that result in a working increment of the product. On the beginning of each sprint the scrum team plans the sprint in an event called sprint planning, where the whole scrum team decides what work will be done in that sprint. The result of sprint planning is the sprint backlog that is a set of tasks that have been taken from the bigger product backlog. The sprint backlog sets the goal for that sprint, but as work is done during the sprint the development team and PO will negotiate if for any reason something is not going as expected and the scope needs to be adjusted.

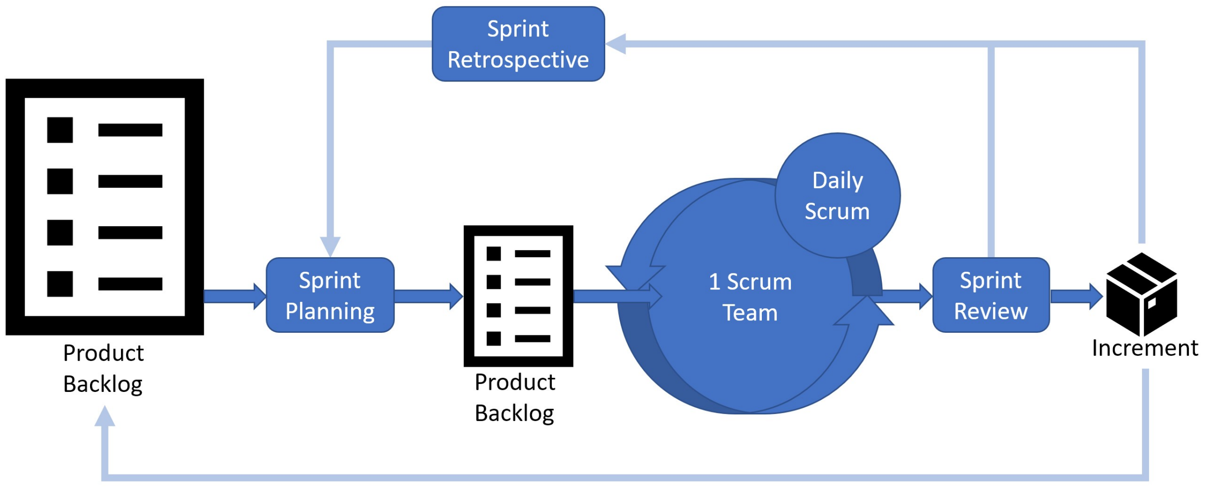


Figure 2.2: Graphical depiction of the Scrum Framework's development cycle.

During the sprint itself the development team implements the requirements in the sprint backlog as they see fit. Every day there should be a short meeting, the daily scrum, where the team inspects the progress toward the sprint goal and plans the work for the next 24 hours. It is structured around three questions that each team member answers keeping the sprint goal in mind:

- What did I do yesterday?
- What will I do today?
- Is there something that impedes my or our progress?

After the sprint is over the sprint review is held to check if the goals for the sprint have been met. It is here that the new increment is presented (thus sometimes receiving the name of ”demo”) to the scrum team and key stakeholders to elicit discussion and feedback that can then be used in the future through new requirements that would be added to the product backlog.

Finally, the scrum team convenes in the sprint retrospective to think about what went well and what would require improvements for the next sprint. The things that are inspected in this meeting include everything from people and relationships to process and tools. The result of the retrospective should be a list of action points to be able to implement the identified needed improvements.

It is good to notice that all these different events follow the idea of time-boxing, with a set amount of time not to be exceeded, for each of them. This is to keep the meetings on the given topic and avoid wasting the time of the whole team in things that should be discussed in smaller groups or not at all. The scrum master is in charge of guiding the team to keep into the intended time-box. A summary of the different scrum events with their recommended time-boxing [30] can be seen in table 2.1 bellow.

Event Name	Main purpose	Time-box
Sprint Planning	Creating the backlog and setting the goal for the new sprint	8h
Daily Scrum	Daily check of the progress towards the sprint goal and solve impediments	15min
Sprint Review	Demonstration and feedback of the achievements of the sprint	4h
Sprint Retrospective	Recognizing what went well and what went wrong to improve in the next sprint	3h

Table 2.1: Scrum events with the maximum recommended time-box for sprints of 1 month given by the Scrum Guide [30]. These should be shorter in shorter sprints.

After all this, the scrum team starts a new sprint with a new sprint planning event to restart the cycle. Scrum, in the spirit of the agile manifesto, clearly guides towards collaboration within the team and with the client, continuous improvement and continuously delivering working software.

As stated at the beginning of this section, scrum is very widely utilized, but not always in its pure form. There are variations like ScrumBan that fuses concepts of Scrum and Kanban (a lean methodology) or something like a Scrum/XP hybrid. These in particular are more well-defined variants with their own sets of guidelines but then there are the millions of variants that appear in each individual team due to experiencing some part of the framework as bothersome or just not useful in that particular case. These variants contain modifications to the basic Scrum Framework that are called ScrumButs [35]. They usually take the form of the following sentence: ”We use Scrum, but for reason X, we take workaround Y”. Some of these workarounds are temporary for a good reason, but sometimes the workaround or the reason might be completely misguided and end up harming the processes. This concept will be further discussed in section 2.5, where some of these variants will be considered as AntiPatterns as they have a similar structure.

2.1.2 Lean Startup

Lean Manufacturing is a production method that focuses on minimizing waste derived from the way of working of Toyota. Lean as a widely known concept was popularized by Womack and Jones in the early 1990s [36]. What made this way of approaching manufacturing so different was the focus on flexibility and continuous improvement. Instead of having a production line being able to produce just a specific product an effort should be made to be able to change what a given production line is producing. Another famous element of lean is the ability to use the ”stop” button, stopping the whole production line, whenever an issue is encountered, for it to be immediately fixed instead of potentially wasting much more time later.

Lean Startup [27] fuses Lean and Agile to build a business level methodology directed towards technology startups where the ability to move fast and change directions is essential. Startups do not have the luxury of spending lots of time developing a product behind closed doors and then releasing it when it is ready, as it just might not have a fit in the market. Unlike Agile which focuses on delivering working software often, Lean Startup’s approach is to deliver features or feature prototypes as fast as possible to the end users to learn from their reactions directly. The main loop of the Lean Startup process is Build-Measure-Learn where the process starts in reverse from setting a hypothesis of what feature or product that the markets/end users would be interested on that we would want to learn about, then setting a set of measurable parameters that would indicate that the hypothesis was correct and then build it. Once built the product/feature is delivered to the end users to measure and thus learn if the hypothesis was correct. The results of this process are then used to decide how the future of the feature, product or whole company should be molded.

The changes in direction that are derived from what is learned on the Build-Measure-Learn cycle are called pivots. Pivoting, a core principle of Lean Startup, is a course correction done to find a new hypothesis or strategy for the startup to be successful, there are many examples of startups that flourished after making even radical pivots [31, 3]. There are many types of pivots, some related to simple feature sets and their scope, others regarding the higher level business model and those that directly require changes to the underlying technologies used.

The other big principle of Lean Startup that has been adopted very widely is the concept of Minimum Viable Product (MVP). A MVP is a version of a product that requires the least amount of effort to create but maximizes the amount of learning from customers to decide whether it is a good path to follow further or not. In the case of a software product it does not even need to be a program itself, but for instance a video that depicts the functionality and can be presented to customers as if it was something already built. A famous example of this is how the creators of the file-sharing tool DropBox first tested their idea for the product, with a video exemplifying what DropBox would later become without writing a single line of code [21].

2.2 Constant Need for Change

Software development is in essence just a chain of changes that modify the code of the software to provide more value or make it better. There really are only 4 main reasons for software to change: new features, bug fixing, improvement of the design and performance optimization.[13] The speed at which the world moves and changes makes the need for making these changes as fast and efficiently as possible even that more important. This is why the Agile methodologies are a great fit for today's software development environment. Conboy really encapsulates this in his definition of agility as "the continual readiness to rapidly or inherently, create change, proactively or reactively embrace change, and learn from change, through its internal components and relationships with its environment" [9].

Implementing features and bug fixes is the daily work of any software developer. Most of the tasks in the backlogs of projects will feature this kind of changes. They should be straight forward, delivering the expected behavior to the clients. But for it to be efficient the codebase must be malleable and not present friction whenever a change needs to be made.

On the other hand, design improvements and performance optimizations do not change the behavior itself but either makes the codebase itself better or makes the software work more efficiently. In particular design improvements are rather important but sadly inexplicably relegated to the status of work that will be done later when there is nothing else ”more important” to do.

2.3 Factors That Inhibit Change

There is no question that being able to make changes to a codebase without encountering obstacles that make it slower and cumbersome is of the uttermost importance for a software development project to be able to progress, as stated in section 2.2. There are many reasons and attitudes that will lead to a codebase losing its malleability and ending up being very difficult to change. It is also a vicious cycle that keeps getting worse over time if there is no intervention to try to fix it, making it ever more difficult to ever be solved. In the next two subsections sections we will cover the concepts of technical debt in subsection 2.3.1 and legacy code in subsection 2.3.2 as factors that inhibit change.

2.3.1 Technical Debt

Technical debt (TD) is a metaphor, coined by Cunningham [10], that illustrates how software development teams can at times make suboptimal solutions seeking a fast initial payoff that will inevitable be counterproductive on the long term. The typical example of this would be risking the internal quality of the codebase in favor of delivering features faster. Over time these deficiencies in code quality build up in a similar way as financial debt does, if nothing is done to pay it off. A codebase with a large amount of TD will require more time to implement any given new change.

There are innumerable causes that can lead software development teams to incur in TD. Although time pressure is the overwhelmingly most common cause of TD but not the only one. Other causes can be the inherent complexity of the underlying design or the already accumulated TD, no adherence to good standards or techniques, lack of skills of the developer itself and insufficient testing. A big consequence of accruing TD in addition to a low-quality codebase is the delay in subsequent deliveries as the TD is on the way. Other consequences range from more people related ones through increased stress, demotivation and fall in productivity to even the financial losses that the increasingly larger delays entail. [28]

Not all TD taken is the same and has the same reasoning or even consciousness behind it. The classification of TD into a quadrant [16], as proposed by Fowler, is a good way to visualize this. In table 2.2, you can see the Technical Debt Quadrant with its two axes reckless/prudent and deliberate/inadvertent. The horizontal axes reckless/prudent represents the understanding of the consequences of the TD that the team will be incurring in while the vertical axis deliberate/inadvertent represents the realization of the actions incurring in TD.

	Reckless	Prudent
Deliberate	”We do not have time for design”	”We must ship now and deal with consequences”
Inadvertent	”What is Layering?”	”Now we know how we should have done it”

Table 2.2: Technical Debt Quadrant as presented by Fowler [16].

Recklessness is the biggest problem in both deliberate and inadvertent cases. Taking some TD is reasonable in many cases, but it should always be done in a conscious and deliberate manner. That way we can have a reasonable expectation that it can also be paid back. Lack of anticipated knowledge but good form, like what the bottom-right quadrant depicts, is also better than pure recklessness, as we can expect that an implementation made following good quality standards otherwise will be easier to change to the more appropriate design.

Another way to categorize TD is through the use of a type categorization, where TD is categorized by the area that it originates from or what it affects. Some of the most notable types of TD are: Testing debt (missing tests, no test strategy or not executed tests), design debt (problems in the design or architecture itself that affect maintainability), defect debt (known and unknown defects that have not been fixed) and documentation debt (deficiencies in the documentation, be it being outdated, incomplete or completely missing) [38]. This categorization is good to understand what kind of approach to take to combat the particular type of TD in the project in question.

The biggest enemy of accumulated TD is the same as what usually initially accrued it, time pressure. A software development team and also management must understand that however urgent new features and changes are, if TD is not dealt with in a timely manner it will ever increase. You have to start paying your debt back at some point or risk complete failure, when a simple new feature ends up needing weeks, instead of just merely days if the codebase had been maintained better.

2.3.2 Legacy Code

The concept of legacy code is traditionally understood as old code that was built with outdated technologies and techniques but still runs some fundamental part of a business and thus needs to be maintained as is although difficult, as it would be too expensive and time consuming to completely rewrite with modern technologies [6]. It usually generates a sense of disgust and dread, with a sense that the codebase will probably be difficult to understand and work with, making changing anything in it very arduous.

A more modern interpretation is that legacy code is just code inherited from someone else [24]. In particular when that person is not available anymore to help you understand and guide you through it. This kind of legacy code is everywhere as software development projects flow faster than ever and the developers themselves change workplaces every couple of years, leaving their work as legacy for others to continue developing or maintain.

But these definitions do not really guide us to a clear solution when dealing with any kind of legacy code. This is where Feathers' definition of legacy code as code without tests is much more helpful [13]. A codebase without tests is very difficult to work with, as you cannot be certain if any changes you are making to this unfamiliar codebase will break something or not. Tests also work as great documentation, elucidating what the intentions of the original functionality has been.

Focusing in this last definition we can see that to get control of legacy code we need to get it under test. This is no easy task and need to be done systematically and incrementally as needed.

2.4 Code Smells

A popular concept in software engineering is that of bad code smells, usually code smells or just smells for short. Kent and Fowler popularized this concept with a list of 22 code smells that they had identified and were able to concisely describe [14]. Most of them are now common knowledge amongst developers like for example the following ones: Duplicate code, long function, long parameter list, global data and mutable data. As defined by Fowler ”a code smell is a surface indication that usually corresponds to a deeper problem in the system” [15]. What this means is that code smells are quickly identifiable indications of possible problems in the codebase. Not all smells end up being bad as for example some long methods have a reason to exist, but it is usually thought of as a smell and should be investigated to corroborate that it is not one. Fowler’s definition also indicates that usually the smelly parts might not be the problem itself but might be a signal of deeper problems.

One important aspect of code smells is that they usually have one or many associated simple refactorings that can help getting rid of the smell. For example in the case of long functions the extraction of function refactoring will make it easy to divide the logic of the function into smaller ones.

One way to detect possible code smells automatically is the use of static source code analysis tools [37]. Static source code analysis tools analyze the source code without the need to compile and run it, with different tools discerning different properties found in the code [25]. Some of the tools like for example linters are usually run continuously in the Integrated Development Environments (IDE) or editors used to write the code. This way the tool can immediately warn the developer of suspicious code as it is being created. Other tools provide visualized representations of the code and the relation between different files, classes and functions.

2.5 AntiPatterns

Code smells might be a sign of underlying and more encompassing patterns which can be represented as AntiPatterns. AntiPatterns are applied not only to code but to the associated software architecture, design and project management of software development projects too. The biggest difference comes in the formal representation used for AntiPatterns that describes not only the AntiPattern itself but also its causes and a possible generalized refactored solution to get rid of it. AntiPatterns were popularized by Brown et al. in their book ”AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis”[7].

Usually most AntiPatterns are implemented in a disguise or context that might initially seem beneficial but end up having harmful consequences on the long run. A good example of this is the golden hammer AntiPattern which in its most concise definition would be the use of the same solution everywhere without thinking if it is a good fit for the given problem because it worked well in another context. Although the intention is to use something that has been proven to be beneficial it ends up working against itself in the wrong context.

The main root causes for AntiPatterns to emerge in a software development project are the same as the ones that will make the success of a project very difficult: Haste, apathy, narrow-mindedness, sloth, avarice, ignorance and pride [7]. The list clearly shows that to be able to avoid AntiPatterns some kind of conscientious decision needs to be made to keep the project under control.

To be able to work efficiently with AntiPatterns a template is to be used for their representation. The full AntiPattern template used by Brown et al. [7] can be seen in table 2.3. This is a way to formally describe an AntiPattern in detail. Using these template, we are able to catalogue harmful behavior that we find regularly in software development projects and link them to a generic solution that can be used to get rid of it.

Brown et al. [7] divide their identified AntiPatterns into three types: Software Development, Software Architecture and Software Project Management AntiPatterns. An example of a Software Development AntiPattern would be the previously mentioned Golden Hammer AntiPattern. Reinventing the Wheel is an example of an architectural AntiPattern where every problem is approached as it were completely new, and a new solution would be devised for it although a similar or exactly equal problem might have already been solved before. Finally, an example of Software Management AntiPattern would be the Analysis Paralysis AntiPattern when instead of trying to implement a solution and iterate over it the perfect solution is sought for too long, making the development process grind into a halt.

One other area of software development that has been analyzed through AntiPatterns is Scrum. The previously explained ScrumButs, deviations from what the Scrum Framework advises, show clear signs of AntiPatterny behavior. Eloranta et al. [12] found 14 anti-patterns in their research of the different ScrumButs in 11 different companies and 18 different teams. Some examples of the AntiPatterns they found where called: Too long Sprint, testing in next Sprint, invisible progress, business as usual and long or non-existent feedback loops.

AntiPattern Name	Unique noun or phrase that is used to refer to it. Usually it has a pejorative undertone to denote it is not desired behavior or design
Also Known As	Any other names that it might have received
Most Frequent Scale	Whether it affects the application, system or enterprise level
Refactored Solution Name	Identifying name of the refactored solution
Refactored Solution Type	Whether the refactored solution involves software, technology, process or role changes
Root Causes	List of the root causes pertinent to this AntiPattern
Unbalanced Forces	The areas that collide to force this AntiPattern for example the management of functionality, performance and complexity
Anecdotal Evidence	Optional. Often heard phrases related to the AntiPattern
Background	Optional. General background that might help understanding
General Form of this AntiPattern	The generic representation of what the AntiPattern looks like
Symptoms and Consequences	Bullet list of symptoms and consequences associated to the AntiPattern
Typical Causes	Bullet list of identified deeper causes
Known Exceptions	What are the specific occasions that this AntiPattern might not be bad
Refactored Solutions	A refactored solution for the general form of the AntiPattern presented as a list of steps to take
Variations	Optional. List of any known mayor variations
Example	The description of a concrete example showing the AntiPattern and how the refactored solution is applied to it
Related Solutions	Any relations to other AntiPatterns that might be similar and how they differ from each other

Table 2.3: Full AntiPattern template as utilized by Brown et al. [7].

3 Methodology

In this chapter we will go through the methodology used for the research done in this thesis. Section 3.1 will contain a brief explanation on the research setup, followed by section 3.2 presenting the central research questions and finally section 3.3 will contain a brief overview of how the research was performed.

3.1 Research Setup

This research was carried out in the context of a software startup. The startup operated in the field of customer satisfaction survey sending and the data analysis of the responses of the surveys. The startup had developed their own dashboard web application for the responses of the surveys and data analysis to be presented to the customers.

The codebase of the small startup presented clear signs of accumulated technical debt after over five years of development. This accrued technical debt had clearly affected the ability to make changes to the codebase smoothly. Also, inexplicable bugs that require lots of time to identify and fix were found quite frequently. In addition to the technical debt in the codebase itself there were also signs of problems and deficiencies in the project management and agile methods used. All this had clearly affected the throughput of the development team and was restraining the quick progress that the software of a startup would require to keep the initial success ongoing.

Thus the startup wanted to amortize its technical debt and make the needed changes into its processes to be able to grow without being hindered by itself.

3.2 Research Questions

The research questions of this thesis is:

- RQ1: How can code smells be used to find underlying AntiPatterns?
- RQ2: How to use AntiPatterns to identify and fix problems in a software development project with technical debt and project management issues?

The main idea is to find out how the underlying AntiPatterns that can be first sensed through code smells can be identified and how the knowledge of these AntiPatterns' existence could be used to deal with the technical debt of a codebase. AntiPatterns not being limited to just code, can possibly have a wider reach encompassing the whole software development project.

3.3 Performing the Research

The research was done through the analysis of the codebase and processes of the aforementioned startup in the form of a case study in chapter 4. The case study analyzed and identified the potential code smells found. These code smells were then analysed to find the underlying AntiPatterns that they represent, giving a potential solution for the problems and thus having a guide to reverting the technical debt accrued. If there were potential AntiPatterns not currently found in literature, new AntiPatterns were proposed.

4 Case Study

This chapter features the case study of the technical debt of the backend codebase of the software of a startup company through the use of AntiPatterns. Section 4.1 includes a overview of how the startup, its product and the development team have evolved in its five years of life. This is followed by the depiction of the features and state of the software product itself with a focus in the backend was the core of the case study in section 4.2. After that, section 4.3 features the study of the backend codebase through the code smells and AntiPatterns found in it. Finally, section 4.4 features the study of the analysis of the Scrum process used by the project through the lens of ScrumBut AntiPatterns.

4.1 Case Company

The software startup under study has been developing their current product, a customer satisfaction survey sending and survey response analysis tool for companies, for five years. Their main product is the dashboard in which the survey responses and data analysis is visualized for the customers.

The company has followed the lean startup principles to be able to develop a product that would have a good fit in the market. The product's focus pivoted several times through the years of development and particularly during the first year of development as would be expected for a technology startup. The growth of the development team and the main events of the five years of development can be seen at a glance in table 4.1.

The initial version of the software product was developed as an outsourced project by a consultancy company using purely Clojure [19] during the first 6 months of development. Clojure is a programming language that runs on the Java Virtual Machine (JVM) and thus has interoperability with Java [23]. Clojure is a very popular language among Finnish consultancy companies. Of course, this first few months saw the initial idea go through different forms, taking feedback from the first acquired and potential customers, until it somewhat arrived into what is the core of today's product.

After this, due to the niche nature of Clojure and difficulty to find a developer that would comfortably work with it, the development was continued by a lone Java developer for the

Year	Development team	Notable events
0.5	Consultancy	Initial software version developed in Clojure
1	1 developer	Transition to of business logic to Java with Clojure still serving as the endpoint layer
2	2 developers	Rapid growth of the company
3	2 developers	Lead developer hired. Beginning of growth induced problems
4	4 developers	Development team grows and growth induced problems are patched. It is decided that Clojure must be get rid of on the long term
5	6 developers	Achieved some kind of stability and there is more time for introspection, although new features continue to have a high priority

Table 4.1: Development team growth and notable events of the case company over the years.

up until the beginning of year three of development. During this time, the main backend of the product was shifted into a Clojure endpoint layer that immediately redirected to Java code where the vast majority of the logic would reside from then onward, with some original background tasks still running in Clojure. This would later become clearly a source of problems to the smoothness of change making to the codebase. This period of time was the start of the fast growth of the revenue of the company with the number of customers increasing dramatically from just a few to the tens.

This increase in customers saw an exponentially greater change in the data volumes being handled by the application. On the latter half of the third year of development a new Java developer was hired as lead developer, with the intention to build a real development team for the software product. At this point, the initial implementation started to show signs of not being able to support the load of the organic growth of the customer base. Some of the problems were solvable by upgrading the machines running the software, but some were problems caused by the not scaling design that worked well with small amounts of data but was growing increasingly slow with the new vastly greater data loads. This are the first clear signs of technical debt in the project.

During the last two years of development the software team grew little by little to have between five and six developers. Some focused more on backend development while others had more of a focus in frontend development. These years saw the use of some of the Scrum Framework principles but with several ScrumButs, like no sprints or no clear time boxing of Scrum events. These two years featured ironing out the most significant obstacles and hindrances, but the startup needed to keep moving. A couple of pivots happened again during this time frame that changed a bit the market segment that the product aimed to captivate. It was also perceived that the original codebase done with Clojure was one of the mayor obstacles in making changes, as now most of the code was written in Java, but the project was still being built through the Clojure building tool Leiningen [17]. Also due to some complex dependency conflicts the project's dependencies could not be updated easily. Another consequence from the dependencies being unupdatable, was being stuck on Java's version 8. This is why it was decided that the main backend would be transformed into a very industry standard Spring MVC project, removing the Clojure endpoint layer and using the available business logic that was already nearly completely written in Java.

4.2 Case Product

The software product was composed of the main backend and frontend (customer dashboard). In addition, there were six other services that dealt with sending the surveys, giving the responses to the main backend and performing some analysis on the responses. All of the services were hosted in Amazon Web Services (AWS) [2], Amazon's cloud computing service. In this case study we will focus on the main backend.

The main backend was built with Clojure during the first year of development. It was a simple web application with a database. It not only contained the API for the frontend to use the data in its database, but some background tasks like an importer that brought the responses of the surveys from a third-party tool used to send the surveys. Although being in the same Leiningen project, the importer was run in its own AWS Elastic Beanstalk instance through an environment variable. In addition to these two independent applications the same project contained an admin dashboard done with ClojureScript [20] (a compiler that compiles Clojure code into JavaScript). This admin dashboard and the importer code hadn't been changed after the second year of development.

As previously mentioned, most of the code written from the second year of development onward was written in Java. To achieve this the only developer at that point built a interoperation helper to be used between Clojure and Java. This helper file hadn't been changed since the second year of development too. This helper helped with the translation of objects between that that Clojure and Java understood. What this meant in practice is that every single object passed to the Clojure side was of the Java Map type. The MapHelper class was created in Java to make it easier to make the transformations of objects into Maps.

Another special characteristic of the Java code that was a consequence of the need to interoperate with Clojure was the use of classes with only static methods. This was due to Clojure being a functional programming language and thus not understanding classes and objects. This made the Java code very unidiomatic and in consequence more difficult to write and read with an idiomatic Java mindset.

The testing was clearly lackluster. The Clojure side had 23 tests in 5 files, all written during the first few months of development. These tests were run on every build, but as their coverage was very low and nothing was really modified in the Clojure code but adding and modifying endpoints of the web application, these tests would always pass. On the Java side there were 505 tests in 55 files of which 120 were marked as ignored and thus would not be run, unless the annotation used for ignoring them was removed. The Java tests were not automated tests as they could only be run manually on the IDE and were never a part of the official verification of new releases.

4.3 From Code Smells to AntiPatterns

To get a good grip on what were all the things that needed changes in the project under study an analysis of the codebase and the practices used was performed. The intention was to find the underlying AntiPatterns that could be hiding behind the very apparent code smells that had been noticed. This section starts with the static analysis of the code that surfaced many of the smells in the project in subsection 4.3.1. This is followed by a presentation of the identified code smells in subsection 4.3.2. Finally, the underlying AntiPatterns found are presented with some concrete solutions that were planned or performed in subsection 4.3.3.

4.3.1 Static Code Analysis

In many cases when working on a codebase that developers are very familiar with, developer might lose their sense of smell and thus be unable to notice the most obvious code smells. Another typical reason is, as per usual, haste and time pressure that blinds the sense of the problems to be more localized than they really are, as the attention of developers is focused on the tasks at hand and not the codebase as a whole. This is exactly what was going on the project of this case study. The need to build new features and fix critical bugs impeded developers from using some time and distance to find the underlying problems.

In this case, the use of static code analysis tools was found to be a useful way to find the most obvious and repeated problems in the codebase and through that get an idea of what areas and how things were not going optimally. The two static code analysis tools used in this case study were CodeMR [8] and IntelliJ IDEA's code inspection tool [22].

These tools have complimentary feature sets as they have very different approaches on what and how they analyze in the code. CodeMR has a more visualization centric approach that measures that quality of the code on different metrics on the Java class level while IntelliJ IDEA's code inspection tool shows concrete potential problems on the code itself line by line. Due to limitations in the tools not understanding the interoperation between Clojure and Java only the Java code was analyzed as the Clojure code was going to be removed in the short term anyways. The tools, their reports and results got from these tools are presented detail in the following sections:

CodeMR

CodeMR is a software quality tool that supports multiple languages and has integrations for multiple IDEs. It provides insight into the quality of a codebase through a very wide array of quality attributes with coupling, complexity, cohesion and size being the most notable ones. All this metrics bind together within them many related aspects of code quality. Table 4.2 shows a brief explanation of what these metrics mean and what they entail in the context of CodeMR's reports.

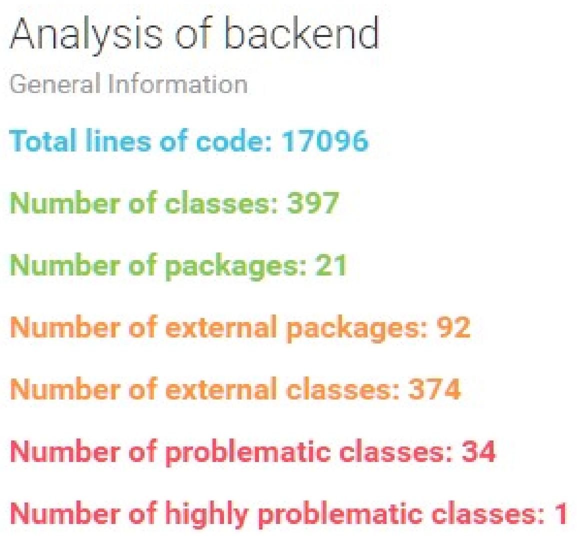
The Java code of the backend of the application under study was analyzed with the help of the CodeMR IntelliJ IDEA plugin version 2020.4.1-release-2020.2. Test files were ignored for this analysis. Figure 4.1 shows the general information of the code provided by the CodeMR report. From the list we can see that the amount of code is closing 20000 lines of code, which is nearing a point where the size itself will start hiding details and the easiness to find something a developer might be looking for. This is not alarming though, as programs will inevitably grow as time goes on, but this also requires focusing more on a good structure and design to counterweight the problems that come from the increased

Coupling	Denotes how the class in question depends of others. When coupling is high, the possibility of changes in one place might require changes elsewhere too.
Complexity	Implies that the code is difficult to understand. With high complexity the possibility of inadvertently adding bugs is increased.
Cohesion	Measures how closely the methods of a class are related to each other. Low cohesion might be a sign of a class having more than one responsibility. A lack of cohesion might make the code more difficult to maintain and test.
Size	Size one of the most repeated code smells. Be it a package, class, method or variable name, increased size decreases the readability and thus maintainability of the code.

Table 4.2: Explanation of CodeMR metrics.

size. The number of classes and packages seems reasonable, with just under 19 classes per package. From this numbers we can also derive that the average number of lines per class is just over 43, which is a very good size. The number of external packages and classes is a bit more concerning, as the number of external classes is nearly the same as the native ones. The use of external libraries is normal in today's software development to avoid reinventing the wheel when somebody has done the work already and made it available for others to use. In any case, a review of the external dependencies would be advisable based on this numbers. Finally, CodeMR had identified that 34 classes were problematic and 1 was highly problematic.

Regarding to the four main metrics, CodeMR provided the pie chart view in figure 4.2, where each chart visualizes the amount of classes with high or low values of the metric. From them we can immediately see that complexity and coupling seem to be the biggest possible problems in the codebase. According to CodeMR 35.5% of the code has the highest level of complexity and only just over a third of the code was categorized as haven medium to low complexity. This corroborates the feeling of developers that the code is quite difficult to reason about in many places. With what regards to coupling, just under half of the code was deemed to have medium to very high coupling. This was also corroborated by the experience of the developers that had found out that making changes



Analysis of backend

General Information

Total lines of code: 17096

Number of classes: 397

Number of packages: 21

Number of external packages: 92

Number of external classes: 374

Number of problematic classes: 34

Number of highly problematic classes: 1

Figure 4.1: General information of the backend CodeMR analysis.

usually cascaded to a sea of changes all over the codebase. Lack of cohesion and size seemed to fair a bit better. In both cases clearly over half of the code was identified to have medium to low levels of both metrics and no instances of very high levels were found. The experience and feelings of the developers seemed to match with what CodeMR's metrics indicated.

For a more detailed inspection of the codebase CodeMR also provided a variety of graphs and detailed metrics for each class. The graph in figure 4.3 shows all the classes analyzed in their respective packages. The graph shows the four metrics with the colors indicating complexity, the shapes indicating coupling, the size of the symbols indicating size and the cohesion visualized by the net of dependencies between the symbols. With this more detailed view it was easier to understand and pinpoint which classes would be the potential problem makers. Some of the packages have been marked with letters for further referencing.

The first thing that became clear looking at the graph was that there were a few classes that seemed to be used all over the codebase. One great example of this is the class in the package (b) which was used to do every single database query and thus needed to be used in the service classes of the codebase. Most classes in package (a) and its subpackages (a.1-7) had lots of dependencies, which might be reasonable as these packages contained all the main service classes which usually deal with the business logic of the software.

Another thing that was apparent from the graph was that although the average size of classes was just over 43 lines of code, the way the real lines of code were distributed was

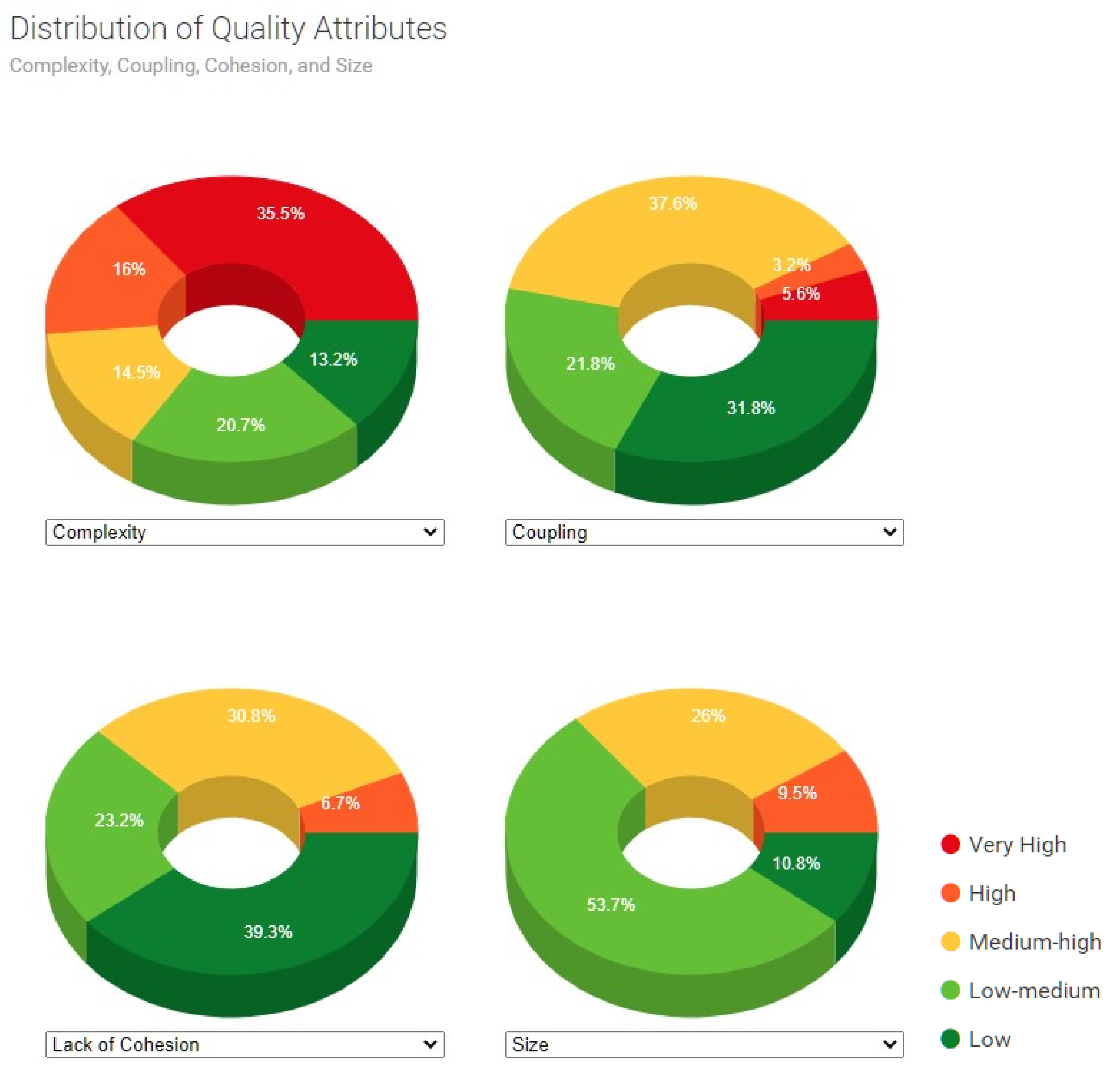


Figure 4.2: CodeMR metrics from the backend analysis.

very heterogeneous. There were a considerable number of classes that clearly were either too big or were nearing that limit. Looking into the detailed data provided by CodeMR there were 11 classes with more than 300 lines of code with the biggest class that can be seen as a big red star on package (c) having 950 lines of code.

With the exception of package (a), which had many more complex and coupled classes, most packages seemed to have one particularly complex and coupled class. This seemed to suggest that there was always a main class with too few helping classes around it.

CodeMR's analysis provided corroborating evidence that there were areas of the codebase

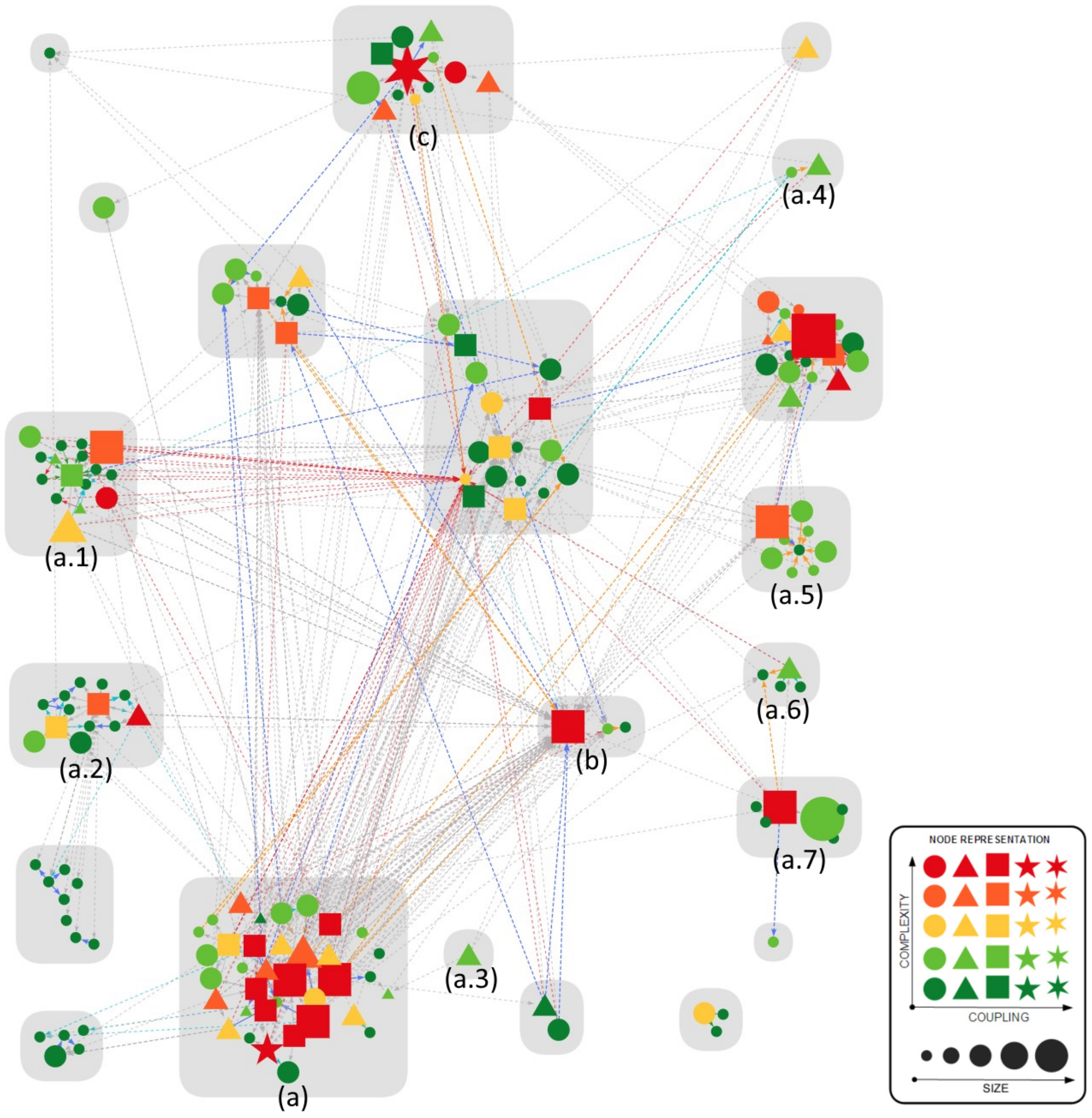


Figure 4.3: CodeMR graphic showcasing the relationship between classes in their respective Java packages.

that might have or cause problems. The classes that caused the greatest concern are listed in table 4.3 which matched the expectations of the developers.

Based on the metrics of CodeMR at least the following bad code smells from Martin Fowler's and Kent Beck's list [14] would be expected to be present: Divergent change, shotgun surgery, feature envy, insider trading, large class and data class. These would need to be verified first with the help of IntelliJ IDEA's code inspection and then further

Class Name	Coupling	Complexity	Lack of Cohesion	Size
Class A	very-high	very-high	high	high
Class B	very-high	high	medium-high	medium-high
Class C	very-high	medium-high	medium-high	medium-high
Class D	very-high	medium-high	medium-high	high
Class E	very-high	medium-high	medium-high	medium-high
Class F	very-high	medium-high	medium-high	medium-high
Class G	very-high	medium-high	medium-high	medium-high
Class H	very-high	medium-high	medium-high	low-medium
Class I	very-high	medium-high	low-medium	low-medium
Class J	very-high	medium-high	low	low-medium
Class K	high	medium-high	low	low-medium
Class L	high	medium-high	low-medium	medium-high

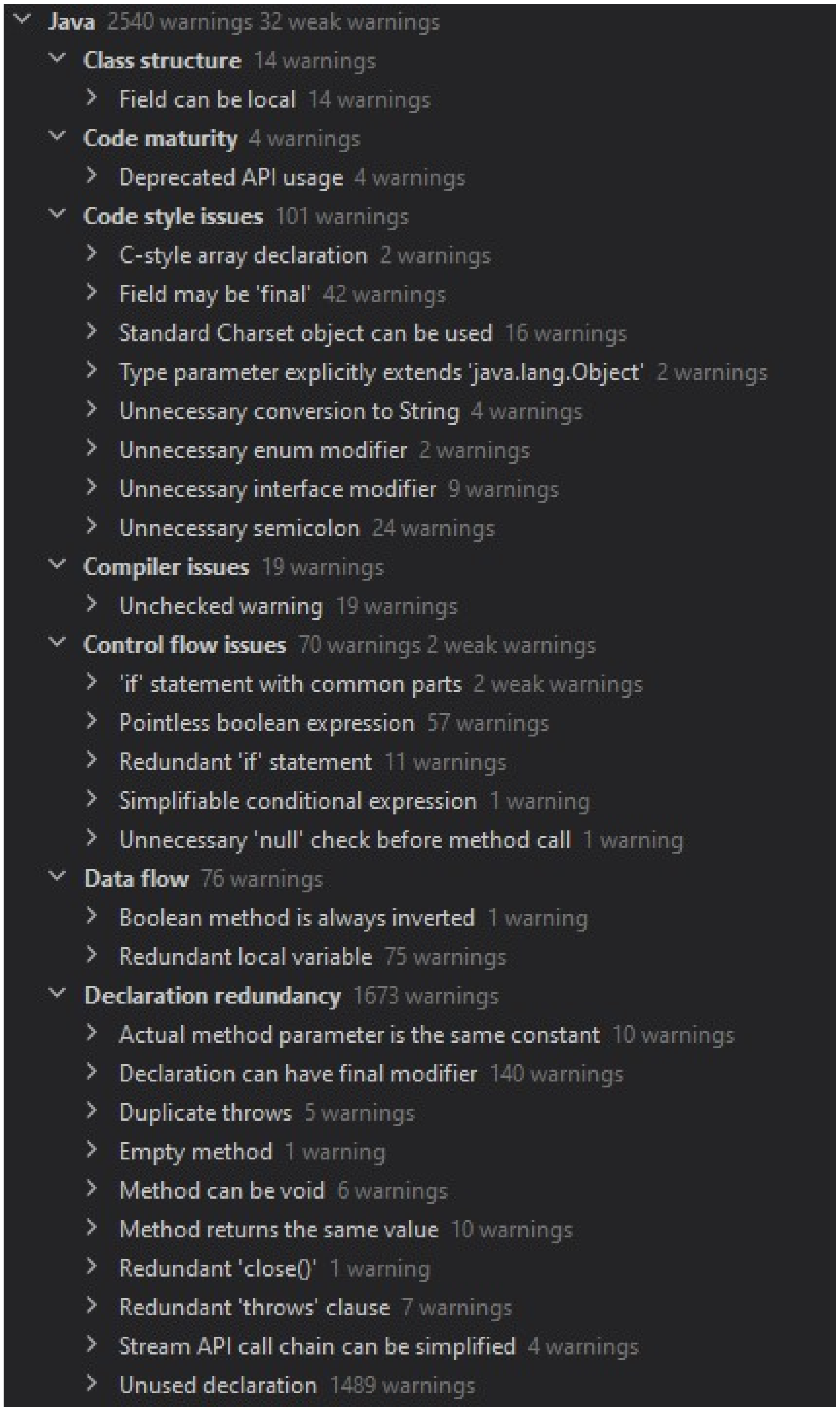
Table 4.3: Classes with high complexity, coupling, lack of cohesion or size identified by CodeMR.

by manually looking through the problematic classes themselves.

IntelliJ IDEA code inspection

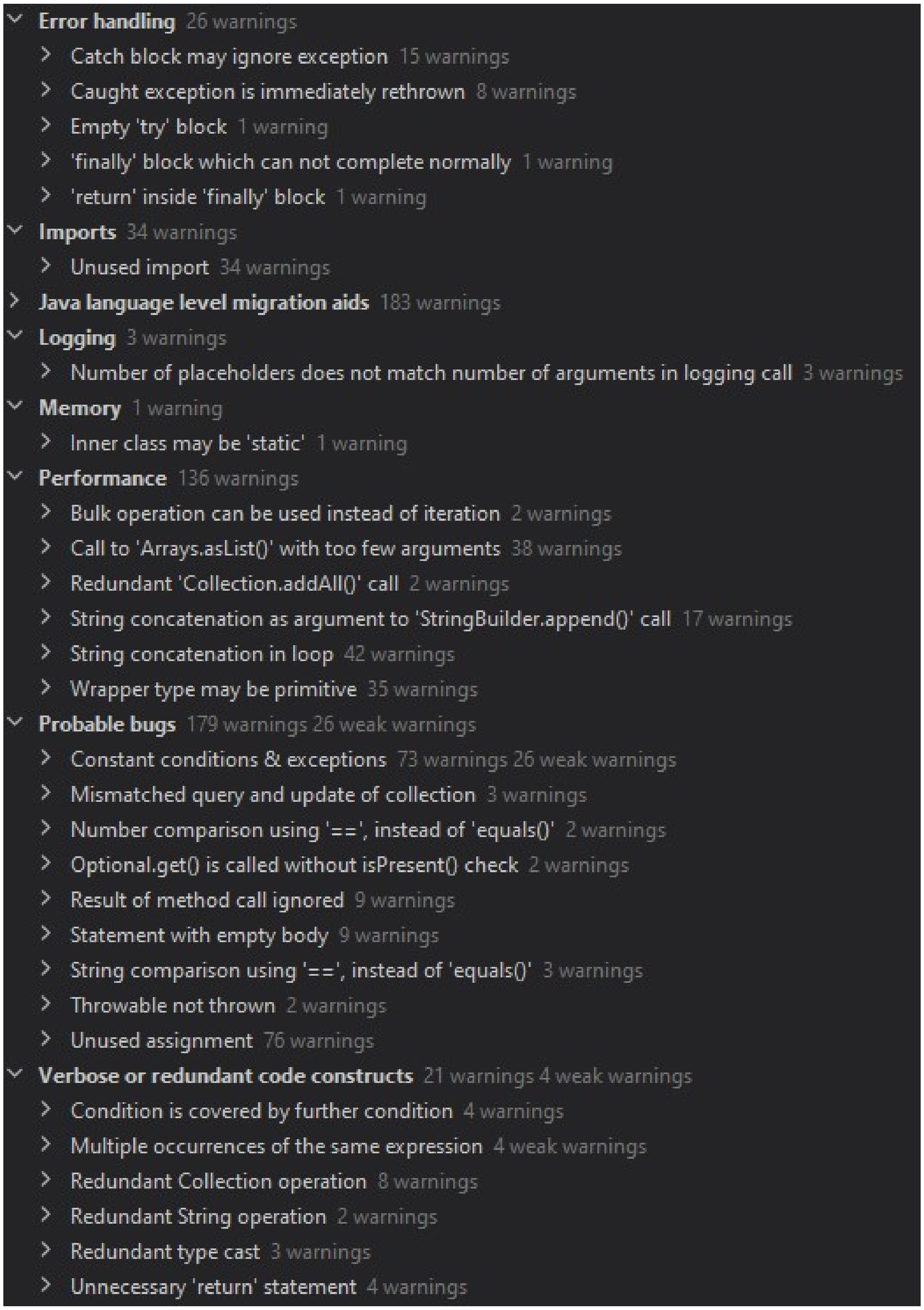
IntelliJ IDEA offers static analysis of the code in real time as the developer writes it. This feature is called code inspection and can be run for a given part of the project or the whole codebase at once. In comparison to the static code analysis made with CodeMR, IntelliJ IDEA's code inspections try to warn of concrete problems in the code line by line in the form of warnings. The tooling even offers suggested fixes that can be automatically applied by the push of a button. These are basically automated refactorings for commonly encountered code smells.

Running the inspection tools for the whole Java codebase of the backend under study on version 2020.2.2 of IntelliJ IDEA resulted in the report in figures 4.4 and 4.5. Most of the problems found were directly related to the coding style of Java which gave the most insight into the code smells present. The rest of categories were just redundant deprecation suppression instances under the general category and the use of the StopWatch class by google that has been marked as an unstable API by themselves under the JVM languages category.



```
Java 2540 warnings 32 weak warnings
  Class structure 14 warnings
    Field can be local 14 warnings
  Code maturity 4 warnings
    Deprecated API usage 4 warnings
  Code style issues 101 warnings
    C-style array declaration 2 warnings
    Field may be 'final' 42 warnings
    Standard Charset object can be used 16 warnings
    Type parameter explicitly extends 'java.lang.Object' 2 warnings
    Unnecessary conversion to String 4 warnings
    Unnecessary enum modifier 2 warnings
    Unnecessary interface modifier 9 warnings
    Unnecessary semicolon 24 warnings
  Compiler issues 19 warnings
    Unchecked warning 19 warnings
  Control flow issues 70 warnings 2 weak warnings
    'if' statement with common parts 2 weak warnings
    Pointless boolean expression 57 warnings
    Redundant 'if' statement 11 warnings
    Simplifiable conditional expression 1 warning
    Unnecessary 'null' check before method call 1 warning
  Data flow 76 warnings
    Boolean method is always inverted 1 warning
    Redundant local variable 75 warnings
  Declaration redundancy 1673 warnings
    Actual method parameter is the same constant 10 warnings
    Declaration can have final modifier 140 warnings
    Duplicate throws 5 warnings
    Empty method 1 warning
    Method can be void 6 warnings
    Method returns the same value 10 warnings
    Redundant 'close()' 1 warning
    Redundant 'throws' clause 7 warnings
    Stream API call chain can be simplified 4 warnings
    Unused declaration 1489 warnings
```

Figure 4.4: Detailed view of the Java warnings by the IntelliJ IDEA Code Inspection.



- Error handling 26 warnings
  - Catch block may ignore exception 15 warnings
  - Caught exception is immediately rethrown 8 warnings
  - Empty 'try' block 1 warning
  - 'finally' block which can not complete normally 1 warning
  - 'return' inside 'finally' block 1 warning
- Imports 34 warnings
  - Unused import 34 warnings
- Java language level migration aids 183 warnings
- Logging 3 warnings
  - Number of placeholders does not match number of arguments in logging call 3 warnings
- Memory 1 warning
  - Inner class may be 'static' 1 warning
- Performance 136 warnings
  - Bulk operation can be used instead of iteration 2 warnings
  - Call to 'Arrays.asList()' with too few arguments 38 warnings
  - Redundant 'Collection.addAll()' call 2 warnings
  - String concatenation as argument to 'StringBuilder.append()' call 17 warnings
  - String concatenation in loop 42 warnings
  - Wrapper type may be primitive 35 warnings
- Probable bugs 179 warnings 26 weak warnings
  - Constant conditions & exceptions 73 warnings 26 weak warnings
  - Mismatched query and update of collection 3 warnings
  - Number comparison using '==', instead of 'equals()' 2 warnings
  - Optional.get() is called without isPresent() check 2 warnings
  - Result of method call ignored 9 warnings
  - Statement with empty body 9 warnings
  - String comparison using '==', instead of 'equals()' 3 warnings
  - Throwable not thrown 2 warnings
  - Unused assignment 76 warnings
- Verbose or redundant code constructs 21 warnings 4 weak warnings
  - Condition is covered by further condition 4 warnings
  - Multiple occurrences of the same expression 4 weak warnings
  - Redundant Collection operation 8 warnings
  - Redundant String operation 2 warnings
  - Redundant type cast 3 warnings
  - Unnecessary 'return' statement 4 warnings

Figure 4.5: Detailed view of the Java warnings by the IntelliJ IDEA Code Inspection.

The 533 typos found by the tool under the proofreading category were for the most part caused by the use of some sort of acronyms and a few words that the dictionary just didn't recognize. Not using proper casing strategies like camelCasing also were marked as typos, due to the words fusing together in the eyes of the tool. Also, some of the warnings real typos in the names of variable and method names that just hadn't been noticed. All three types of typos found could be the source of potential problems in readability of the code.

The Java category contained understandably most of all the warnings the tool reported. The first thing that was noticed was the huge amount of declaration redundancy warnings exposed. Over half of all the warnings were of this kind. Looking closer into the subcategories, 1489 of those warnings were unused declaration warnings. This was a direct consequence of the Clojure base code being the entry point of the codebase, and thus the one calling the Java code, which the tool was unable to take into account.

In general, there were lots of different kinds of warnings but some of them were more common and seemed to be more of a repeated code smells rather than sporadic or isolated mistakes. These were the use of pointless boolean expressions, redundant local variables, string concatenations in loops, constant conditions & expressions and unused assignments. Except the use of string concatenations in loops which might negatively affect performance, the rest mostly affect the readability and intentions of the code. In the case of the constant conditions and expressions, most of them were clearly potential bugs that could end up triggering null pointer exceptions.

Based on the warning categories and a cursory look through the highlighted lines of code by the IntelliJ IDEA code inspection tool the at least the following bad code smells from Martin Fowler's and Kent Beck's list [14] would be expected to be present: Mysterious Name, mutable data and temporary field.

To compare the results of both static code analysis tools table 4.4 shows the amounts of found warnings and typos for the same classes that CodeMR found to be more problematic and which were listed in table 4.3. There was some kind of correlation between both results, as the amount of warnings was higher on the classes that CodeMR had on the top of its metrics. Bigger size in particular seemed to be the best indicator for classes with more warnings. Classes A and D were the only two classes with a high rating for size and have clearly the most warnings too. This could simply be due to the increased amount of lines of code being directly proportional with the possibility of incurring in those warning triggering behaviors.

Class Name	Warnings	Weak-warnings	Typos
Class A	61	3	27
Class B	23	0	6
Class C	16	0	17
Class D	39	4	4
Class E	25	1	34
Class F	20	1	13
Class G	22	0	1
Class H	16	0	10
Class I	20	1	15
Class J	16	0	3
Class K	10	0	4
Class L	15	0	19

Table 4.4: The amount of IntelliJ IDEA code inspection warnings and typos for the same classes as in table 4.3.

4.3.2 Identified Code Smells

The static code analysis tools gave some good insight on the possible location of the most obvious problems in the codebase under study. It also opened the veil on what kind of code smells would be expected upon further inspection. The likely code smells that could be inferred from the results of both tools were: Mysterious Name, mutable data, divergent change, shotgun surgery, feature envy, temporary field, large class, large class and data class. These findings needed to be verified with a manual look through the code to see how pervasive they might be. The manual inspection would also uncover other problems and code smells that the static code analysis tools might not have been able to identify.

The manual analysis was done to prioritize the classes presented in the previous subsection, in the order of severity and amount of warnings. The problems found in these particularly big classes would expose patterns that would be present in the rest of the codebase too. The rest of this subsection will go through some of the most interesting and representative classes on the list in a deeper manner, explaining the findings in each and finally having a look at the codebase as a whole in what respects to the code smells identified.

Code smells in Class A

Class A was the class that both static code analysis found to be the most problematic. Its purpose is to process a configuration and send an image version of the application's dashboard to the email of the designated users in the configuration. This task has four clear parts: Process the configuration, retrieving the needed dashboard, creating an image of the dashboard and sending the email with the image attached to it.

The size of the class was directly explained by it doing everything required for this functionality to work in this single class. The main function used to trigger the process was also quite big with 91 lines of code and a very complex conditional flow that went up to five levels deep. Several other functions also were clearly over 50 lines of code long. Another thing that affected the size of the file, was the usage of another class inside the main class instead of having it on its own file. This internal class was potentially even more problematic as some of its functionality wasn't coded as its own functions but that of the parent class e.g. setters for some fields of the class were outside the class itself.

One aspect of the code that was also curious was the passing of the same parameters through the whole flow of the code from one method to another. This was a clear consequence to having all the functions be static and thus void of a common context for the whole flow. The DataSource used to connect with the database was passed from the very first entry point to wherever database connections were needed. This and other not mutating parameters made the parameter lists unnecessarily long. Also, the inconsistency of the location of the DataSource parameter in the parameter lists hampered the readability of the code, being mostly the last parameter but not always.

This class is definitively smelly and it is clear that it should be divided into a few different classes with their own responsibilities. In definitive the class presented the following code smells: Long function, long parameter list, mutable data, divergent change, data clumps and large class.

Code smells in Class D

Class D is used to express the filtering of the data requested on a query. It contains a set of parameters that can then be transformed directly into SQL query parameters. It basically performs the three following tasks: Parse itself from JSON, transform into an SQL query and serialize itself into JSON.

The first thing that immediately smells in the class is the extensive use of just primitive classes to represent all the data that the class contains, although some of it would probably be better suited to some subclasses that would represent part of the contents of the class. What explains the size of this class is the use of a custom serializer and deserializer for JSON. The JSON serializer was built using just the using the Java StringBuilder over 323 lines of code while the deserialization was done with the help of the JSON in Java's (org.json maven package) JSONObject picking each key one at a time over 143 lines of code and nearly completely in just one function. For some reason the class had also annotations used to serialize and deserialize objects into JSON through the FasterXML Jackson library, which seemed redundant due to the extensive use of the custom implementation throughout the codebase.

The smells in this class were more fully encompassing giving the feeling that the whole class should be heavily refactored and take advantage of available tools that make working with JSON much simpler. The main identifiable code smells in the class were: Long function, divergent change, primitive obsession and large class.

Code smells in Class E

Class E is responsible of handling all the different possible configurations used for the dashboards. There are tens of different configurations and more are added when new features require them. The main tasks of the class are to be able to create, read, update and delete the different kinds of configurations in the database.

The class immediately starts with a list of 18 different SQL string constants written all in various different styles. Most are one-liners with varied casing but two of them are divided into several lines to improve readability. Some of these SQL statements have capitalized SQL keywords, others not and some even have a mix of styles in the same statement. The class needs specific queries for some of the different kinds of configurations that it needs to be able to handle. In addition to this the class contains 10 additional internal classes, which makes its code even more convoluted. Excepting one of the many functions in the class, most of them seem to have a reasonable size. The use of temporary variables just before returning them from functions is present in many functions.

All these signs seem to indicate that this class should potentially be divided into a generic Configuration class that would handle the base case and specific classes for those that would require additional logic, improving the readability and the size of each of those classes compared to the current implementation. The smells that his class showed signs of where: Long function, divergent change and large class.

Code smells in the whole codebase

The three examples above showed some of the particularities of the codebase under study that repeat throughout the entire codebase. Although there were just a moderate number of large classes that clearly tried to do too much by themselves there was a repeated tendency for large functions all over the codebase. Another smell that was particularly pervasive was the appearance of long parameter lists due to the architectural design that requires passing some parameters deep through the stack of function calls.

Both change related smells divergent change and shotgun surgery were present. As would be obvious there were more instances of divergent change on larger classes, as they clearly could change for a quite varied set of reasons. The previously mentioned passing of parameters through all the function calls also caused the code to be prone to shotgun surgery, needing to change all many classes and files if one parameter was needed in a deep function but could only be taken at an early stage of the function stack.

Another pattern used very often was the use of public fields, which is an instance of both the global data and mutable data smells. This design decision stemmed from how the interoperation between Java and Clojure had been designed. For it to work all fields that Clojure needed to be able to access needed to be public.

There were some utility and helper classes that held that kind of general functionality available for other classes to use, but there was also a surprisingly large amount of utility kind of functionality embedded into service classes that were then used by other service classes, presenting a clear case of feature envy. In some cases, this even created circular dependencies that were only hidden by the fact that the code is mostly composed of static functions. This would become a real problem once the classes would be transformed into a more idiomatic object-oriented pattern.

In general, the codebase gave a continuous sense of inconsistency, with things done in a variety of ways depending on the day, developer or part of the code in question. Even the code that was modified more often did not seem to be in better shape, suggesting that getting the features done was being prioritized to such extent that refactoring was not being performed in conjunction with daily development.

The code smells that the static code analysis tools had suggested that were present in

Code Smell	CodeMR	IntelliJ Idea	Manual analysis
Mysterious Name		Present	Low
Duplicated Code			Low
Long Function			High
Long Parameter List			High
Global Data			High
Mutable Data		Present	High
Divergent Change	Present		Moderate
Shotgun Surgery	Present		Moderate
Feature Envy	Present		Moderate
Data Clumps			Moderate
Primitive Obsession			Moderate
Repeated Switches			Low
Loops			Low
Lazy Element			Low
Temporary Field		Present	Low
Message Chains			Low
Insider Trading	Present		Low
Large Class	Present		Moderate
Data Class	Present		Moderate
Comments			Low

Table 4.5: Comparison between the code smells identified with the help of the static code analysis tools and the manual analysis.

the code were corroborated in the manual analysis of the code. In addition to those code smells, the manual analysis also surfaced some smells that could not be inferred from the reports of the automated tools. There is a vast number of other tools available that might have helped top find them. Table 4.5 shows the differences between what code smells were identified through the two different static analysis tools and the final manual analysis of the codebase. This table clearly indicates that the tools can help find some problems, but a manual review is able to find more subjective smells that the tools just were not capable to guess with the feature set they were equipped with.

4.3.3 Underlying AntiPatterns

Although many problems in the codebase were able to be identified through the search of code smells, the codebase seemed to have deeper problems than the surface level smells suggested. Most code smells are usually centered on one small section of the code and can appear repeatedly, but they are usually easily fixable with the use of simple code refactorings. What if code smells could be also a sign or consequence of more encompassing problems in the development and architecture of the project and thus be an indicator of certain AntiPatterns?

In search of deeper problematic patterns, the codebase was further analyzed to try to find if any known AntiPatterns or clearly specific AntiPattern to this project were present. For the shake of brevity and conciseness the custom AntiPattern template in table 4.6 was developed. This template contains the essence of the AntiPatterns needed for this discussion and a reference to the original full definition of the AntiPatterns. In addition to these, a new field was added to indicate what code smells might be associated to the given AntiPattern.

AntiPattern Name	The AntiPatterns name and variant names when necessary with the added reference to its original source
Refactored Solution Name	Identifying name of the refactored solution
Root Causes	List of the root causes pertinent to the AntiPattern
Related Code Smells	Code smells that might be symptomatic of the AntiPattern
General Description	A summarized description of the generic representation of what the AntiPattern looks like

Table 4.6: Custom AntiPattern template for the representation of the AntiPatterns in this case study.

The AntiPatterns identified are presented in the following sections starting with the AntiPattern template describing it and continued by some concrete examples of how they were present and plans and actions that were taken to start getting rid of them were applicable. Also, the association with the code smells that might signal their existence are also presented.

Lava Flow AntiPattern

AntiPattern Name: Lava Flow or Dead Code [7]
Refactored Solution Name: Architectural Configuration Management
Root Causes: Avarice, Greed, Sloth
Related Code Smells: Temporary Field
General Description: This AntiPattern is characteristic of software that go through lots of changes in direction in its beginning. Each change in direction generates new code that is then never removed when the idea behind it is abandoned. This results in sections of code that might not clearly be no longer used and that never are removed. This adds to the complexity of the code. Parts of the code might even end up being reused elsewhere, making the dependency structure and the elimination of this dead code even harder.

As already mentioned before, the codebase had undergone many changes in direction leaving code that was no longer used behind without removing it. This AntiPattern is elusive, as developers need to look for it with intention and is rarely found by coincidence. Things like unused variable and temporary fields might be an indication of its existence, but in most cases the code will look just normal, but it just implements functionality that is no longer used.

In the case of a web application like the one under study knowing if the endpoints of the application are actively being used can't really be verified from its code alone. To be able to find out if some endpoints are not being used, and subsequently the code that supports that endpoint, an external analysis must be done. This is exactly what was done to the backend under study. The code of all the client services that used its endpoints were searched for the usage of the endpoints. In addition to this the AWS CloudWatch Logs Insights [1] tool was used to count the amount of http requests done to the endpoints from the http logs of the production servers. The result of this analysis was that 77 endpoints from over 350 were no longer used at all and were thus removed, which resulted in the deletion of 2690 lines of dead code.

Another symptom that might indicate of the presence of this AntiPattern is the existence of columns and tables in the database that are either completely empty or always have the exact same value. Some instances of this were also found and were either immediately removed if no refactoring was needed or were added to the backlog for later removal.

Functional Decomposition AntiPattern

AntiPattern Name: Functional Decomposition or No Object-Oriented Pattern [7]

Refactored Solution Name: Object-Oriented Reengineering

Root Causes: Avarice, Greed, Sloth

Related Code Smells: Long Parameter List, Global Data, Mutable Data, Data Clumps, Data Class

General Description: This AntiPattern is in its simplest term the misuse of an object-oriented language as it were a functional or structural language. This can make the code very convoluted. object-oriented programming has lots of beneficial design patterns which this kind of code does not take advantage of. Depending on how widespread this AntiPattern is the reengineering work needed to fix it might be overwhelming and must be done in increments.

Due to the Clojure roots of the project, the Java code didn't follow the usual object-oriented structure of classes that are instantiated into objects. The Java code relied on the use of static methods that could be run without creating a new instance of most classes. In addition of the fact that most methods were static, smells like long parameter lists, global data and mutable data were clear indicators of the presence of this AntiPattern.

As this was a very overarching AntiPattern in the codebase it would be no easy task to convert the whole codebase into beautiful object-oriented code. As it had already been decided that the original Clojure code would be eliminated, the Java code would be refactored to follow the Java Spring inversion of control patterns as it was migrated to a new Java Spring MVC based repository. When migrating part of the code it was noticed that the current paradigm had made circular dependencies in the code possible that would appear when instantiating the classes instead of using their methods statically. Another thing that this refactoring would alleviate would be the passing of parameters through the functions, as services containing the utility and information needed would be injectable to the objects that would need them.

Spaghetti Code AntiPattern

AntiPattern Name: Spaghetti Code [7]

Refactored Solution Name: Software Refactoring, Code Cleanup

Root Causes: Ignorance, Sloth

Related Code Smells: Long Function, Divergent Change, Shotgun Surgery, Loops, Message Chains, Insider Trading, Large Class

General Description: Spaghetti code needs no introduction, being the most famous of all AntiPatterns and known even to developers that aren't aware of the concept of AntiPatterns itself. This AntiPattern depicts code without a logical structure. This kind of code will be difficult to understand even to the original developer if it has not been in its focus even for a few weeks. The code might contain large classes, with convoluted flows in a single function. Some classes might also contain clearly out of place functionality that then other classes might use, making the dependencies between classes very difficult to understand.

The long classes and functions found through the static code analysis were clear examples of spaghetti code. These classes had one main entry point function that would then go through a very convoluted flow of conditionals and function jumps that made the code not very pleasant to read and thus difficult to understand. Although these classes were clearly in need of refactoring, rarely did the developers have the time to be able to refactor them as they made changes to them.

Another instance of this AntiPattern was the mislocation of functionality in classes were they clearly did not belong. The cause for this kind of pattern was on the minimization of the use of classes, having all the functionality needed for the original class in that singular class, although parts of it would fit better in another service or helper class that could then be used by other classes too. This would have been fine in theory if once the functionality that had been embedded into the class was needed in other classes would then have been refactored out of it. This was not the case creating very weird dependency graphs between classes that made no sense.

To be able to get rid of all the spaghetti code a change in the mentality of when refactoring should be done was needed. The only way to deal with it was to understand that refactoring should just be part of normal development workflow. Instead of just forcing new changes to code that clearly is in need of refactoring, the code should be refactored first to make the change easier to implement and clearer to understand for those that would follow. This incremental approach could then be complemented with bigger refactoring sessions when time was available, or the class would have reached such a good state that just some finalizing touches would be needed.

Reinvent the Wheel AntiPattern

AntiPattern Name: Reinvent the Wheel, Design in a Vacuum or Greenfield System [7]
Refactored Solution Name: Architecture Mining
Root Causes: Pride, Ignorance
Related Code Smells: Duplicated Code
General Description: This AntiPattern refers to the lack of reuse of code from the codebase itself or of external available libraries. It usually stems from the lack of knowledge of what has previously been done or the believe of some developers that think that they could do it better themselves.

This is the only AntiPattern that really did not have any code smells that would even slightly indicate of its presence. The duplicate code would probably be a good indicator of this, but it was not present in the codebase under study. To be able to notice this AntiPattern a logical understanding of what was readily available either in the codebase itself or in open external libraries.

The first example of wheel reinvention in the codebase was the multiple implementations of csv file parsing and writing. Although an external library had been added for this purpose, only some of the instances where handling csv files was required used this external library. Others created their own non-trivial specific implementations that were not reusable by other classes. This was so widespread and low risk that it would be harmonized as a low priority task sometime later.

Another instance of wheel reinvention was a coincidentally similar case where JSON serialization and deserialization was done by hand, instead of relying on the available libraries that could handle this. JSON is a much more complex format than that of a csv file and thus its manual implementation is much riskier. In addition to this the use of the FasterXML/jackson library was used widely enough that it would have been easier to harmonize than that of the csv file handling. The updates would be done in an incremental manner, once the manual JSON handlers would need updates, as it would be much easier and concise to handle through the use of the external library.

4.4 ScrumBut AntiPatterns

Software development is not isolated to the act of writing code. There are processes that control what, when, how and by whom some feature or requirement is developed. Analyzing the code in a vacuum might thus be insufficient, having the same problems resurface in code once fixed the first time, as their source is really elsewhere. The development team of the codebase under study followed a custom variant of the Scrum Framework as many software development teams do.

In an attempt to find if the changes made to the Scrum Framework might be the source of any problems, the team's processes were analyzed through the lens of the ScrumBut AntiPatterns identified by Veli-Pekka Eloranta et al. [12]. For conciseness the AntiPattern template used in the paper will follow the structure in table 4.7. The most notable difference from a normal AntiPattern template is that What is usually the refactored solution of the AntiPattern is in the case of ScrumBut AntiPatterns the Scrum recommendation.

AntiPattern Name	The AntiPatterns name with the added reference to its original source
Scrum Recommendation	The Scrum recommendation related to it
General Description	A summarized description of the generic representation of what the AntiPattern looks like

Table 4.7: Custom AntiPattern template for the representation of the ScrumBut AntiPatterns in this case study.

The ScrumBut AntiPatterns identified are presented in the following sections starting with the ScrumBut AntiPattern template describing it, followed by the concrete example of the behavior in the development team. This is followed by possible corrections that were done or why the AntiPatterny behavior was tolerable in the context of the project. A couple of possible novel ScrumBut AntiPatterns were identified among the practices of the team.

No Sprints ScrumBut AntiPattern

AntiPattern Name: No Sprints (novel)
Scrum Recommendation: 2 weeks sprints
General Description: The development team does not follow the Scrum Sprint structure, making it more difficult for developers to focus the work to be done in any given moment. This can also lead to tasks not being split into smaller pieces as they do not need to fit into a given sprint, making some work more difficult to share the workload of the given task.

Not having Sprints at all is probably not unique to this project but rarer than for example the too long Sprint AntiPattern [12] from which this AntiPattern would be considered to be a variant. The project under study didn't use the concept of sprints at all. A backlog grooming and week (instead of sprint) planning session was held every Monday morning with a demo on Friday evenings. Daily meetings were held every morning and a retrospective session was held once a month.

The lack of sprints clearly had an effect on the quality of the user stories in the backlog, that were in many cases too big to fit in a week but weren't divided into smaller pieces as it was not strictly necessary. This affected the shareability of tasks that would have otherwise been able to develop in parallel.

This was slightly improved with the introduction of a backlog management software that replaced the old spreadsheet-based backlog. The visual aspect and the tools provided by it made it easier to communicate about the tasks at hand and their splitting into smaller tasks. Two-week sprints should probably be at least tested to see if the time restriction would improve the completion of tasks in a more predictable time frame.

Invisible Progress ScrumBut AntiPattern

AntiPattern Name: Invisible Progress [12]

Scrum Recommendation: Progress should be continuously be visualized with burn-down charts

General Description: Either the progress is hidden from the development team or no representation of the progress is produced or in some cases where there is a burn-down chart, unfinished tasks are marked as done. This hinders the awareness of the team on their capabilities and if they should need to make some kind of correction to be able to deliver results faster.

The team originally used a rudimentary spreadsheet backlog which listed the tasks and requirements in order of priority. This however was not a very good tool as it was very difficult to read, search and reason about. The only measure of progress was the amount of lines that had been done as there were very few tasks that were really estimated. It was also difficult to ascertain which tasks to count at any given moment, as the input of the date a task was started and done had to be inputted manually and was not always remembered to be inputted.

This was somewhat alleviated by the previously mentioned backlog management software that replaced the old spreadsheet-based backlog. It made it easier to mark when a task was started, when it was done and even when it was finally in production. Also, the introduction of the use of epics, which in the context of the tool was a group of tasks, brought the first burn-down charts in their context. The estimation of work of the tasks is still very low outside of the tasks that are part of epics and should be improved so that a possible time framed burn-down chart of all the tasks would be generatable.

Testing is Not Required ScrumBut AntiPattern

AntiPattern Name: Testing is Not Required Sprint (novel)

Scrum Recommendation: Testing should be done in conjunction with the implementing code

General Description: Testing is a fundamental part of software development in general but agile software development in particular. Sadly, sometimes it is ignored and seen as a nuisance. Not having tests will make the code more brittle and prone to future failure. Writing tests later is always more difficult and thus makes it even more important for the tests to be written in tandem with the code they test.

This AntiPattern is a variant of the testing in next sprint AntiPattern [12] that considers the case when tests are not really required at all. This is the case of this project where testing was never a priority on its early development and lead to it becoming the permanent state of affairs. Little by little the project grew to such an extent that going back and creating tests for everything would not be feasible. The addition of Java added another layer of complication to it that would have required modifying the Leiningen compilation script to run Java tests too and was never done. Thus, most of the codebase has no tests at all or has tests but that can only be run manually.

Everybody in the team understands their importance but have not had the resources to fix the situation. Testing the code with its current structure and architecture is innately difficult but a known benefit of testing is that when testing code becomes more testable. Now with the removal of the Clojure code in sight, it is important to remember the importance of testing and not repeating the mistakes from before.

Flexible Time-boxing of Scrum Events ScrumBut AntiPattern

AntiPattern Name: Flexible Time-boxing of Scrum Events (novel)

Scrum Recommendation: All Scrum events are time-boxed to maximize efficiency

General Description: Scrum events do not adhere to a fixed time-box. The daily is probably the most affected event due to its intended short duration, but it might affect other events too. Every daily lasts more than the agreed time, as the topics of the meeting are not kept on what it is for. This usually ends up wasting the time of most of the team, as two people start for instance going into the details of a given task's implementation.

The project's daily meeting was time-boxed to 15 minutes but was rarely kept under this constraint. Most times the meeting would contain one or several sections of concrete implementation talk of a given task of a singular developer.

Daily meetings are supposed to be a fast checkup of everyone's ongoing work, to briefly check where at which stage each one is and so that possible blockers can be removed. Any possible further talk needed should be set aside of the meeting and dealt with in smaller groups after the daily is over so that the rest of the team can keep working on their own tasks.

5 Conclusions

Understanding AntiPatterns has the potential to help projects alleviate the technical debt that they have accrued. Sadly, the concept of AntiPatterns is not as widely known or used as that of design patterns. This thesis presented a case study were the codebase of a startup that had clear signs of technical debt was analyzed and improved through the identification and use of the underlying AntiPatterns it presented.

The results addressing the research questions of this thesis showed that in the context of the case study:

- Areas of code with code smells did often have underlying AntiPatterns. Each AntiPattern usually had its own set of code smells that were related to it but there was no direct causation the other way around, rather code smells served as indicators of the need for deeper inspection of the underlying behavior.
- The use of AntiPatterns to analyze and fix the problems in the codebase was very helpful. Looking for AntiPatterns instead of individual problems made it easy to find more overarching patterns and thus using the refactored solution of the AntiPattern enabled planning and making fixes that would improve the codebase's structure and readability as a whole. Using AntiPatterns to analysis the Scrum methodologies used in the project proved to be also a good fit, as it become clear why certain behavior was not desirable and what was the Scrum recommendation towards which to work to, improving the overall performance of the Scrum team.

Another observation and confirmation made was the added difficulty that not having tests adds to the possibility to deal with technical debt in a codebase. Refactoring, the main tool used to improve code without changing its behavior, relies heavily on having tests that can confirm that regressions are not introduced while trying to fix other problems. Adding tests is thus always the first step towards getting rid of the accrued technical debt.

In some instances, some AntiPatterns were continuously reintroduced even after having realized their existence and its active removal in other areas of the code. It is thus of the utmost importance to understand AntiPatterns, particularly those that have already been previously identified, and have good peer review practices that make it possible to catch them before they end up in the production code and before not too long the project is back on square one.

One line of further inquiry that could be beneficial would be a more comprehensive study of static code analysis tools that could aid in the identification of AntiPatterns. Better tooling becomes even more important as the size of the codebase being analyzed increases.

Further corroboration of the central findings in this thesis should also be sought in the context of different kinds of software development projects and types of companies. Is the use of AntiPatterns in technical debt management feasible on projects of all sizes and types? Would it be beneficial to have some kind of AntiPattern repository from previous projects with concrete examples of the refactored solutions implemented? These questions need answering before the results of this thesis could be widely generalized.

Bibliography

[1] *Analyzing Log Data with CloudWatch Logs Insights*. 2020. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/AnalyzingLogData.html (visited on 11/22/2020).

[2] *AWS, Amazon Web Services*. URL: https://aws.amazon.com/ (visited on 11/24/2020).

[3] S. S. Bajwa, X. Wang, A. N. Duc, and P. Abrahamsson. "“Failures” to be celebrated: an analysis of major pivots of software startups". In: *Empirical Software Engineering* 22.5 (2017), pp. 2373–2408.

[4] K. Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherl, and D. Thomas. *Manifesto for Agile Software Development*. 2001. URL: https://agilemanifesto.org/ (visited on 03/10/2020).

[6] K. Bennett. "Legacy systems: Coping with success". In: *IEEE Software* 12.1 (1995), pp. 19–23.

[7] W. J. Brown. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998. ISBN: 9780471197133.

[8] *CodeMR*. URL: https://www.codemr.co.uk/ (visited on 11/17/2020).

[9] K. Conboy. "Agility from first principles: Reconstructing the concept of agility in information systems development". In: *Information systems research* 20.3 (2009), pp. 329–354.

[10] W. Cunningham. "The WyCash portfolio management system". In: *ACM SIGPLAN OOPS Messenger* 4.2 (1992), pp. 29–30.

[11] C. Ebert and M. Paasivaara. "Scaling agile". In: *IEEE Software* 34.6 (2017), pp. 98–103.

[12] V.-P. Eloranta, K. Koskimies, and T. Mikkonen. "Exploring ScrumBut—An empirical study of Scrum anti-patterns". In: *Information and Software Technology* 74 (2016), pp. 194–203.

[13] M. Feathers. *Working Effectively with Legacy Code*. Pearson Education, 2004. ISBN: 9780132931755.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2019. ISBN: 9780134757599.

[15] M. Fowler. *CodeSmell*. 2006. URL: https://martinfowler.com/bliki/CodeSmell.html (visited on 11/11/2020).

[16] M. Fowler. *Technical Debt Quadrant*. 2009. URL: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html (visited on 10/24/2020).

[17] P. Hagelberg. *Leiningen*. URL: https://leiningen.org/ (visited on 11/17/2020).

[18] T. Hall, S. Beecham, J. Verner, and D. Wilson. "The impact of staff turnover on software projects: the importance of understanding what makes software practitioners tick". In: *Proceedings of the 2008 ACM SIGMIS CPR conference on Computer personnel doctoral consortium and research*. 2008, pp. 30–39.

[19] R. Hickey. *Clojure*. URL: https://clojure.org/ (visited on 11/17/2020).

[20] R. Hickey. *ClojureScript*. URL: https://clojurescript.org/ (visited on 11/17/2020).

[21] *How DropBox Started As A Minimal Viable Product*. 2011. URL: https://techcrunch.com/2011/10/19/dropbox-minimal-viable-product/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAL-SHMztWEM4og0RR698Z8iKQtP-ArGF9km2d--aWhdGOLERH47a-UH2Ks2qvDXGj0o8YSlrO-45oEzuvYjKqYj0uaKDqRCRsUfOSBwy2VFz2h9mCOfrpG04viN1L1znYc-xW8McBFzfpYFxYsGxPlZVJn7kv
NiRyateWJtPX (visited on 10/24/2020).

[22] *IntelliJ IDEA Code Inspections*. 2020. URL: https://www.jetbrains.com/help/idea/code-inspection.html (visited on 11/17/2020).

[23] *Java*. URL: https://www.java.com/en/ (visited on 11/24/2020).

[24] E. Lopian. *Defining Legacy Code*. 2018. URL: https://dzone.com/articles/defining-legacy-code (visited on 10/24/2020).

[25] P. Louridas. "Static code analysis". In: *IEEE Software* 23.4 (2006), pp. 58–61.

[26] S. Ramlall. "A review of employee motivation theories and their implications for employee retention within organizations". In: *Journal of American academy of business* 5.1/2 (2004), pp. 52–63.

[27] E. Ries. *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Currency, 2011.

[28] N. Rios, R. O. Spínola, M. Mendonça, and C. Seaman. "Supporting analysis of technical debt causes and effects with cross-company probabilistic cause-effect diagrams". In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019, pp. 3–12.

[29] K. Schwaber. "Scrum development process". In: Business object design and implementation. Springer, 1997, pp. 117–134.

[30] K. Schwaber and J. Sutherland. *The Scrum Guide*. 2017. URL: https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf (visited on 10/23/2020).

[31] H. Terho, S. Suonsyrjä, A. Karisalo, and T. Mikkonen. "Ways to cross the rubicon: pivoting in software startups". In: *International Conference on Product-Focused Software Process Improvement*. Springer, 2015, pp. 555–568.

[32] *The 14th Annual State of Agile Report*. 2020. URL: https://stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494 (visited on 10/23/2020).

[33] E. Tom, A. Aurum, and R. Vidgen. "An exploration of technical debt". In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516.

[34] C. Webb. *Agile Landscape*. 2016. URL: https://www.slideshare.net/ChrisWebb6/last-conference-2016-agile-landscape-presentation-v1 (visited on 10/21/2020).

[35] *What is ScrumBut*. URL: https://www.scrum.org/resources/what-scrumbut (visited on 10/23/2020).

[36] J. P. Womack, D. T. Jones, and D. Roos. *The machine that changed the world: The story of lean production–Toyota's secret weapon in the global car wars that is now revolutionizing world industry*. Simon and Schuster, 2007.

[37] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. "How open source projects use static code analysis tools in continuous integration pipelines". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 334–344.

[38] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman. "A case study on effectively identifying technical debt". In: *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. 2013, pp. 42–47.