Joonas Uusi-Autti

Exploring the Prevalence and Common Types of Technical Debt in a CRM Software

Master's Thesis in Information Technology

March 24, 2024

University of Jyväskylä

Faculty of Information Technology

Author: Joonas Uusi-Autti

Contact information: jopeuusi@student.jyu.fi

Supervisor: Tommi Mikkonen

Title: Exploring the Prevalence and Common Types of Technical Debt in a CRM Software

Työn nimi: Teknisen velan yleisyys ja yleisimmät tyypit asiakkuuksienhallintaohjelmistossa

Project: Master's Thesis

Study line: Software and telecommuunications technology (Ohjelmisto ja tietoliikennetekniikka)

Page count: 51+0

Abstract: This master's thesis researched the prevalence of the technical debt and its common types in a customer relationship management software. The software is used by a large company and it has been built solely for this companys needs. The software has been running for over ten years and the prevalnce of the technical debt has been unknown but it is presumed that it exists.

The research was conducted with a case study model, utilixing "code smells", which are useful to gather information about technical debt and its types. The data was gathered using two different tools. First one was the IDE's (integrated development environment) own code inspection tool and the second one was PMD and it was decided using previous research and downloaded from the IDE's marketplace.

To examine the code smells a frame and inclusion/exclusion critrias were built. With the help of these frames, it was possible to examine that the software has a medium level technical debt and most common types were linked to the performance parts of the software.

Keywords: Technical debt, CRM, code smells

Suomenkielinen tiivistelmä: Tässä pro-gradu tutkielmassa tutkittiin teknisen velan yleisyyttä ja sen yleisimpiä muotoja asiakkuuksienhallintaohjelmistossa. Tarkastelun kohteena oleva

ohjelmisto on ollut käytössä suurella yrityksellä ja se on rakennettu juuri tämän yrityksen tarpeisiin. Ohjelmisto on ollut käytössä yli kymmenen vuotta, mutta teknisen velan yleisyys ei ole ollut tiedossa. Teknistä velkaa oletetaan kuitenkin olevan olemassa kyseisessä ohjelmistossa.

Tutkimuksessa käytettiin tapaustutkimus -menetelmää, hyödyntäen "koodihajuja", joilla voidaan saada tietoa teknisestä velasta yleisesti ja sen yleisimmistä muodoista. Teknistä velkaa tutkittiin kahdella eri työkalulla, jotka olivat IDE:n (integrated development environment) oma koodihajutyökalu ja IDE:n marketplacesta saatava koodihajutyökalu PMD. Jälkimmäiseen työkaluun päädyttiin olemassa olevan tutkimuksen perusteella.

Koodihajujen tutkimiseen rakennettiin raamit, jotka sisällyttivät tietyin kriteerein koodihajuja. Näiden raamien avulla pystyttiin selvittämään, että ohjelmistosta löytyy kohtalainen määrä teknistä velkaa sekä yleisimpien teknisen velan tyyppien liittyvän suorituskykyyn.

Avainsanat: Tekninen velka, CRM, koodihajut

List of Figures

List of Tables

Contents

1 Introduction

In the ever-evolving landscape of software development, organizations try to maintain a competitive edge through innovative and efficient software solutions. The development of a large-scale software systems often involves making choices between practicality and long-term sustainability (Timbó 2023). If these choices are not carefully managed, they can accumulate in a form of hidden cost, known as *Technical Debt*.

Technical debt represents the compromise between delivering software quickly to meet instant business needs and investing the time and effort required to ensure long-term maintainability, scalability and quality. Much like financial debt, technical debt collects interest over time. It hinders development speed and increases the risk of software defects, inefficiencies and diminished customer satisfaction. (Timbó 2023).

This thesis studies technical debt within the context of a large company's Customer Relationship Management (CRM) software. CRM systems are tools that facilitate customer interactions, sales and overall business success (Chalmeta July 2006). In large organizations, CRM software often evolves over time to accommodate changing business requirements, leading to the potential accumulation of technical debt.

The primary objective of this thesis is to conduct a case study, focusing on the CRM software. Within this context, the following specific objectives will be pursued with the help of code smells: assess the prevalence of technical debt within the CRM software and identify the common types of technical debt.

The remainder of this thesis is organized as follows: **Chapter 2** provides a literature review, presenting an overview of Technical Debt, CRM, agile software development and code smells. **Chapter 3** describes the research methodology, detailing the case study approach, data collection methods and analysis techniques. **Chapter 4** presents the background for the case study. **Chapter 5** presents the case study findings, including the prevalence of technical debt and common types identified. **Chapter 6** discusses the implications of the findings and proposes recommendations for addressing and managing technical debt in large-scale CRM software.

2 Literature review

2.1 Agile software development

There have been several different software development processes during the short lifetime of a agile software development. Some of them have survived and stayed, while many of them are already in the past.

Agile software development is iterative and flexible approach to software development that focuses on collaboration, customer feedback and the ability to adapt to changing requirements. It aims to deliver functional software increments in short cycles, known as iterations or sprints, while maintaining a focus on customer needs and continuous improvement. The roots of the agile software development can be traced back to 1970s and 1980s when iterative and incremental development approaches started gaining popularity. The *"Agile"* term and the principles came into existence in 2001 with publication of the Agile Manifesto. (Beck et al. 2001).

The four main values for agile software development are:

1. Individuals and interactions over processes and tools. The agile movement puts effort to the relationship of the software developers and to the close team work as opposed to the institutionalized processes. (Abrahamsson et al. 2002).

2. Working software over comprehensive documentation. The main focus of the software team is to continuously deliver working software. New releases are produced at a high pace (even daily). The developers have to keep the code base clean, straightforward and from a technical point of view, as up-to-date as possible. This decreases the documentation burden to a reasonable level. (Abrahamsson et al. 2002).

3. Customer collaboration over contract negotiation. Values the relationship between the developers and the clients over strict contracts, even though the importance of well-documented contracts is essential and grows at the same pace as the project. The negotiation process should have main focus on achieving and maintaining a viable relationship. Ag-

ile software development is focused on delivering business value immediately and by that, decreasing the risk of contract issues. (Abrahamsson et al. 2002).

4. Responding to change over following a plan. The development group, which contains both software developers and customer representatives should be compenent enough to consider possible adjustment needs during the whole development process. This means that the participants are prepared to make needed changes and that existing contracts are formed in a way that supports and allows these modifications. (Abrahamsson et al. 2002).

With the ongoing pressure to deliver new features with tight time schedules, the agile methods can create technical debt in the project. And if it is not handled efficiently, it can accumulate and make the software error prone and diminish performance in the software. (Philippe Kruchten November 2012).

2.2 Scrum

Scrum is one of the most popular and widely adopted frameworks in agile software development and it was first introduced in 1997 (Hron 2018). Scrum increases flexibility and produces a system which is responsive, considering the initial and additional requirements in the development process (Schwaber 1997). Scrum splits the development process into iterations, called sprints. In Scrum, the customer is the product owner and requirements are represented in the form of user stories. The backlog of the product is updated continuously and serves also as a documentation.

Software product releases are planned based on the variables in Table 1 below. These variables form the initial plan, can change during the project and has to be taken into account in a successful development methodology.

Normally Scrum process consists of three phases: pre-game phase, development phase and post-game phase.

Pre-game phase. The pre-game phase includes planning and architecure. *Planning* includes definition of the system, product backlog, requirements originating from different teams (eg. sales, marketing). Requirements are prioritized and given a time period. In the *Architecture*,

Variable	Meaning
Customer requirements	How the current system needs enhancing
Time pressure	what time frame is required to gain a competitive advantage
Competition	What is the competition up to and what is required to best them
Quality	What is the required quality, given the above variables
Vision	What changes are required at this stage to fulfill the system vision
Resource	What staff and funding are available

Table 1: Software product releases in SCRUM. (Schwaber 1997)

the design of the system, including the architecture is planned based on the backlog. If the system needs enhancements, the backlog items are identified with the problems they may cause. (Abrahamsson et al. 2002).

Development phase. The development phase, also known as the *game phase* is the agile part of the Scrum process. The development phase acts as a black box, where every possible outcome is expected. The different environmental and technical variables which may change during the process are observed and controlled during the sprints development phase. Rather than only taking these into consideration in the beginning of the project, Scrum process controls them constantly in order to be flexible and adaptative. (Abrahamsson et al. 2002).

Post-game phase. The post-game phase includes the release of the product. This phase is achieved when the agreement has been made that the environmental variables such as the requirements are completed. The post-game phase also includes the documentation, integrations and testing for the system. (Abrahamsson et al. 2002).

In Scrum, technical debt is almost always inevitable phenomenom. And it usually involves the process on how to manage it. In most cases, it is unclear who is responsible for it. Is it product owner, the team or who? But it is a concept that should be taken into consideration. (Frederico Oliveira 2015).

2.3 Technical debt

Sometimes when using these aforementioned agile methods, or other similar processes, there is a possibility to gain *technical debt*. The metaphor of technical debt in software development was first introduced by Cunningham (October 1992). The metaphor was created to acknowledge and think about the problem that when something is done "quick and dirty", it sets technical debt on the project, much like financial debt. Through the years, it has been refined and extended by numerous other researchers (Philippe Kruchten November 2012). The metaphor of technical debt was not in massive use in it's early days but from 2000s and especially since 2010, it has been used to define basically all kinds of software flaws. Even though it has more spesific meaning (Kruchten et al. 2013).

One of the more recent definitions of technical debt is from Steve McConnell, *A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)*. (Kruchten et al. 2013).

The landscape of technical debt consist of visible and mostly invisible parts and it is divided into Evolution issues (evolvability) and Quality issues (maintainability). Both visible issues (evolution and quality) are linked to the mostly invisible issues. Evolution issues consists of new features and additional functionality, while quality issues consists of defects and low external quality. Mostly invisible part is divided into two subsections, *architecture* and *code* as seen in Table 2.

Architecture	**Code**
Architectural debt	Low internal quality
Structural debt	Code complexity
Test debt	Code Smells
Documentation debt	Coding style violations

Table 2: Mostly invisible parts of technical debt. (Philippe Kruchten November 2012)

Most agree on the fact that the cause of technical debt is schedule pressure and it does not matter if the teams's working method is agile and iterative or if it is more old-fashioned

waterfall-like process. In the end it comes down to the choices made along the way. The choices in coding include choosing expedient or suboptimal solution in the present. It is often the result of taking shortcuts or making trade-offs to meet the deadlines. While technical debt is not just bad coding, the tools for detecting technical debt are for analyzing the codebase. When using these static code analyzers there lies a danger of undetecting such technical debt which is not detectable by these tools, eg. structural or architectural debt. (Philippe Kruchten November 2012).

Technical debt is not a sudden one-time phenomenom and developers and stakeholders have to remember that addressing technical debt is an ongoing process. It is essential to find a balance between delivering new features and maintaining a healthy codebase, even though scheduling pressure is always existing.

2.4 Code smells

Code smells are a term used in software development to describe certain patterns or characteristics in the source code that indicate potential design or implementation issues. They are called "code smells" because they are indicative of underlying problems in the codebase, much like bad odors can indicate problems in the physical world. Code smells do not necessarily indicate bugs, but they do suggest that the code could be improved to enhance maintainability, readability and extensibility. (Team 2023-07-20).

Code smells are essential to identify early in the development process to avoid future complications and technical debt. Developers often use code smells as a signal to refactor the code, making it cleaner and easier to maintain.

Detecting and addressing code smells is an ongoing process in software development. Tools like static code analysis and code review can help identify code smells. Refactoring is the practice of restructuring the code to remove code smells while preserving its external behavior. (Kralj 2023). And by paying attention to code smells and refactoring regularly, developers can maintain a healthy, maintainable codebase that is easier to understand, modify and extend over time.

Table 3 in the next page, presents common examples of code smells. And some of these code smells are subjective, so they are based on opinions and experience. Every code smell is not supported by concrete evidence and some may be inspired by aesthetic reasons. Typecasts are one of these, some developers try to minimalize the use of typecasts and some do not see any harm in using them and do not want to describe them as code smells. (Eva van Emden 2002).

Code smells are not always precise and this is related to the fact that code smells are subjective. For every project, one needs to define what the actual parameters are, e.g. which variable naming practices are used and what is the maximum size of classes and methods that are allowed.

Code smell	**Definition**
Long Method	A method that is excessively long, making it hard to read and understand. It can lead to duplication and make it challenging to maintain or modify
Large Class	A class that has grown too big, containing too many methods and responsibilities. This can lead to poor cohesion and high coupling
Duplicated Code	Repeated code snippets across the codebase, indicating a lack of abstraction or code reuse
God Class	A class that does too much, handling multiple responsibilities and violating the Single Responsibility Principle
Long Parameter List	A method with an excessive number of parameters, making the method call cumbersome and reducing readability
Feature Envy	When a method in one class excessively uses the data or methods of another class, indicating that the method might belong more naturally to the other class
Switch Statements	Overuse of switch or if-else statements, which can indicate a need for better polymorphism and object-oriented design
Data Clumps	When several data elements appear together in multiple places, suggesting they should be grouped into a single data structure
Primitive Obsession	Relying too heavily on primitive data types instead of using objects and classes, leading to code duplication and decreased maintainability
Shotgun Surgery	When a single change requires modifications to many different classes, indicating a lack of cohesion and proper encapsulation

Table 3: Common examples of code smells. (Kralj 2023)

3 Methodology

3.1 Research setup

This research was performed using the Customer Relationship Management software used by the case company. And the main focus was on the backend codebase of the case software. The codebase, from some parts is quite old as is the whole softwares lifecycle. There was a hunch that the software has technical debt, but the prevalence and the types of the technical debt were mainly only guesses. And they were only guesses, since there has not been done a research to examine the technical debt types and how common it is. There might be several reasons for accumulation of the technical debt but time pressure and the fast cycle of deploying the product over the years, could be the main reasons for it.

With these reasons, the product owner allowed that a master's thesis would be done with the product, so that the company and the developer team would get insights if the ongoing technical debt.

3.2 Research question

The research question of this Master's thesis is:

RQ1: What is the prevalence of technical debt in this software and what are the most common types of technical debt that developers face in this context?

Research question focuses only to the back-end code and every file which does not contain the *".java"* suffix, is left out from the analysis. By this it is possible to narrow down the scope, since the application at hand is very large.

3.3 Research methodology

This research uses case study as a research methodology. Case study is described to be deep and practical based research or experiment which examines complex phenomena in their natural settings to increase the understanding of the phenomenom. (Roberta Heale January

2018).

The steps in case study are the same as in other types of research. First is to define the single case or identifying a group of similar cases. Then a search conducted about what is known about the case, which usually involves a literature review to gain more information about the case(s) and informs the development of research question. The nature of the data is usully qualitative, but not in every case. Case studies aim to provide rich, detailed and contextually relevant insights into a particular phenomenon, organization, event or individual. It is usually used in social studies but also suitable for other fields as well. (Roberta Heale January 2018).

The analysis is important factor in case study. Case studies tend to be more selective and focus usually on one or two issues that are the most important parts to understand from the system, which is being examined. (Tellis 1997-06).

3.4 Data collection

In this case study, data collection was made with the help of code smells. Two different tools are used for data collection, Intellij Idea's code inspector tool and PMD plugin for Java code inspection. With these tools it was possible to gather crucial information about the technical debt from the software and visualize it. Today's IDE's (integrated development environment) provide various different tools for code smells. In this thesis, the used tools were free of charge and downloaded from the Intellij's IDEA's marketplace and in Section 3.5, more information about these tools can be found. The data was gathered on 23.09.2023 from the projects master branch, without any harmful bugs and software was running as it should.

3.5 Research instruments

In this section we go through the tools to be used in the code base analysis and why they were chosen. The idea was to use one tool directly from Intellij IDEA and the other one should be a plugin based inspector tool.

3.5.1 Intellij IDEA's code inspector

Intellij IDEA holds set of built-in code inspection tools. With these tools, user can detect and correct abnormal code in the project before compiling. It can eg. detect dead code, find bugs and improve the overall code structure. (documentation July 18, 2023a).

Some times the inspection requires global code analysis and by default, they are disabled in the editor. These inspection tools are for eg. code smells. To get a full and detailed report, these inspections are ran manually. (documentation August 21, 2023b).

Intellij IDEA's own code inspection is the first tool to find code smells and to get data about technical debt in the project. To get more reliable data about technical debt, code base is ran by other inspection tool also. The tool provided by Intellij IDEA needed some configuration to search only *.java* files from the project and it was done directly from the inspection tool.

3.5.2 PMD

The second tool was decided from these four different tools inFusion, JDeodorant, PMD and JSpIRIT. These tools were used by Paiva et al. (2017-10-06) in their research and the tools analyze Java code. The building of the exclusion criteria began by investigating if these tools were available in Intellij IDEA, since in the referenced research, the tools were applied in Eclipse IDE.

Only JDeodorant and PMD were available in the Intellij IDEA's marketplace. The reserch made an analysis of the accuracy of the used tools and they held high of the precision value over recall. From these two metrics, PMD had higher precision in the systems they tested with these tools. And therefore, the second code smell inspection tool is PMD. PMD inspects only *.java* by default, so further configuration was not needed.

3.6 Performing the research

This research was done with the analysis of the whole backend codebase and using case study as a research framework. The software is so large, that the research was forced to focus only to backend code. Also since the backend holds the logic to the software, it was

reasonable to narrow the scope and inspect the technical debt where it usually accumulates.

The case study was to identify and analyze potential code smells from the software. From the found code smells, analysis was gathered to represent how common technical debt is and what kind of technical debt is most the common in this context. From the analysis, a set of guidelines and future actions was represented how to minimize the technical debt.

The data analysis made with the help of the Intellij IDEA's own code inspection tool and a plug-in PMD code inspection tool, required some textfile parsing and inclusion criterias to set the scope for the research more reasonable.

The HTML file which was created by PMD code inspection tool was almost two million lines long, since it held the information eg. about the code smell, the java class from which it was found and the spesific description of the code smell in the html table. And because it was not great to read for human eye, it needed parsing. The HTML file held the information as shown in Listing 3.1 below:

Listing 3.1: PMD result example

```
<tr>
<td align="center">317920</td>
<td width="*%">C:\hidden\hidden\hidden\hidden\theJavaFile.java</td>
<td align="center" width="5%">632</td>
<td width="*">
<a href="https://pmd.github.io/pmd-6.55.0/pmd_rules_java_design.
html#lawofdemeter">
Potential violation of Law of Demeter (object not created locally)
</a>
</td>
</tr>
```

And the parsing was done with a small Python program shown below in Listing 3.2:

Listing 3.2: Python program for parsing html file and writing results to a output file

```
import re
from collections import Counter

# Read the HTML content from the file
input_filename = 'report.html'
with open(input_filename, 'r') as input_file:
    html_content = input_file.read()

# Define a regexp pattern to match the text between '#' and '">'
pattern = r'#(.*?)">'

# Use the regular expression to find all matches in the HTML content
matches = re.findall(pattern, html_content)

# Count occurrences of each warning message
warning_counts = Counter(matches)

# Calculate the total count of all warning messages
total_count = sum(warning_counts.values())

# Sort the warning messages by count in descending order
sorted_warnings = sorted(warning_counts.items(),
key=lambda x: x[1], reverse=True)

# Write the total count at the top of the list
sorted_warnings.insert(0, (f"Total␣Count:␣{total_count}", total_count))

# Write the sorted warning messages and their counts to a new file
output_filename = 'output_file.txt'
with open(output_filename, 'w') as output_file:
    for warning_message, count in sorted_warnings:
      output_file.write(f"{warning_message.strip()},␣Count:␣{count}\n")
```

The program in Listing 3.2 created a text file containing the total count of code smells, the

code smell and the count of how many times it occurred in the file as seen in Listing 3.3 below:

Listing 3.3: Example of the output file

```
Total Count: 317920, Count: 317920
cyclomaticcomplexity, Count: 1900
```

After the file was readable and code smells sorted by count, it was necessary to go through the list and select the code smells to the final results by an inclusion criteria. Every code smell was found from the PMD documentation (PMD October 6, 2023e) and with that, it was possible to check if the code smell matched to the inclusion criteria. The inclusion criteria is shown below. By this inclusion criteria, the final sum of different code smells was 45.

Inclusion criteria
Clean code principles
Design
Performance
Best practices
priority higher than medium, except in performance

Table 4: PMD inclusion criteria

The Intellij IDEA's own code inspection tool also generated a html file from the analysis results and it was hard to read and required parsing and inclusion criteria with the same idea as the PMD's results.

The analysis results from Intellij IDEA were quite different in their form and it needed some skimming and analysing the text to find patterns. After finding the patterns from the text file, which was copied from the html page, it was clear that by including the lines with the words 'inspection' and 'group', it was possible to gather the correct data from it.

The text file from the html page had the structure presented in Listing 3.4:

Listing 3.4: Example Intellij IDEA's code inspection

```
IntelliJ IDEA inspection report:
Inspection tree:
'Inspections Results' project 5575 warnings 2089 weak warnings 2421 typos

    General group   102 warnings 1 030 weak warnings
        Duplicated code fragment inspection   1 030 weak warnings
            class JavaClassService   1 weak warning
                WEAK WARNING Duplicate code: lines 119-130
```

And for parsing right contents from it, a Python program was created as seen from Listing 3.5:

Listing 3.5: Python program for parsing Intellij IDEA's results

```
input_filename = 'intellijidea.txt'
output_filename = 'intellijidearesult.txt'

# Open the input file in read mode
with open(input_filename, 'r') as input_file:
    # Read all lines from the input file
    lines = input_file.readlines()

# Filter lines containing the words "group" and "inspection"
# but not "'group'"
or "'inspection'"
filtered_lines = [line for line in lines if (' group  ' in line.strip()
and "'group'" not in line) or ('inspection' in line.strip() and
"'inspection'" not in line)]

# Open the output file in write mode and write the filtered lines
with open(output_filename, 'w') as output_file:
    output_file.writelines(filtered_lines)
```

The parsing had to be done differently with keyword 'group', the file contained different

text parts containing the word. But the right lines were captured adding whitespace to the keyword as seen in the Python program.

And with the help of the Python program, the final text file was in the form as seen from Listing 3.6:

Listing 3.6: Intellij IDEA's final output file

```
IntelliJ IDEA inspection report:
  General group   102 warnings 1 030 weak warnings
      Duplicated code fragment inspection   1 030 weak warnings
      Redundant suppression inspection   102 warnings
  HTML group   4 warnings
      Obsolete attribute inspection   4 warnings
```

After the file was readable, an exclusion criteria was added to the process to narrow down irrelevant code smells and finalizing the the actual code analysis for the results. The exclusion criteria for Intellij IDEA's code inspection was as follows:

Exclusion criteria
HTML group
JVM language group
Proofreading group
Kotlin group
Weak warnings
Code style issues group
Code maturity group
Documentation related

Table 5: Intellij IDEA's exclusion criteria

For both code inspection tools, the main priority was performance, design and only Java code.

4 Case study

4.1 Case company

The company and the software in this case study are anonymous due to the request of the company, so the background for the case company and the case software are quite general and short and aliases are created for them to ease the writing process. The aliases are *Good Test Oy*, for the company and *Silverback*, for the software. Software's type is CRM (Customer Relationship Management) system and it has been built solely for this company and it has been in use for over a decade.

The Good Test Oy is a Finnish company founded many years ago with a strong focus on the pulp and paper, energy and other process industries. Over the years it has evolved and diversified its operations in many ways to become a strong company in its areas of expertise.

The Good Test Oy has thousands of employees and a lot of subcontractors which deliver solutions to the main company.

4.2 Case software

The Silverback's lifecycle started over 15 years ago and has been in use ever since. Silverback's software type is customer relationship management and it is one of the most used softwares in the company. The software is a web-application using Java Spring framework which is the most popular application development framework for enterprise Java.

The Silverback holds important information about the company's customers, suppliers and other stakeholder groups. It has integrations to many other platforms and services and the software can have over 1000 simultaneous users. It also has a mobile application, which is not updated so often and is excluded from the research.

The Silverback is developed by a subcontractor with a team of three members at the moment. The developer team has seen changes during the years and software has seen over 30 different developers during its lifecycle but the core has been the same for many years, which has

been a strong point for the development of the software. In the next section, case software's characteristics and history are gone through in general.

4.3 Customer relationship management

A CRM (Customer Relationship Management) software is a type of business application designed to help organizations manage and analyze interactions and relationships with their customers. It provides a centralized system for storing customer data, tracking customer interactions and managing various aspects of the customer lifecycle. It is customer-focused system which aim is to create and add value to for the company and its customers (Chalmeta July 2006). For example, a company creates database about their customers, which shows sufficient and detailed relationships so that eg. management and salespeople could access the information and provide services and needs for their customers. It also helps the company utilize its customers profitability to their full potential, in present and in the future (Yurong Xu 2002).

Normally, CRM has four characteristics, salesforce automation, customer service and support, field service and marketing automation. Characteristics have an impacts as shown in Table 6 below. These characteristics are the baseline for CRM applications.

Characteristics	**Impact**
Salesforce automation	Empowered sales professionals
Customer service and support	Customer problems can be solved efficiently through proactive customer support
Field service	Remote staff can efficiently get help from customer service personnel to meet customers' individual expectations
Marketing automation	Companies can learn clients' likes and dislikes to better understand customers' needs

Table 6: CRM characteristics and impacts. (Yurong Xu 2002)

First wave of CRM solutions are dated back to 1980s and early 1990s and the market grew

rapidly in the 2000s. First web-based CRMs were introduced by SAP in 1999 (Yurong Xu 2002) and with internet's involvement functions of a CRM changed a lot. With the help of the internet customer can actually transact with the companies and by that the companies can implement more functionality to their customers and create value for them.

5 Results

5.1 Most commmon types of technical debt

When examining the results from both inspection tools, most of the code smells were related to the *Performance* and *Maintainability* issues. The inclusion and exclusion criterias for Intellij IDEA's and PMD inspection tools also quided towards performance and maintainability.

5.2 Intellij IDEA's code smells

The total code smell count from Intellij IDEA's code inspection tool was 10085 (after inclusion/exclusion criteria) and from that with the help of the exclusion criteria, the Java group had the most code smells with the count of 3347. Intellij IDEA's inspection report included different sub groups which then holds more information about the group. It also states the count of warnings and weak warnings in each group. In table below, the count of warnings have a suffix '**W**' and weak warnings have a suffix '**w**'.

In Figure 1 below, total code smells by group are visualized by percentages. The counts are from the data after implementing inclusion and exclusion criterias. The biggest percentage was from Java group, which has mainly performance and design related smells.

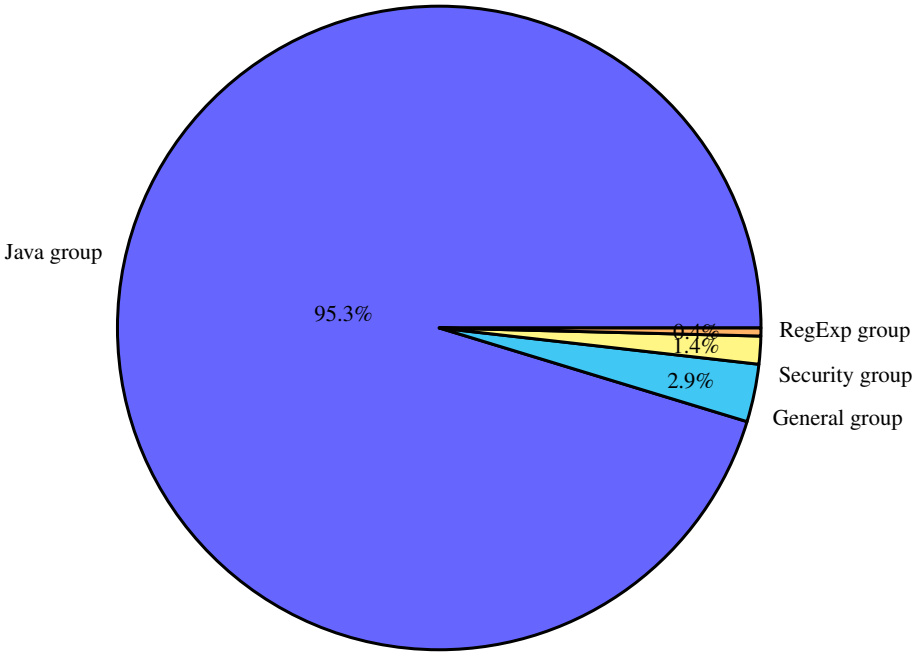

Figure 1: IntelliJ IDEA's Results Chart

In Table 7, a more detailed view of the results is presented. Empty cells indicates that there are not anymore code smells in that particular area of code smells.

Table 7: IntelliJ IDEA inspection report overview

Java group	**General group**	**Security group**	**RegExp group**
Java language level migration aids group, Count: 1875**W**	Redundant suppression inspection, Count: 102**W**	Vulnerable imported dependency inspection, Count: 50**W**	Duplicate character in character class inspection, Count: 12**W**
Javadoc group, Count: 873**W**			Unnecessary non-capturing group inspection, Count: 1**W**
Class structure group, Count: 331**W**			
Verbose or redundant code constructs group, Count: 70**W**, 28**w**			
Memory group, Count: 74**W**			
Performance group, Count: 42**W**			
Control flow issues group, Count: 3**W**,3**w**			
Numeric issues group, Count: 16**W**			
Resource management group, Count: 15**W**			
Probable bugs group, Count: 13**W**			
Threading issues group, Count: 3**W**			
Cloning issues group, Count: 1**W**			

In the next sections, the most important smells are gone through in more detail from every code smell group. Every code smell definition in the following sections, has been searched from the JetBrains code inspection documentation. The documentation was an aid to describe the code smells in a more detailed way, if it was possible or if the smell was found from the documentation.

5.2.1 Java Group

The largest primary group for code smells was Java group. The inspection tool found a total of 5406 warnings and 840 weak warnings from the Java group. There were total of 12 sub-groups and since the number is quite low, every sub-group is also described in more detail. Except the Javadoc group, since it is document related. Also as stated before in the exclusion criteria, weak warnings are not included to the final results, so only notable warnings are gone through in more detail.

Java language level migration aids group: had a total of 1875 warnings and all of the warnings were in the subsub-group called **Java 5 group** and the warnings are described below.

1. **'BigDecimal' legacy method called inspection:** this reports BigDecimal legacy method usages in the project and this had 96 warnings.
2. **Raw use of parameterized class inspection:** this inspection identifies situations where a programmer is using a generic class without specifying its type parameters. For example using List<T>, without specifying the type parameter (eg. `List<String>`). This may result in runtime errors or make the code less readable and maintainable.

Class structure group: had a total of 331 warnings and all of them were in the **Non-final field in 'enum' inspection**. This means that when programmer declares a field in an enum without marking it as final, code inspection tool declares it as a violation.

Verbose or redundant code constructs group: had a total of 70 warnings 28 weak warnings and the notable warnings were spread on to four different inspections.

1. **Condition is covered by further condition inspection:** this inspection is designed to find situations where a conditional statement is unnecessary due to a prior condition. This inspection is aimed at improving code clarity and efficiency. Redundant conditions can make code harder to read and maintain. This had 61 warnings.
2. **Explicit array filling inspection:** this identifies instances where an array is explicitly filled with the same value for all elements. Instead of manually populating each element with the same value, it suggests using the `Arrays.fill` method. This had one

warning.

3. **Manual min/max calculation inspection:** This reports situations where a programmer manually calculates the minimum or maximum of two values instead of using the built-in `Math.min` or `Math.max` methods (JetBrains May 16, 2023a). This had four warning.
4. **Unnecessarily escaped character inspection:** this reports instances where a character is unnecessarily escaped in a string literal. Example string `[]` can be modified to `-;[.]`. This had four warning.

Memory group: had a total of 74 warnings and every warning was included in the **Inner class may be 'static' inspection**. This identifies the situations where an inner class could be declared as static. Making the inner class static can have benefits in memory usage and can also improve performance.

Performance group: is one of the most important groups in this inspection. Performance group had a total of 42 warnings with four different types.

1. **Call to 'list.containsAll(collection)' may have poor performance inspection:** this inspection is concerned with the time complexity and might have poor performance related issues, espescially if the list type is eg. `LinkedList`, as it involves iterating through the entire list to check for each element. This had two warnings.
2. **Call to 'set.removeAll(list)' may work slowly inspection:** this inspection is also concerned with the time complexity and the `set.removeAll(list)` may work slowly if the Set is implemented as a `TreeSet`. With HashSet the operation is generally fast. This had six warnings.
3. **Early loop exit in 'if' condition inspection:** this identifies if the code can be refactored to more simpler form when there is a early exit from a loop.
4. **'InputStream' and 'OutputStream' can be constructed using 'Files' methods inspection:** this identifies situations where instances of `InputStream` or `OutputStream` can be more efficiently created using methods from the Files utility class. Example

   ```
   InputStream inputStream = new FileInputStream(new File("path/to/f
   ```

   instead of this, one would adviced to do this as shown in Listing 5.1 below:

Listing 5.1: InputStream method example

```
InputStream inputStream = Files.newInputStream(Paths.get("path/to/file"));
```

this will provide a more concise syntax but also allows better handling of IOExceptions.

Control flow issues group: had a total of three warnings and 24 weak warnings. All the notable warnings were in the **Minimum 'switch' branches inspection**. This reports the situations where a switch statement has more branches than necessary. This helps to simplify the form of the switch statement when some cases have the same behavior.

Numeric issues group: this had a total of 16 warnings and they were spread into three different inspections.

1. **Cast group:** had one inspection of **Integer multiplication or shift implicitly cast to 'long' inspection**. This is built to identify situations where the result of an integer multiplication or shift operation is implicitly cast to a long. This inspection helps ensure that the result is correctly handled and potentially avoids overflow issues. This had six warnings.
2. **Unary plus inspection:** this is designed to identify situations where a unary plus operator (+) is used on a numeric expression unnecessarily. Example: int value = +5;. This had seven warnings.
3. **Unpredictable 'BigDecimal' constructor call inspection:** this focuses on identifying situations where the BigDecimal constructor is called with a floating-point value. This had three warnings.

Resource management group: this had in total of 15 warnings and all of the warnings came from the **AutoCloseable used without 'try'-with-resources inspection**. This is designed to identify situations where an AutoCloseable resource is being used but not enclosed with a try-with-resources statement.

Example of an autocloseable statement in Listing 5.2 below.

Listing 5.2: AutoCloseable statement

```
//Using using try-with-resources
AutoCloseableResource resource = new AutoCloseableResource();
resource.doSomething();
resource.close();
// VS.
// Using try-with-resources
try (AutoCloseableResource resource = new AutoCloseableResource()) {
  resource.doSomething();
}
```

Probable bugs group: this had a total of 13 warnings and they were spread into three different inspections.

1. **Redundant operation on empty container inspection:** this is designed to identify situations where operations are performed on a container that is known to be empty. E.g. declaring `List<String>` list variable and immediately checking if it is empty `(if (list.isEmpty()) )`. This had one warning.
2. **Sorted collection with non-comparable elements inspection:** this identifies situations where one is using a sorted collection, such as `TreeSet`, with elements that are not comparable. This had nine warnings.
3. **Suspicious date format pattern inspection:** this identifies problems with date formatting, where a date format pattern string might lead to unexpected results or errors. This had three warnings.

Threading issues group: this had in total of three warnings and every warning was in the same inspection scope **Busy wait inspection**. A busy wait occurs when a loop repeatedly checks a condition without yielding or sleeping, consuming CPU resources unnecessarily. For example shown in Listing 5.3:

Listing 5.3: A Busy Wait

```
while (!condition) {
 // Perform some actions
}
```

If condition is not changing inside the loop, this can result in high CPU usage as the loop keeps checking the condition repeatedly without allowing other tasks to run.

Cloning issues group: this had in total of one warning and it was in the inspection **Cloneable class without 'clone()' method inspection**. This identifies the situations where a class implements the Cloneable interface but does not provide a public `clone()` method.

5.2.2 General Group

The code inspection found total of 102 warnings and 1030 weak warnings from the general group. Warnings were limited to only one inspection group called Redundant suppression inspection. In this group, also weak warnings were taken into consideration, since the weak warnings contained duplicate code inspection and it is from maintainability persperctive, quite important tool to identify technical debt. Every weak warning was in the duplicated code fragment inspection group.

Redundant suppression inspection: this reports usages of the elements, that can be removed because the inspection they affect is no longer applicable in this context (JetBrains May 13, 2023b). For example the use of `@SuppressWarning` annotation.

Duplicated code fragment inspection: this reports every duplicated code block from the inspected scope (JetBrains May 13, 2023b).

5.2.3 Security Group

There were total of 50 warnings and 143 weak warnings in the security group. Every notable warning, was linked to the Vulnerable imported dependency inspection, which means larger and more complex projects tend to need various third-party dependencies that help

develop productivity, extending the common libraries and frameworks functionality. And this code smell states that there might be some kind of vulnerability issues in 50 different dependencies, that the project uses. (JetBrains September 7, 2023c).

5.2.4 RegExp Group

In the RegExp group code inspection tool found total of 13 warnings and 67 weak warnings. Notable warnings included the Duplicate character in character class inspection with the count of 12 and Unnecessary non-capturing group inspection with one occurence.

Duplicate character in character class: this reports duplicate characters inside regular expression class. These duplicate characters are not necessary and can be removed (JetBrains May 17, 2022a).

Example: [aabc] **->** [abc].

Unnecessary non-capturing group: this reports any unnecessary non-capturing groups, which do not have influence on the result (JetBrains May 13, 2022b).

5.3 PMD code inspection tool code smells

The total code smell count from PMD code inspection tool was 18408 (after inclusion/exclusion criteria) and from that with the help of inclusion criteria, Consecutive Appends Should Reuse code smell from the Performance primary smell group held the most code smells with the count of 3410.

In Figure 2 below, total code smells by group are visualized by percentages. The counts are from the data after implementing inclusion and exclusion criterias. Biggest percentages here are from Performance, but the PMD inspection tool was more precise in terms of dividing the smells into smaller parts and the other areas are closer to it, unlike in Figure 1, where Intellij IDEA's inspection results are visualized.

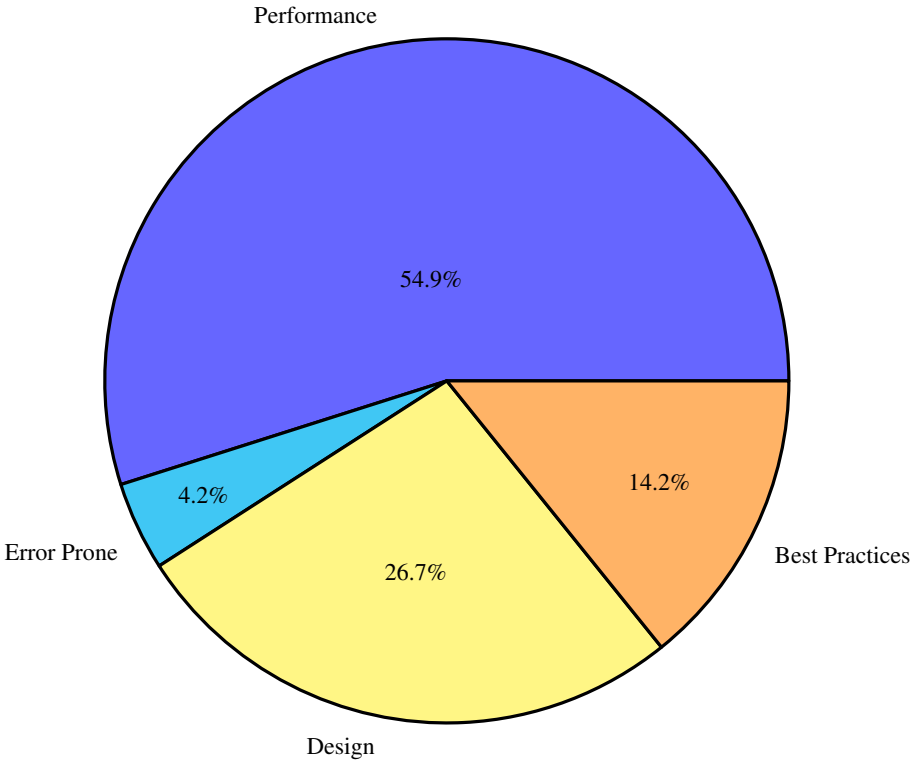

Figure 2: PMD Results Chart

In Table 8 a more detailed view of the results is presented. Empty cells indicates that there are not anymore code smells in that particular area of code smells.

Table 8: PMD inspection report overview

Performance	**Error Prone**	**Design**	**Best Practices**
Consecutive Appends Should Reuse, 3410	Return Empty Collection Rather Than Null, 423	Cyclomatic Complexity, 1900	Guard Log Statement, 2016
Avoid Instantiating Objects In Loops, 1658	Constructor Calls Overridable Method, 279	Modified Cyclomatic Complexity, 1535	Avoid Reassigning Parameters, 544
Consecutive Literal Appends, 1198	Return Empty Array Rather Than Null, 30	Avoid Catching Generic Exception, 1343	System Println, 56
Redundant Field Initializer, 754	More Than One Logger, 13	Avoid Throwing Raw Exception Types, 134	
Append Character With Char, 734	Avoid Branching Statement As Last In Loop, 5	Class With Only Private Constructors Should Be Final, 5	
Inefficient StringBuffering, 733	Proper Clone Implementation, 5	Abstract Class Without Any Method, 2	
Use StringBuffer For String Appends, 468	Logger Is Not Static Final, 4		
Insufficient StringBuffer Declaration, 404	Suspicious Equals Method Name, 4		
Add Empty String, 266	Broken Null Check, 2		
Avoid Calendar Date Creation, 139	Wquals Null, 2		
Use Index Of Char, 70			
Integer Instantiation, 60			
Avoid FileStream, 41			
Long Instantiation, 40			
Simplify StartsWith, 29			
String To String, 24			
String Instantiation, 15			
Avoid Array Loops, 13			
Optimizable To Array Call, 10			
Boolean Instantiation, 10			
Avoid Using Short Type, 9			
Too Few Branches For A Switch Statement, 8			
Unnecessary Wrapper Object Creation, 7			
Useless String ValueOf, 3			
Use Arrays As List, 2			
Inefficient Empty String Check, 1			

In the next sections, the most important smells are gone through more detailed from every primary code smell group. In the performance group, only smells with the count of 500 and more or if priority equals/is higher than medium-high, are gone through. If a rule is deprecated it is not gone through in a more detailed way.

5.3.1 Performance

Consecutive Appends Should Reuse code smell means that consecutive append calls with `StringBuffer`/`StringBuilder` should be chained. This can improve the performance by producing a smaller bytecode, reducing overhead and improving inlining. The priority for this is medium.

Avoid Instantiating Objects In Loops this means that one should avoid creating new objects in a loop and should be checked if it is possible to create them outside of a loop. Priority for this smell is medium.

Consecutive Literal Appends means that one should not call

`StringBuffer`/`StringBuilder.append(...)`

with literals, since the literals are constants, they can already be combined into a single `String` literal and this String can be appended in a single method call (PMD October 29, 2023d). Priority for this smell is medium.

Redundant Field Initializer this means that Java will initialize fields with known default values so any explicit initialization of those are redundant and results in a larger class file (approximately three additional bytecode instructions per field) (PMD October 29, 2023d). Priority for this smell is medium.

Append Character With Char means that a programmer should avoit concatenating characters with `StringBuffer`/`StringBuilder.append` methods (PMD October 29, 2023d). Priority for this smell is medium.

Inefficient StringBuffering this means that a programmer should avoid concatenating non-literals in a `StringBuffer` constructor or `append()` method. Main reason is that JVM

will create and destroy intermediate buffers (PMD October 29, 2023d). Priority for this smell is medium.

Avoid FileStream this means that the classes `FileInputStream` and `FileOutputStream`, contains a finalizer method which will cause pauses in garbage collection. The `FileReader` and `FileWriter` constructors instantiate `FileInputStream` and `FileOutputStream` and causing garbage collection issues while `FileInputStream` and `FileOutputStream` classes finalizer methods are called (PMD October 29, 2023d). Priority for this smell is high.

Long Instantiation this means that when calling `new Long()` constructor, it causes memory allocation that can be avoided by the static `Long.valueOf()`. It makes use of an internal cache that recycles earlier instances making it more efficient memory-wise. Also, `new Long()` has been deprecated since JDK 9 for that reason (PMD October 29, 2023d). Priority for this smell is medium-high.

String Instantiation this means that a programmer should avoid instantiating String objects (`String foo = new String("foo");`), since String objects are immutable (PMD October 29, 2023d). Priority for this smell is medium-high.

Boolean Instantiation this means that a programmer should avoid instantiating Boolean objects (`Boolean bar = new Boolean("true");`). `new Boolean` is deprecated since JDK 9.0 and preferable way is eg. `Boolean.TRUE` (PMD October 29, 2023d). Priority for this smell is medium-high.

Avoid Using Short Type this means that Java uses the *short* type to reduce memory usage, not to optimize calculation. But, JVM does not have arithmetic capabilities for the short type. In fact, the JVM must convert it into int, then do the calculations and finally convert the short back to int. So any storage gains which the short type gained, may be offset by adverse impacts on performance (PMD October 29, 2023d). Priority for this smell is high.

5.3.2 Error Prone

Return Empty Collection Rather Than Null this means that if a method returns a collection, such as an array, a collection or a map, it should return an empty one rather than null.

This automatically removes the need for null checking and avoid `NullPointerExceptions` (PMD November 5, 2023c). Priority for this smell is high.

Constructor Calls Overridable Method this means that a constructor in a Java class calls a method that can be overridden by subclasses. This can lead to unexpected behavior, eg. `NullPointerException`, during object construction, especially when the subclass overrides the invoked method. To avoid this problem, a programmer should only use methods that are static, private or final in the constructors (PMD November 5, 2023c). Priority for this smell is high.

More Than One Logger this means that normally a class is using only one logger. This rule supports slf4j, log4j, Java Util Logging and log4j2 (PMD November 5, 2023c). Priority for this smell is medium-high.

Avoid Branching Statement As Last In Loop this means that using a branching statement as the last part in a loop may be a bug and/or it might be confusing (PMD November 5, 2023c). Priority for this smell is medium-high.

Proper Clone Implementation this reports the situations where a objects `clone()` method, should be implemented with `super.clone()` (PMD November 5, 2023c). Priority for this smell is medium-high.

Suspicious Equals Method Name this means that the methods name and number of parameters are suspiciously close to `Object.equals()` method, which can denote an intention to override it. Instead of overriding the method `Object.equals()`, it overloads it instead. Overloading `Object.equals` method can be confusing for other programmers, error-prone and hard to maintain (PMD November 5, 2023c). Priority for this smell is medium-high.

Broken Null Check this means that the null check is broken since it will throw a

`NullPointerException`

itself. It is likely that a programmer has used `||` instead of `&&` or vice versa (PMD November 5, 2023c). Priority for this smell is medium-high.

Equals Null this report the situations where a tests for null should not use the `equals()` method. The == operator should be used instead (PMD November 5, 2023c). Priority for this smell is high.

5.3.3 Design

Cyclomatic Complexity means that the complexity of methods directly affects maintenance costs and readability. Cyclomatic complexity assesses the complexity of a method by counting the number of decision points in a method and adding one for the method entry. These decision points are places where the control flow jumps to another place in the program and they include all control flow statements, such as if, while, for and case. Generally, complexity numbers range from 1-4 (low complexity), 5-7 (moderate complexity), 8-10 (high complexity) and 11+ (very high complexity) (PMD November 5, 2023b). Priority for this smell is medium.

Modified Cyclomatic Complexity this is the same as **Cyclomatic Complexity** but with speficied limit. By default, the number is 10.

Avoid Catching Generic Exception this reports situations where generic exceptions such as `NullPointerException`, `RuntimeException` or `Exception` are used in a `try-catch` block. Exceptions should be defined as precisely as possible (PMD November 5, 2023b). Priority for this smell is medium.

Avoid Throwing Raw Exception Types this means that is hould be avoided to throw certain exception types. Rather than throw a raw `RuntimeException`, `Throwable`, `Exception` or `Error`, one should use a subclassed exception or error instead (PMD November 5, 2023b). Priority for this smell is high.

Class With Only Private Constructors Should Be Final this reports the situations where classes should be made final because they cannot be extended from outside their compilation unit. This is because all their constructors are private, so a subclass could not call the super constructor (PMD November 5, 2023b). Priority for this smell is high.

Abstract Class Without Any Method this means that if an abstract class does not provide

any methods, it may be acting as a simple data container that is not meant to be instantiated. In this case, it would better to use a private or protected constructor in order to prevent instantiation than make the class abstract (PMD November 5, 2023b). Priority for this smell is high.

5.3.4 Best Practices

Guard Log Statement means that when using a log level, it should be checked if the loglevel is actually enabled or otherwise the associate String creation and manipulation should be skipped (PMD November 5, 2023a). Priority for this smell is medium-high.

Avoid Reassigning Parameters this means that reassigning values to incoming parameters is not recommended. Temporary values should be used instead since then it will not break the principle of least astonishment and making the code more understandable. It should be noted that this rule considers both methods and constructors (PMD November 5, 2023a). Priority for this smell is medium-high.

System Println this refers to situations where `System.(out|err).print` are used. Normally they are used for debugging purposes. By using a logger this behaviour can enabled/disabled at will (and by priority) and avoid clogging the Standard out log (PMD November 5, 2023a). Priority for this smell is medium-high.

5.4 Prevalence of the technical debt

Measuring prevalence of technical debt in a codebase is a challenging task but static code analysis tools help to identify it. In this research, it has to be noted that there has not been done anything like this before. And there was a lot of code smells generated by the static code analysis tools, so solely on this information, it can be stated that the prevalence of the technical debt would quite high. Both static code analyzers tools reported thousands of warnings from the project and while many of them has medium or low severity, many of them also had higher severity. The main focus was in the performance side in both inspection tools and both of them reported lots of different code smells and warnings.

Even though the count of the smells and warnings were high, the age of the software has to be taken into account. The roots of this CRM's lifecycle dates back to 2006 and without constant refactoring and maintaing of the code base, the count of code smells and warnings could be a lot higher. From that point of view, the prevalence of the technical debt would medium size at worst. And when taking the whole count of lines in the project, the prevalence would drop to low, since the count for total lines of java code is almost one million and the project total count of code lines is almost 3.5 million.

Since there is not anything to objectively define the technical debt, it comes down to the point of view to define the prevalence. And since the software should be bug-free, performing at its best, highly maintainable and clear to inspect and read for other programmers, the bottom line for prevalence of the technical debt would be medium in the scale of low, medium, medium-high and high.

6 Discussion

6.1 Common types and prevalence of technical debt

Even though the case software did not show any clear signs of technical debt, it was probable that it existed. And as shown in Section 5, the existence of it was clearly shown. The main area where technical debt was found was performance related. This was mainly due to the inclusion and exclusion criterias. Many of the reported code smells in total, were related to the style of programming, meaning that there can be multiple ways of executing a code block and one approach is usually faster and better than the other. Also there were code smells with deprecated functionalities, which may still work but should be refactored to the updated state.

Also, a lot of code smells were referring to coding practices which can easily lead to errors, harder maintainability and harder understanding. These code smells were eg. duplicate code blocks, style violations in a sense that in some cases programmer should avoid using one approach even though it works or something has extra parts when it would work also with less code and issues with class, method and statement structures. These prementioned areas can accumulate technical debt especially in the area of maintainability.

When performing conclusions of the prevalence of the technical debt question, it is harder to create a simple answer. Like stated in Section 5.4, there is not a clear way to measure it. The scale for the prevalence was created only to state the prevalence in some readable form. The measuring for it was done only by the researcher with the help of the data gathered by the tools and the common information from the software. The medium level prevalence could be higher or lower if someone else would have done the examination from the same data. Depending on the fact how familiar one is with the current state of the software and the gathered data. With that said, the prevalence of the technical debt in the software can't be viewed as an absolute truth, but a more of a point of view from a one developer.

6.2 Future and recommendations

6.2.1 Examples of preventing technical debt

The foundings of this master thesis, are heavily related to performance and maintainability issues. The researchers opinion is that the most important ways to prevent technical debt in this scope, are code refactoring and documentation. Code refactoring concerning found code smells would be quite easy, since the the master thesis will give the exact parts which to refactor. Documentation is always helpful for future developers in the project, but also to the current ones. It will clear the ongoing development and well documented project, will be helpful in the future if some kind of bugs etc. are found. It will clearly tell how some feature should be working, and helps the eg. debugging the issue. Below, there are some most of the important ways of preventing technical debt. Including the most important ones regarding this thesis, code refactoring and documentation.

Priorize and triage. Identify and prioritize the specific areas of technical debt based on their impact on the system and business goals. Triage issues to address critical debt first. Eg. prioritize for performance issues. (Thakkar 2022-07-19).

Refactoring. Plan and execute systematic refactoring efforts to identified code smells and issues. Break down large refactorings into smaller, manageable tasks. (Khan 2023-07-21).

Test coverage. Improve test coverage to ensure that changes to the codebase are protected well eoungh by automated tests. Address areas with low or no test coverage, especially those associated with high technical debt. (Marcin Dryka 2024-01-22)

Documentation. Enhance and update documentation to improve code readability and understanding. Clear documentation can help developers address technical debt more effectively. (Marcin Dryka 2024-01-22)

Monitoring and measuring. Continuously monitor and measure technical debt metrics to track improvements over time. (Marcin Dryka 2024-01-22)

Strategic planning. Incorporate technical debt reduction into the strategic planning for future releases. Allocate dedicated time in development cycles to focus on addressing technical

debt. (Marcin Dryka 2024-01-22)

6.2.2 Why refactor and renew codebase?

For longing the lifecycle of a software, it should go through analysis and renewal of codebase from time to time. Depending on the time issues and importance of the parts which are refactored, this is a process that product owner should take into account. In the scope of this Master's thesis, performance and maintainability are the parts that should be looked into more detail as seen from the code smells.

Performance aspect of a software is important, because for users, it is more pleasurable way of using the software when it is performing at its best. Usually when the software is performing well, it is also reliable for users, when taking eg. memory usage into account.

Maintainability is important aspect for developers, since when a software is getting new features and is getting refactored, it is important that the code base for the project is maintainable. When describing mainatability, it usually means minor code changes, while maintaining good readability, so that possible new developers can learn the software as quick as possible. Also it is highly important for current developers also, since many times their work cross each others work and it also eases the workload when finding possible bugs while debugging for example.

6.2.3 Future considerations

Since it was stated in the results, that the prevalence of the technical debt is medium in the project when it comes down to total count of code lines, the count of code smells and the age of the software there are some ideas and actions that should be done. And the preferred actions should at least include factors from time pressure and resourcing the budget.

The time pressure with new features which is one of the greatest factors that creates technical debt, it would be a good plan to allocate time and resources also to maintain the health of the code base. A healthy codebase would build certainty for the future. And the process should go on continuously, not just every once in a while, so that the prevention of the technical

debt would be as high as possible.

This master thesis can help to set a routine of exploring technical debt in the project scope. If same or similar kind of procedure is carried out eg. bi-monthly, statistics created out of the results, it can be helpful of seeing how the software is doing in terms of code smells and probable technical debt. This could prevent bugs, help maintainability and detect minor performance issues before they become major.

Also, the results presented in this master thesis can set the base for the future procedures. The technical part of the results should be fixed, so that performance and maintainability would be in better shape. The results are handed out in a rather technical way, so it would be quite easy for a certain developer to fix the issues presented in the results.

6.3 Limitations

This study has its limitations when researching the technical debt from the project. The scope focused only to back-end code and while the logic of the software is there, it is highly possible and even probable that there lies more technical debt if the whole project would be examined. As said about the total lines of code, back-end Java code is roughly one third from the actual total lines of code.

When creating the inclusion and exclusion criteria, a lot of coding best practices were excluded from the project, due to the project's unique nature and the fact that different developers write code in different ways and there is not a single way of doing it. Also the inclusion and exclusion criterias exclude lot of weaker warnings, which alone may not be as important or severe, but when weak warnings accumulate, they might create a situation where they are in fact more severe than initially thought.

When taking these limitations into account, this thesis brings some overall view for the questions what the prevalence of the technical debt is and what are the common types of it. More detailed perspective would require analysing the whole codebase and section by section score the severity and fix the issues.

7 Conclusions

The aim of this study was to research technical debt, the prevalence of it and the common types in a CRM software. The precise research question was "What is the prevalence of technical debt in this software and what are the most common types of technical debt that developers face in this context?".

The results gathered from the data, showed that the prevalence of the technical debt in this context is at medium level. The assessment of the prevalence was done solely by the researcher, with knowledge about the software itself, software's current state and the gathered data.

The total amounts of code smells related to technical were initially so large that some exclusion and inclusion criterias had to be applied. With the built criterias, the total amounts were 10085 from Intellij IDEA's tool and 18408 from the PMD tool. Most common types were related to performance issues in both tools, so it is safe to say that the software's most common technical debt type is performance issues.

Bibliography

Abrahamsson, Pekka, et al. 2002. “Agile Software Development Methods: Review and Analysis”, visited on August 21, 2023. https://arxiv.org/ftp/arxiv/papers/1709/1709.08439.pdf.

Beck, Kent, et al. 2001. *Manifesto for Agile Software Development.* Visited on July 27, 2023. https://agilemanifesto.org/iso/en/manifesto.html.

Chalmeta, Ricardo. July 2006. “Methodology for customer relationship management” (). Visited on June 11, 2023. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=56f3fa514238cf67cca08f9809ee488adc0af840.

Cunningham, Ward. October 1992. “The WyCash Portfolio Management System” (): 29–30. Visited on May 28, 2023. https://dl.acm.org/doi/pdf/10.1145/157709.157715.

documentation, Intellij IDEA. July 18, 2023a. *Code inspections.* July 18, 2023. Visited on August 31, 2023. https://www.jetbrains.com/help/idea/code-inspection.html.

———. August 21, 2023b. *Run inspections.* August 21, 2023. Visited on August 31, 2023. https://www.jetbrains.com/help/idea/running-inspections.html.

Eva van Emden, Leon Moonen. 2002. “Java Quality Assurance by Detecting Code Smells”, visited on September 3, 2023. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4974d6245dca38cb16370c964907b9aba5a614af.

Frederico Oliveira, Viviane Santos, Alfredo Goldman. 2015. “Managing Technical Debt in Software Projects Using Scrum: An Action Research”, visited on March 23, 2024. https://doi.org/10.1109/Agile.2015.7. https://www.hanssamios.com/dokuwiki/_media/managing_technical_debt_in_software_projects_using_scrum.pdf.

Hron, Michal. 2018. “Scrum in practice: an overview of Scrum adaptations”, visited on August 27, 2023. https://scholarspace.manoa.hawaii.edu/server/api/core/bitstreams/c7dad016-2792-4ec5-96a5-1941a7d43c11/content.

JetBrains. May 17, 2022a. *Code Inspection: Duplicate character in character class.* May 17, 2022. Visited on October 21, 2023. https://www.jetbrains.com/help/phpstorm/regexp-duplicate-character-in-character-class.html.

———. May 13, 2022b. *Code Inspection: Unnecessary non-capturing group.* May 13, 2022. Visited on October 21, 2023. https://www.jetbrains.com/help/datagrip/regexp-unnecessary-non-capturing-group.html.

———. May 16, 2023a. *Code Inspection: Condition can be replaced with 'min()'/'max()' call.* May 16, 2023. Visited on October 29, 2023. https://www.jetbrains.com/help/phpstorm/php-condition-can-be-replaced-with-min-max-call.html.

———. May 13, 2023b. *Code Inspection: Redundant suppression.* May 13, 2023. Visited on October 21, 2023. https://www.jetbrains.com/help/phpstorm/general-redundant-suppression.html.

———. September 7, 2023c. *Vulnerable dependencies.* September 7, 2023. Visited on October 21, 2023. https://www.jetbrains.com/help/idea/package-analysis.html.

Khan, Noor UL Ain. 2023-07-21. "The Art of Code Refactoring: Strategies for Clean and Maintainable Code", visited on February 28, 2024. https://medium.com/@noorulaink00/the-art-of-code-refactoring-strategies-for-clean-and-maintainable-code-9eac8afa58b6.

Kralj, Kristijan. 2023. "21 Deadly Code Smells You'll Wish You Discovered Years Ago", visited on September 3, 2023. https://methodpoet.com/code-smells/.

Kruchten, Philippe, et al. 2013. "Technical Debt: Towards a Crisper Definition". *ACM SIGSOFT Software Engineering Notes* 38 (5). https://apps.dtic.mil/sti/pdfs/AD1015409.pdf.

Marcin Dryka, Matt Warcholinski, Olga Gierszal. 2024-01-22. "How to Reduce Technical Debt – a Guide for CTOs", visited on February 28, 2024. https://brainhub.eu/library/how-to-deal-with-technical-debt.

Paiva, Thanis, et al. 2017-10-06. "On the evaluation of code smells and detection tools". *Journal of Software Engineering Research and Development* 5. Visited on September 6, 2023. https://doi.org/10.1186/s40411-017-0041-1. https://jserd.springeropen.com/articles/10.1186/s40411-017-0041-1.

Philippe Kruchten, Iped Ozkaya, Robert L. Nord. November 2012. “Technical Debt: From Metaphor to Theory and Practice” (): 18–21. Visited on June 3, 2023. https://resources.sei.cmu.edu/asset_files/WhitePaper/2012_019_001_58818.pdf.

PMD. November 5, 2023a. *Best Practices code smells.* November 5, 2023. Visited on November 5, 2023. https://pmd.github.io/pmd/pmd_rules_java_bestpractices.html.

———. November 5, 2023b. *Design code smells.* November 5, 2023. Visited on November 5, 2023. https://pmd.github.io/pmd/pmd_rules_java_design.html.

———. November 5, 2023c. *Error Prone code smells.* November 5, 2023. Visited on November 5, 2023. https://pmd.github.io/pmd/pmd_rules_java_errorprone.html.

———. October 29, 2023d. *Perforamnce code smells.* October 29, 2023. Visited on October 29, 2023. https://docs.pmd-code.org/latest/pmd_rules_java_performance.html.

———. October 6, 2023e. *PMD Documentation.* October 6, 2023. Visited on October 7, 2023. https://pmd.github.io/pmd/index.html.

Roberta Heale, Alison Twycross. January 2018. “What is a case study?” (). Visited on August 21, 2023. https://ebn.bmj.com/content/ebnurs/21/1/7.full.pdf.

Schwaber, Ken. 1997. “SCRUM Development Process”, visited on August 25, 2023. http://damiantgordon.com/Methodologies/Papers/Business%20Object%20Design%20and%20Implementation.pdf.

Team, SoftTeco. 2023-07-20. “What Is Code Smell And How To Reduce It?”, visited on January 29, 2024. https://softteco.com/blog/whats-code-smell.

Tellis, Winston. 1997-06. “ntroduction to Case Study”. *The Qualitative Report* 3. Visited on September 6, 2023. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=e6408ea90ae0050ade47a40eec7aa6204e553092.

Thakkar, Bindiya. 2022-07-19. “What is technical debt? How to prioritize and avoid with examples”, visited on February 28, 2024. https://blog.logrocket.com/product-management/what-is-technical-debt-examples-prioritize-avoid/.

Timbó, Rafael. 2023. “Technical Debt In Agile and Scrum”, visited on January 14, 2024. https://www.revelo.com/blog/technical-debt-in-agile.

Yurong Xu, Binshan Lin, David C. Yen. 2002. “Adopting customer relationship management technology”, visited on June 19, 2023. http://modir3-3.ir/article-english/article140.pdf.