university of groningen

faculty of science and engineering

Is Design Pattern Grime Related To Technical Debt?

Ana Terna
Karol Machnik

university of groningen

faculty of science and engineering

University of Groningen

Is Design Pattern Grime Related to Technical Debt?

Bachelor's Thesis

To fulfill the requirements for the degree of
Bachelor of Science in Computing Science
at the University of Groningen under the supervision of
Daniel Feitosa (Computer Science, University of Groningen)
and
Ayushi Rastogi (Computer Science, University of Groningen)

Ana Terna (s4396618)
Karol Machnik (s4323521)

July 6, 2023

Contents

Abstract

The GoF (Gang-of-Four) design patterns have had a significant impact on software quality and have become a fundamental part of software design. They provide developers with proven solutions to common programming problems by promoting code reusability, abstraction, and maintainability. However, accumulations of artifacts that deviate from the intended structure and principles of design patterns lead to a phenomenon called pattern grime. Technical debt is another leading cause of decreased code quality and has emerged as a result of time constraints, shortcuts, or suboptimal solutions which require refactoring efforts. This research study explores the relationship between design pattern grime and technical debt in Java projects. By analyzing four metrics related to class and modular grime, we investigate the accumulation of grime and its association with technical debt. Two approaches are utilized to measure technical debt: employing static code analysis tools and considering self-admitted technical debt instances extracted from code comments. Moreover, to analyze the relationship between grime and technical debt, we employed two statistical methods: t-test and chi-square. The findings reveal that class grime is associated with a decrease in the accumulation of technical debt. Conversely, technical debt is linked to an increase in the number of alien attributes, while the number of alien public methods remains unchanged. Moreover, the presence of technical debt is linked to a decrease in afferent coupling and an increase in efferent coupling. Through this comprehensive investigation, we contribute valuable insights into the intricate interplay between design pattern grime and technical debt, shedding light on their implications for software systems. These findings provide researchers and practitioners with a solid foundation for further exploration and considerations in the context of system complexity, maintainability, and modularity.

Acknowledgments

Ana

Working on this bachelor's thesis was a stepping stone on my path to becoming a more knowledgeable and equipped software engineer. Therefore, this experience would not have been this valuable and rewarding for me without a number of people that helped or guided me along the way. I would like to express my gratitude to my supervisor, Daniel Feitosa, for guiding me in writing my first contribution to the research world and for showing me how to stay tuned to the things that matter rather than wandering in different directions. This was, without a doubt, one of the challenging things to overcome when there is so much you can discover. I would also like to thank my family for being a constant emotional support and for listening to my "highly technical" conversations even if they do not nearly relate this filed. Lastly, I would like to, unapologetically, thank myself for believing in me and for being consistent throughout this journey of completing my BSc degree.

Karol

I would like to thank my supervisors, Daniel Feitosa and Ayushi Rastogi for their invaluable guidance throughout the process of this bachelor project. I would also like to thank my colleague, Ana, for being extremely helpful and dedicated during the entirety of this project.

1 Introduction And Motivation

In recent years, the field of software engineering has seen significant advancements in terms of the development and maintenance of large-scale software projects. However, as these projects grow in complexity, the presence of various challenges becomes apparent. One such challenge is technical debt (TD), which refers to the accumulated cost of shortcuts, compromises, and suboptimal design decisions taken during software development [1]. While technical debt can be treated as an investment to allow rapid delivery in time-critical situations [2], it should be carefully managed to ensure that it does not become a liability. Specifically, the accumulation of TD in a system can lead to a significant decrease in system quality and performance, higher maintenance costs, eventual system decay, and even lead to customer dissatisfaction [3].

Technical debt can manifest itself in different ways: from written code with poor documentation and design flaws to the accumulation of self-admitted instances of TD. Thus, one way to investigate technical debt is to identify instances of self-admitted technical debt (SATD). The self-admitted aspect emerges from developers *intentionally* choosing a solution to a problem that is known to be suboptimal. Moreover, it can manifest itself in various forms, including code comments that indicate areas for improvement or common issues, tags or annotations within the codebase highlighting potential flaws or unfinished refactoring, and even documented discussions or issues in project management platforms like GitHub[1] or Jira[2]. These forms of SATD serve as explicit reminders of the compromises made during development and indicate areas that require future improvement to reduce the costs of software maintainability.

Another challenge that arises with the growth in system complexity regards design patterns [4]. In object-oriented programming, the GoF design patterns play a crucial role in ensuring structured and efficient design [5]. Specifically, it offers proven solutions to recurring design problems and enables developers to create modular, flexible, and extensible software systems. These patterns are grouped into three categories: creational, structural, and behavioral patterns. Creational patterns focus on object creation mechanisms, structural patterns deal with composition and relationship between objects, and behavioral patterns define the interactions between objects. Consequently, the use of design patterns benefits software developers by enhancing the maintainability, reusability, and overall quality of the software [5]. However, there are circumstances where the layout of some design patterns does not adhere to the standard structure, and the implementation of patterns can sometimes result in a phenomenon known as “pattern grime” which can arise from an accumulation of a buildup of artifacts unrelated to the pattern instance [6]. For instance, grime can be introduced to a Command pattern [7] instance by introducing public methods that are not part of the command interface. This, in turn, leads to a deterioration in the quality and maintainability of software systems [6]. While the concept has been discussed in the software engineering community, empirical studies exploring the relationship between pattern grime and technical debt are limited.

This thesis aims to investigate the potential relationship between pattern grime and technical debt in the context of Java projects. Uncovering the relationship between pattern grime and technical debt can have practical implications for software development teams. It can help identify specific design patterns that are more prone to grime or are associated with higher levels of technical debt than others. Also, the study aims to cover how technical debt is linked to variations in different grime levels. This, in turn, will allow for better decision-making during the design and implementation phases of software projects. By addressing pattern grime proactively, developers can potentially mitigate the accumulation of technical debt and improve the long-term sustainability and design of their software systems.

To achieve these goals, this thesis will undertake a comprehensive study of two Java projects. By examining Java codebases, we aim to collect instances of pattern deviations and technical debt and investigate their relationship at different levels such as class, grime instance, and pattern level which are detailed in Section 3.

[1] https://github.com

[2] https://www.atlassian.com/software/jira

Finally, by adopting a systematic approach and employing appropriate metrics and extraction tools, we aim to provide valuable insights into the relationship between these two phenomena.

1.1 Research Question

The absence of literature directly addressing the relationship between pattern grime and technical debt, as discussed earlier and outlined in Section 2, motivated us to undertake this study. Thus, the primary objective of this research is to investigate and examine the potential correlation between pattern grime and technical debt. As a result, we aim to answer the following main research question:

Is design pattern grime related to technical debt?

Consequently, the main question can be split into the following sub-questions:

RQ1 What is the relationship between the presence of grime and (self-admitted) technical debt?

- RSQ1 Is the presence of grime associated with (self-admitted) technical debt and vice versa?
- RSQ2 Which grime metrics demonstrate a more pronounced association with the presence of (self-admitted) technical debt?

RQ2 What design patterns indicate a higher association between grime and (self-admitted) technical debt?

1.2 Research Outcomes

Through our investigation, we have shed light on the correlation between technical debt and design pattern grime by carefully examining the questions outlined earlier. Our study employed an extensive analysis approach that encompassed multiple levels, including class, instance, and pattern.

In terms of deliverables, we provide the generated datasets (i.e., containing grime and TD instances) along with scripts utilized in this research, ensuring the reproducibility of our methodology for future research. These scripts cover various essential steps, such as dataset generation, TD collection (i.e., through static code analysis) as well as conducting T-Test and Chi-Squared analyses at the class, instance, and pattern levels.

This research endeavor will contribute with valuable insights into the relationship between technical debt and design pattern grime, equipping researchers and practitioners with a foundation for further exploration and understanding in this domain.

1.3 Distribution of Work

In order to accomplish the desired outcomes, our research combines a collaborative effort involving a team of two people. As such, each student was responsible for the delivery of individual tasks that were established at the beginning of the project. Apart from that, some parts of the project were accomplished with an equal contribution from both sides. Lastly, the distribution of work can be seen below:

- Analysis of data: equally divided according to our responsibilities.
- Design pattern and pattern grime extraction: Ana Terna.
- Technical Debt extraction through static code analysis: Ana Terna.
- Self-admitted Technical Debt extraction through comment analysis: Karol Machnik.

The distribution of sections for the thesis paper follows as such:

- Ana Terna: Chapter 1, Sections 2.1, 2.2, 2.3, 2.5, 3.1, 3.2, 3.3, 3.5 - T-test, 4.1, 4.2, 5.1, 5.3, 6, 7
- Karol Machnik: Chapter 1 - ”Introducing SATD”, Section 2.4, 3.4, 3.5 - Chi-Squared, 4.3, 5.2, 5.3 - SATD paragraph, Chapter 6 - SATD paragraph. 7 - SATD Paragraph

1.4 Thesis Outline

This thesis is organized as follows: In Chapter 2, we provide a comprehensive review of the relevant literature and existing studies on pattern grime, technical debt, self-admitted technical debt, and existing tools that aim in extracting their instances in software projects. This literature review will establish a solid foundation for our research and help identify gaps that this thesis aims to address.

Chapter 3 outlines the study design and execution, detailing the methodology, data collection process, and tools employed in our investigation. We explain the criteria used for selecting the Java projects and provide an explanation of the extraction tools, validation process, and analysis techniques applied.

Chapter 4 presents the results of our study, including the assessment of the amount of pattern grime accumulated, the assessment of technical debt and self-admitted technical debt, and any observed correlations between the two. We analyze and interpret the findings based on the statistical evidence collected.

In Chapter 5, we focus on the discussion of the obtained results and their implications by providing an in-depth analysis and interpretation of the findings, highlighting key observations and possible reasons behind our results.

Chapter 6, the research addresses potential threats or limitations that could have influenced the validity or generalizability of the results. Particularly, it aims to identify and discuss any factors or threats that may have impacted the research process or the outcomes of the study.

Finally, Chapter 7 concludes the thesis by summarizing the key findings, discussing their implications, and suggesting areas for future research. We reflect on the significance of the study, and its limitations.

2 Background Literature

This chapter provides background information about the main concepts of our research. In Section 2.1 we discuss the evolutionary aspect of design patterns and pattern grime and explain its implications on software projects. Moreover, in Section 2.2, we present the existing algorithms that detect occurrences of pattern grime and provide a comparison of these tools. In Sections 2.3 and 2.4 we delve into the existing research studies that describe technical debt. Specifically, we list the existing tools employed in identifying occurrences of TD using static code analysis and examining source code comments and discuss the consequences it has on code quality and maintainability. Lastly, in Section 2.5 we go over the main findings concerning the correlation between pattern grime and technical debt.

2.1 Design Pattern Grime

Design patterns are established and widely adopted solutions to recurring programming problems. They provide reusable templates and best practices for designing software systems [5]. As discussed in Chapter 1, software systems can accumulate instances of artifacts (e.g., methods or classes) that are not included in the design pattern rationale. This phenomenon has been defined by Izurieta and Bieman [6] as pattern grime, which is the *"degradation of a design pattern instance due to the accumulation of artifacts unrelated to the instance"*. Design pattern grime can manifest in various forms, such as misplaced or misused design pattern elements (e.g., public methods, attributes), excessive code complexity (e.g. afferent or efferent coupling), redundant or duplicated code, and non-standard implementations. More specifically, Izurieta and Bieman [8] identified concrete forms of grime manifestations. *Class grime* refers to class-related elements such as the number of attributes or public methods. *Modular grime* refers to dependencies between different classes participating in the pattern instance and outsider classes. In turn, this concerns the concept of afferent coupling which regards the incoming dependencies (i.e., how many external entities rely on the functionality provided by the pattern instance) and efferent coupling measures the external dependencies of a class within a pattern instance (i.e., number of classes or components used by a given pattern-participant class). Lastly, *organizational grime* concerns the distribution of classes in a design pattern in various packages. For example, Figure 1 showcases a concrete illustration of modular grime. More specifically, within the context of the Adapter pattern, the presence of modular grime becomes evident as the *Client* directly relies on the *ConcreteAdapter*, rather than being unaware of the adapter's presence. The respective design introduces an unnecessary dependency because this deviation is not compliant with the intended pattern structure.

Grime is associated with numerous flaws and shortcomings within the codebase. For instance, Izurieta and Bieman [9] concluded that grime is associated with extra efforts in testing, maintaining, and extending design patterns. Furthermore, in a later study, Feitosa et al. [10] performed an industrial case study in which they uncovered the factors that influence the accumulation of pattern grime. The study showed that there are strong correlations between a class and modular grime (e.g., dependencies between classes) with a decrease in correctness, performance, and security. Additionally, they found that levels of pattern grime are related to the project size, the type of design pattern that is used, and the individual developer that is responsible for the codebase. For example, in larger projects, the probability of encountering grime is more likely to be higher. Consequently, they found that grime tends to show higher correlations with pattern classes that have a larger number of rule violations such as alien attributes, classes, and unnecessary efferent coupling. Lastly, the levels of pattern grime tend to be lower when the Singleton pattern is concerned and higher when Factory Method is employed. Higher accumulations of grime are also linked to decreased levels of quality attributes such as performance, security, and correctness which is detailed in the following research by Feitosa et al. [4]. Specifically, this study determined that class grime, characterized by the number of alien public methods, alien attributes, and pattern efferent coupling, displayed the strongest correlations to all the quality attributes analyzed. Moreover, highly complex design patterns (e.g., State, Strategy, and Factory Method) tend to accumulate higher levels of grime and violations primarily due to their complex maintainability.

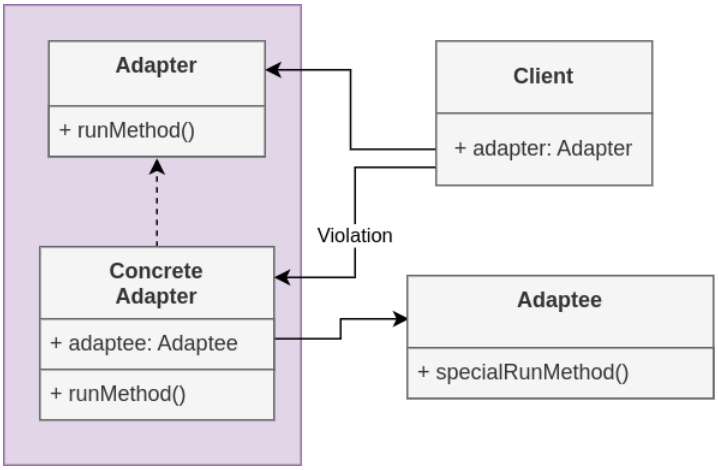


Figure 1: Example of modular grime (afferent coupling) in the Adapter pattern.

2.2 Design pattern detection

The novelty of this research grants us a large degree of freedom over the methods and approaches employed in design pattern detection (DPD). This can include employing tools that extract instances of design pattern deviations or analyzing benchmark datasets of design patterns to detect grime. Benchmark datasets themselves do not detect design patterns automatically. They serve as standardized sets of code examples that contain instances of design patterns, providing a basis for evaluating and comparing different pattern detection techniques or tools [11]. Intensive research has undergone into creating a standardized benchmark for DPD analysis. Fontana et al. [12] introduced a benchmark called DPB that compares different pattern mining methods. Moreover, the research came up with a repository containing different design pattern instances based on user validation which constitutes a promising step toward a standard DPD technique. However, DPB imposes some constraints in the evaluation and comparison of different algorithms from DPD and is limited in design pattern versatility. Another research presented a benchmark based on the automatic generation of testbeds using graph theory which analyzes class diagrams in Java source codes for DPD [13]. Nonetheless, the generated testbeds are not able to encompass some complex Java features and may produce false positives (i.e. unexpected pattern occurrence). As a result, the lack of a standardized benchmark for pattern evaluation, due to limitations of the datasets such as size, precision, and diversity of pattern instances [14], requires a different approach to gathering the necessary data.

Several tools for DPD have been introduced, however, most of them provide a conceptual description of the algorithm while others impose limitations on the output format that is generated [15, 16]. Tsantalis et al. [17] proposed a tool that automates the pattern detection process and is based on a similarity-scoring algorithm (SSA) that calculates the similarity scores between the vertices (i.e., inter-vertex) of the selected pattern and system graph (i.e., holds relationships between classes). The advantages of this approach are the ability to detect pattern instances that do not precisely follow the pattern structure, the open-source nature of the tool, and the intuitive interface as well as the output format. The main limitation of this algorithm is that of focusing on inter-vertex instead of inter-graph (i.e., relationship between multiple graphs) similarity. This limitation was leveraged by J.Dong et al. [18] by creating a new DPD approach focusing on the template matching method for computing the similarity score between the sub-graphs of two graphs. This tool provides a user interface that displays the similarities between different pattern-participating classes and can be used in combination with SSA for future research.

2.3 Technical Debt

Technical debt is a financial metaphor in software engineering introduced by Cunningham (1992) [1] and refers to sub-optimal implementation or design decisions that can lead to short-term benefits. The definition of technical debt has since then been broadened. For instance, in a seminar on TD management in software engineering, Bill Curtis [2] broke down this metaphor with regard to the research and development community. Specifically, he concluded that researchers attribute this definition to sub-optimal design decisions that impact the maintenance of the software, whereas, in industry, developers treat TD as a collection of software flaws that impose a serious concern on software systems. Specifically, the industry puts emphasis on the refactoring cost rather than on a potential decrease in software maintainability, and TD is a serious concern when it is considered damaging enough that any refactoring investments pay off.

Technical debt has since then gained a reputation as a major concern for software engineers. For instance, a 2010 CAST report [19] introduced a starting benchmark on the structural quality of IT business applications. The data in this report was gathered from 288 software systems of 75 organizations from different industries and was used for technical debt estimation based on structural analysis of quality flaws. The report concluded that on average, there is a 2.82$ cost of TD per line of code which, for an average-sized system of 374,000 lines of code in their sample, the total TD is estimated to $1,055,000$\$. Apart from that, TD can lead to degraded performance, low maintainability, delivery delay of the software, and extra costs and efforts during future maintenance which lead to team demotivation and stakeholders dissatisfaction [3].

Furthermore, a significant body of research has focused on developing techniques and tools for identifying, measuring, and managing technical debt, which gave rise to the concept of technical debt management (TDM). Primarily, the main focus has been on prioritizing TD reduction using cost analysis [20]. This study shows that technical debt is paid off as a lucrative investment in software systems and the main trade-off that developers seek is between the cost (i.e., time spent on refactoring) and the quality gained from the refactoring. Recently, Lendarduzzi et al. [21] presented a state of the art of TD prioritization and concluded that there is currently a lack of a validated set of tools for TD prioritization. However, the research found that Architectural Debt and Code Debt [22] are the most prioritized types of debt. As a result, some instances of TD have a higher risk of negatively affecting the quality of the system, thus the need for prioritization is paramount. For future research, valuable information can be disclosed when examining the relationship between highly prioritized TD items and pattern grime. For example, analyzing this relationship can guide developers in their refactoring strategies: treating TD and grime reduction as a whole unit if they are highly correlated.

Researchers have investigated various aspects of technical debt, including its management and relationship to software quality. In the aforementioned study [22], technical debt management incorporates eight phases out of which TD repayment, identification, and measurement received the most extensive research attention. Moreover, the most common approaches for technical debt identification, that have been researched, are code analysis, dependency analysis, check list (i.e., compare with a list of TD scenarios), and solution comparison (i.e., actual solution vs optimal solution). A popular approach to measure TD is through static code analysis (i.e., analyzing the source code of a program without executing it) tools which imply the automatic detection of TD in software artifacts. In a study conducted by Avgeriou et al. [23], a number of static code analysis tools were analyzed based on features such as security, robustness, efficiency, and their popularity among the software engineering community. The research concluded that both SonarQube [3] and CAST [4] are in great demand among developers and offer secure and efficient TD detection. SonarQube, a widely used static code analysis tool, incorporates various rules and metrics to identify instances of technical debt in codebases. For instance, SonarQube can be employed together with tools like GitHub API [5] in order to deliver a stand-alone application that provides a list of technical debt items and code metrics for a large number of public GitHub [6] repositories

[3] https://www.sonarsource.com
[4] https://doc.castsoftware.com/
[5] https://docs.github.com/
[6] https://github.com/

[24].

Other approaches for collecting technical debt items are accomplished through various means, including the analysis of source code comments, issues, and pull requests which gives rise to the concept of self-admitted technical debt. SATD occurs when developers consciously acknowledge the presence of suboptimal code sections, design choices, or other areas that require improvement within the software project. Incorporating SATD analysis in conjunction with static code analysis techniques provides a comprehensive view of the codebase's overall quality and maintainability which, in turn, provides a nuanced understanding of how grime and technical debt are correlated.

2.4 Self-Admitted Technical Debt

Research on SATD in source code was pioneered by Shihab and Potdar in 2014 [25]. They analyzed four large codebases and used srcML [26] to extract code comments from the source code. They manually read through 101,762 comments and found 62 patterns that they believed to be an indicator of technical debt. It was found that $2.4\% - 31\%$ of files contained self-admitted technical debt and only between $26.3\% - 63.5\%$ of SATD instances were removed from projects after they were introduced. Furthermore, this research concluded that developers with more experience have a higher chance of committing code with instances of SATD and that deadline pressure and code complexity have a low correlation with an increase in the amount of SATD instances. Maldonado and Shihab expanded on this research in 2015 [27]. They analyzed more than 30K comments and found that SATD can be classified into five types: design debt, defect debt, documentation debt, requirement debt, and test debt. The most common of these being design debt which made up $42\% - 84\%$ of comments. All of these comments were extracted using JDeodorant [28] and manually analyzed and classified. A further study by Bavota and Russo [29] reinforces the findings of the previous two studies by conducting a differentiated replication on a database of over 600K commits. They used srcML to extract the code comments and used regular expressions with the heuristics defined by Shihab and Potdar [25] to classify the extracted comments. Another study by Zampetti et al. [30] analyzed how SATD is addressed in five Java open-source projects. They looked at 1) whether SATD was being "accidentally" removed by investigating the evolution of SATD instances in source code, 2) how much of SATD is acknowledged in commit messages, and 3) what changes occur in the source code when developers remove SATD. Their findings conclude that 1) By checking whether SATD comments were removed following the removal of i) the whole class or ii) the entire method they found that approximately between $20\% - 50\%$ of SATD comments were accidentally removed. 2) Only around 8% of SATD removal is acknowledged in commit messages, and 3) SATD is often addressed by specific changes to method calls or conditionals.

Up until 2018, the state of the art for extracting instances of SATD was mostly done manually. This changed when Xia et al. [31] proposed an automated approach. They created a framework that preprocesses the text descriptions of comments and extracts features to represent each comment. They used these features to train classifiers and then used these classifiers to predict whether the comment is an instance of SATD. They found that their approach had an average F1-score of 0.737, which improved upon Potdar and Shihab [25] by 499.19%. When compared with a natural language processing-based baseline [32] they also improved the F1-score by 27.95%.

Mário A. de Freitas Farias et al. [33] developed an improved contextualized vocabulary for identifying SATD. They analyzed the patterns of a previously defined contextualized vocabulary and registered their level of importance in identifying SATD items. Then, they performed a qualitative analysis to investigate the relationship between each pattern and type of debt. Finally, they performed a feasibility study using a new vocabulary that they have improved based on the results of previous empirical studies. They found that more than half of the new patterns were considered either decisive or very decisive to detect technical debt items. The patterns were also able to identify different types of SATD as described by Maldonado et al. [27].

More recently, Li et al. [34] have looked into identifying SATD in sources other than source code. This research analyzed 23.6M code comments, 1.3M commit messages, 3.7M issue trackers, and 1.7M pull request sections in 103 open-source projects. They used a Convolutional Neural Network (CNN) [35] approach and achieved an average F1-score of 0.611. Their findings conclude that: 1) instances of SATD are evenly spread among all sources, 2) issues and pull requests are the two most similar sources regarding the shared number of SATD keywords, followed by commits and code comments, and 3) there are four kinds of relations between SATD items in different sections. The current state of the art for mining SATD from source code is outlined by Sabbah et al. [36] and is using different pre-trained language models such as Word2Vec, bidirectional encoder representations from transformers (BERT), and FastText for feature extraction [37, 38, 39]. These features are then used to train classifiers such as random forest, support vector machines, or CNN [40, 41, 42]. This allows for the analysis of large datasets which would not have been possible to do manually.

For the purposes of this project, we chose to use a tool [7] developed by Li et al. [43] to extract and classify code comments. This paper analyzed instances of SATD in an industrial project within 3 sources: code comments, issues, and commit messages. They also interviewed 12 software developers to understand their perception of what SATD really is, how it is managed, and how this management can be potentially improved. Among other things, their research found that: 1) 79.1% of the identified SATD is code/design debt followed by documentation debt and requirement debt at 9.5% and 7.7% respectively, which leaves test debt at 3.7%, and 2) 8 out of 10 interviewees agreed that SATD identified from code comments is indeed TD from their perspective.

2.5 Correlation

While technical debt and design pattern grime are distinct concepts, they share common features and potential inter-dependencies. Both can arise from factors such as time pressure, altered requirements and structure of a system, or insufficient developer experience as discussed above. There is currently a lack of substantial research that shows exactly how TD and grime are correlated. One of the few research papers that discusses the relationship between TD and grime is given by Izurieta et al. [44]. The paper investigates the impact of design pattern decay on system quality and concluded that temporary grime results in higher technical debt scores than persistent grime. However, it presents a couple of limitations: the approach of TD extraction could be improved by looking at other ways of measuring TD (e.g., code comments), the research does not go further into exploring what types of design patterns lead to higher TD items, and lastly, results are inconclusive toward persistent grime. Later on, in his dissertation, Griffith [45] discussed some aspects of the topic, however, the emphasis of the research is largely placed on examining the effects of pattern grime on software quality rather than finding the relationship between technical debt and grime. One of the few researchers that presented a framework in development for conducting an empirical study on grime and its effect on other design defects such as TD was also conducted by Griffith [46]. One of the research hypotheses that the author presented is that "*Grime has a negative effect on the technical debt of a software system as a whole.*" and on "*...pattern realization.*". Consequently, the author proposed a road map for conducting this empirical study, which mainly resembles the approach employed in this research.

To conclude, there is limited research that explores the correlation between design pattern grime and technical debt. As a result, this study aims to fill this gap by examining the extent to which design pattern grime contributes to technical debt accumulation. By analyzing existing tools and methodologies for detecting and measuring design pattern grime and technical debt instances, this research seeks to provide insights into their relationship and potential impact on software quality.

[7] https://github.com/yikun-li/satd-in-industry

3 Study design and execution

The following chapter presents the methodology and procedures used to detect design patterns, pattern grime, and (self-admitted) technical debt, in two Java projects. The chapter's subsections include the detection and analysis of design patterns in Section 3.1, identification of pattern grime in Section 3.2, the use of static code analysis for assessing technical debt in Section 3.3, analysis of comments indicating self-admitted technical debt in Section 3.4, and statistical analysis in Section 3.5. These subsections provide a concise overview of the specific techniques and tools employed to explore and understand these aspects of software development.

For the scope of this research, we decided to focus on two Java projects. First, we analyzed a large and non-trivial system, QuestDB (v7.2)[8], primarily the *core* module of this Java project because it contains the main functionalities of this application. We chose this system because of its large codebase and popularity which consolidates it as a real-world, complex software system, mainly because larger projects tend to have more intricate codebases and a higher potential for grime and technical debt [4]. Moreover, we also considered JHotDraw (v9.0)[9] which is a smaller system and is selected as a benchmark system for pattern detectors that has been employed in a couple of relevant studies [10, 4] where it was used for analyzing and evaluating the design pattern and grime detection tools.

3.1 Design pattern detection

To collect pattern instances, we employed two open-source tools. The first tool, SSA (Similarity Score Analysis - v4.13)[10], which was briefly introduced in Section 2.2, identifies 12 GoF patterns: Adapter/Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State/Strategy, Template Method, and Visitor. The reason for using this tool stems from its ability to accurately detect a substantial number of design patterns, as reported by Tsantalis et al. [17] and the fact that it was previously used in aforementioned research papers for DPD [10, 4]. However, a key limitation of this tool is its inability to identify all classes that participate in the pattern instance (e.g., *ConcreteCreator* of the Factory Method pattern). Consequently, to address the aforementioned limitation of SSA, we used SSA+ (v1.0)[11] which was adopted as an additional tool to detect extended pattern-participant (PP) classes: *Concrete Creator, Product* for Factory Method pattern, *Concrete Prototype* for Prototype pattern, *Leaf* for Composite pattern, *Concrete Decorator, Concrete Component* for Decorator pattern, *Concrete Observer* for Observer pattern, *Concrete State/Strategy* for State/Strategy pattern, *Concrete Class* for Template Method pattern, *Subject* for Proxy pattern. The validation of SSA+ was also conducted in a previously mentioned study [10] and the outcomes of the research served as a motivation for us to incorporate this tool in our research.

3.2 Pattern grime detection

Based on the output of DPD tools, the next step was to detect the grime metrics across all pattern instances. For that, the tool that was utilized in grime detection is *spoon-pttgrime* (v0.1.0)[12], developed by Feitosa et al. [4] which calculates a number of grime metrics for each pattern instance:

- *mg-ca* - pattern instance afferent coupling (Modular grime).
- *mg-ce* - pattern instance efferent coupling (Modular grime).
- *cg-na* - number of attributes that are not part of the original pattern definition (Class grime).
- *cg-npm* - number of public methods that are not part of the original pattern definition (Class grime).

[8] https://github.com/questdb/questdb
[9] https://github.com/wumpz/jhotdraw/tree/9.0
[10] https://users.encs.concordia.ca/~nikolaos/pattern_detection.html
[11] https://github.com/search-rug/ssap
[12] https://github.com/search-rug/spoon-pttgrime

```
<pattern name="Factory Method">
    <instance mg-ca="3" mg-ce="7">
      <role name="Creator" element="io.questdb.std.IOURingFacade" cg-na="0" cg-npm="6"/>
      <role name="FactoryMethod()" element="io.questdb.std.IOURingFacade::newInstance(int):
      io.questdb.std.IOURing"/>
      <role name="ConcreteCreator" element="io.questdb.std.IOURingFacadeImpl" cg-na="2" cg-npm="8"/>
      <role name="Product" element="io.questdb.std.IOURing" cg-na="0" cg-npm="8"/>
    </instance>
    ...
</pattern>
```

Listing 1: Snippet of the XML file with grime metrics per pattern instance

The reason for employing these grime metrics is based on their ability to evaluate various aspects of each grime type. Specifically, analyzing the modular grime aspect provides insights into the dependencies and relationships between classes which provides a clear picture of the overall system quality and pinpoints highly concentrated grime areas. Additionally, class grime detection through metrics such as alien attributes and public methods allows for a more granular analysis at the class level which increases the accuracy of grime which then directly can be correlated to instances of technical debt.

Additionally, we used Python[13] to parse the XML output file of the *spoon-pttgrime* tool, for any subsequent pre-processing operations performed on the dataset. Moreover, Python scripts were utilized for statistical analysis. The reason for choosing this programming language is the wide range of libraries specifically designed for working with XML data (e.g., lxml), for data manipulation and visualization (e.g., pandas, matplotlib), and for conducting statistical analysis (e.g., NumPy, scikit-learn, SciPy). Thus, the first step was to process the XML file by scanning through each instance of the pattern and extracting each PP class and its associated class grime metrics. For example, in Listing 1, the class `IOURingFacade.java` is a PP class with zero alien attributes and six alien public methods and serves as the *Creator* class for the Factory Method pattern. Moreover, the *Creator* class declares the factory method, *newInstance()*, that returns a new Product class, `IOURing.java`, however, the implementation of this method is overwritten by *ConcreteCreator* class `IOURingFacadeImpl.java`. As for the Python script that extracts the metrics at the class level, it processes the *element* tag of each *role* in the XML and extracts each package of the PP classes along with its class grime metrics given by *cg-**. The next step is to consider the entire pattern instances rather than classes. As a result, we created another script that assigns each instance a unique ID and extracts the *mg-** metrics, the number of PP classes, the design pattern it adheres to, and the sum of *cg-** metrics based on the respective PP classes. Overall, a visualization of the design pattern and grime extraction process is depicted in Figure 2. Additionally, the primary benefit of employing these open-source tools lies in their efficacy, specifically in gathering essential data facilitated by a conventional input/output file format across these tools. Furthermore, the accuracy of the results has been successfully verified across a couple of aforementioned studies, further strengthening the reliability of these tools.

[13] https://www.python.org

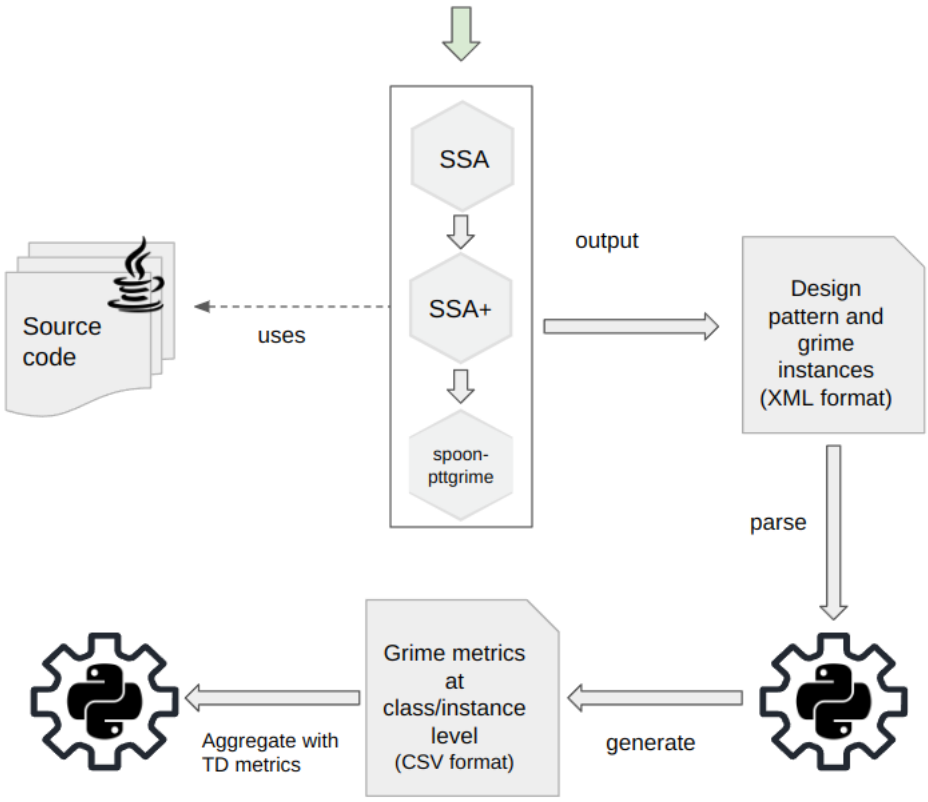


Figure 2: Visualization of the design pattern and grime detection process.

3.3 Static code analysis on technical debt

In this section, we describe the process of extracting technical debt instances in Java projects using SonarQube. SonarQube is an open-source static code analysis tool that supports various programming languages and has been widely used in the industry and in the software engineering community [47]. It provides a comprehensive set of features to analyze and assess code quality and incorporates a range of predefined coding rules and metrics to identify potential issues, bugs, vulnerabilities, and technical debt. Specifically, each rule introduces a new TD item and the severity level of the respective code smell. For the scope of this research, we are interested in *minor*, *major*, and *critical* severity levels [48] of each rule. In order to collect each TD item in a Java class, we discard bugs and vulnerabilities but rather focus on SonarQube build-in code smells metric that is introduced as a result of violated rules in each class.

For this research, we created a bash script, publicly available on GitHub [14], to collect TD instances. It takes as input a list of GitHub repositories, automates the cloning of these repositories (i.e., creating a copy of the project on a local machine), compiles the source code, and uses the Maven[15] *package* build command to translate the project into a binary format. Next, the script generates a *sonar-project.properties* file which contains the configurations of the project and the necessary credentials for connecting to an active SonarQube server. In order to run the SonarQube analysis, the script generates the configuration settings for SonarScanner [16] which is a command-line tool that is used to initiate and perform code analysis on the respective project.

[14] https://github.com/anaterna/BscThesis
[15] https://maven.apache.org/
[16] https://docs.sonarqube.org/9.8/analyzing-source-code/scanners/sonarscanner/

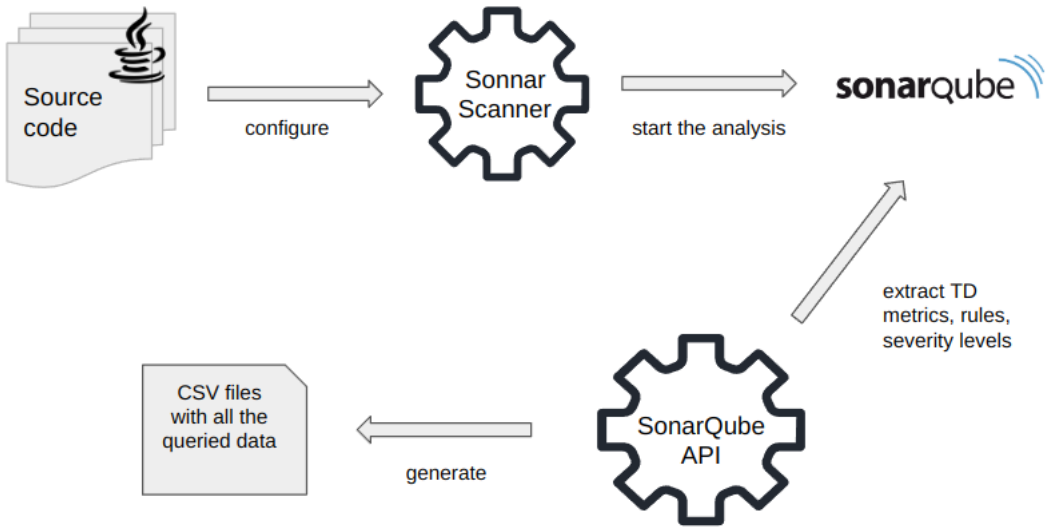


Figure 3: Visualization of the technical debt detection process.

SonarQube offers a comprehensive web API [17] that provides endpoints for retrieving information about code components, metrics, and issues detected during analysis, including technical debt. By leveraging the SonarQube API, we successfully retrieved the necessary data pertaining to each class within the software project. Moreover, the script performs all the necessary requests to the web API in order to retrieve the following information based on each *project-key* (i.e., identifies an analyzed project) that was assigned to each project during analysis:

- for each class: classpath, rule ID, description of the rule, severity level
- for each class, the number of *critical*, *major*, *minor* issues which summed up, give the total amount of TD

Thus, a concise explanation of the static code analysis process is depicted in Figure 3 which presents an overview of the workflow utilized for extracting technical debt instances from SonarQube based on the aforementioned execution steps.

3.4 Comment analysis on self-admitted technical debt

To extract and classify source code comments from Java projects we used the reproduction package developed by Li et al. [43]. The tool uses a Multitask Text Convolutional Neural Network (MT-Text-CNN) approach. It is essentially a Text Convolutional Neural Network (Text-CNN)[49] approach but the output layer has been modified to be task-specific. Text-CNNs are a novel approach that has been used in previous SATD detection works [50, 51]. See the paper by Li et al. [34] for more information.

The classifier is able to extract source code comments from Java classes and classify them into four categories: *code|design debt, requirement debt, documentation debt*, and *test debt*. Examples of what the tool qualified as each classification are:

code|design debt - “todo: introduce fairness factor”

requirement debt - “not yet implemented, taking the non-correlated sub-query out as a join”

[17] https://docs.sonarqube.org/latest/ extension-guide/web-api/

documentation debt - "An unfinished sample drawing editor with limited support for the href=link"

test debt - "todo: test key write failure"

This tool was run for every file ending in the `.java` extension in a selected directory and every comment with their respective classification was saved into a CSV file. Then, another script was used to calculate the amount of each instance of SATD and the total count per file. This file was later merged with the pattern grime and SonarQube results for analysis.

3.5 Statistical analysis

This section aims to give an overview of the various statistical methods used to investigate the relationship between grime and TD (as detected by both source code analysis and NLP). Hence, we introduce two statistical techniques, including t-test, and chi-square analysis. By utilizing these methods, we aim to assess the influence that grime has on (SA)TD and how (SA)TD dictates the concentration of grime in two Java projects. The analysis seeks to uncover potential correlations, trends, and differences between the two phenomena, providing valuable insights into the nature of their relationship. Lastly, this section aims to describe the necessary foundation and steps to be able to utilize these tools. Moreover, each statistical method is applied to answer a specific research question or strengthen any subsequent results.

T-test

To answer the first two questions of this research that focus on assessing the influence of grime on TD growth and how the presence of TD impacts grime accumulation, we perform an independent two-sample t-test analysis on different combinations of the dataset. T-test is one of the most commonly adapted statistical tests [52] and is utilized to determine if there is a significant difference between the mean of the two groups and thus allows us to gain insights into the relationship between the two phenomena. Specifically, it can indicate whether there is a significant difference in TD accumulations based on the presence or absence of grime and vice versa. As a result, the analysis was conducted as follows:

Q1 : **Is the presence of grime associated with (self-admitted) technical debt?**

(a) Split the dataset into classes that contain TD and no (class/modular) grime and classes that contain both TD and (class/modular) grime.

(b) Perform a t-test analysis on the two groups of TD

Q2 : **Is the presence of (self-admitted) technical debt associated with grime?**

(a) Split the dataset into classes that contain (class/modular) grime and no TD and classes that contain both TD and (class/modular) grime.

(b) Perform a t-test analysis on the two groups of (class/modular) grime

To analyze the results of the t-test, we evaluate the *p-value* which indicates the probability of observing a difference in the means that resulted by chance and analyze the value of the *t-statistic* to determine the direction (i.e., the orientation of the mean) and magnitude (i.e., the extent of the change) of the difference between the means. Specifically, if the *p-value* is below the significance level (i.e., $\alpha = 0.05$ standard), it indicates that the difference between the groups is statistically significant which allows us to reject the null hypothesis of no significant difference, hence, we conclude that the presence of grime influences the amount of TD with regards to RSQ1. Moreover, if the observed *t-statistic* is positive, then the first group (only TD) registered a higher mean, while a negative *t-statistic* suggests that the second group (TD and grime) has a higher mean that the first group. Finally, we will examine the absolute value of the *t-statistic* to determine the actual difference between

the means of the two groups that are compared.

Chi-squared test

The Chi-Squared goodness of fit test [53] is used to determine whether there is an association between two categorical data. It is used to check whether the differences in the observed and expected frequencies are statistically significant or simply up to chance. In our case, the chi-squared goodness of fit test can tell us whether the presence of pattern grime is related to technical debt in the analyzed source code. In order to achieve this, we perform the following steps:

1. Create the categorical data: $<$ *hasGrime, has(SA)TD* $>$, where *hasGrime* contains four groups of categorical data representing each grime metric and *has(SA)TD* contains two groups representing TD analyzed through static code analysis and SATD detected through comment analysis. The categorical data is in binary format (i.e., 1 - presence, 0 - absence).
2. Perform the Chi-Squared goodness of fit test on the datasets.

To analyze the results of the Chi-Squared test we must look at the *p-value*, and the *test statistic*. The *p-value* measures the level of statistical significance and indicates whether there is evidence to reject the null hypothesis and the test statistic is a numerical measure of the differences between the observed and predicted values. We determine the significance of the results by comparing the *p-value* with the significance level (i.e., $\alpha = 0.05$) and assess the results in the following manner:

H_O : *p-value* > 0.05 - there was no significant association between grime and (SA)TD.

H_A : *p-value* < 0.05 - the presence of grime displays a strong association with (SA)TD accumulations.

4 Results

In this section, we present the findings of the statistical analysis conducted to investigate the relationship between (SA)TD and grime. To examine this relationship, we employed two widely used statistical methods: the t-test and the chi-square test. The t-test was employed to evaluate mean differences between TD and grime groups, focusing on the class, instance, and pattern level. On the other hand, the chi-square test was utilized to assess the association between TD and grime in terms of categorical data. Therefore, by investigating various dataset scenarios, we aimed to gain a comprehensive understanding of how the presence of grime affects TD accumulation and vice versa across two Java projects.

4.1 Data analysis

Based on the data collection phase, we analyzed over 2600 classes and 1600 instances containing TD and grime in JHotDraw and QuestDB. In Table 1, we list all the design patterns per project along with the total number of grime metrics and TD accumulation at the pattern instance level because metrics such as modular grime that concern the number of incoming and outgoing dependencies can be detected only at each instance of the pattern. Thus, we notice that the State pattern incurred the highest amount of grime in both projects based on the number of instances (i.e., 961 in total), grime metrics (e.g., mg-ca = 325327, mg-ce = 27573, cg-na = 11178, cg-npm = 38700) for both class and modular grime, TD analyzed with SonarQube (i.e., 691 in total) and SATD (i.e., 116 in both projects). Moreover, a high increase in grime and TD can be observed in design patterns such as (Object)Adapter, Proxy (i.e., in QuestDB), Bridge, and Decorator. Contrarily, design patterns such as Strategy, Proxy2, and Observer are the least utilized patterns based on the number of recorded instances. Nevertheless, patterns like Factory Method, Prototype, and Composite registered a low number of instances, however, lead to an increased amount of modular (e.g., mg-ca: 3332, mg-ce: 1671 - Factory Method in total), class grime (e.g., cg-na: 708, mg-ce: 2733 - Factory Method in total), TD (e.g., 63 from SonarQube and 24 SATD items) which, for the Factory Method pattern. this could be as a result of a high number of *ConcreteCreator* classes (i.e., 586 pattern classes in total), which in turn can lead to increased coupling with the *Product* classes.

QuestDB					**Pattern Info**						
Pattern	mg-ca	mg-ce	cg-na	cg-npm	TD	Minor	Major	Critical	Instances	Classes	SATD
Factory Method	2595	1527	618	2115	46	1	17	28	17	572	12
Singleton	649	526	189	473	16	0	1	14	120	118	14
(Object)Adapter	96753	15093	8869	16238	220	26	94	97	469	395	39
Decorator	5015	4890	3693	8542	38	4	1	33	43	655	21
State	317989	24715	10333	36057	258	35	105	117	880	689	65
Strategy	2	6	10	7	0	0	0	0	1	1	0
Bridge	8352	777	466	1292	57	15	26	15	37	45	4
Template Method	268	965	715	592	15	0	5	9	35	124	10
Proxy	13799	1200	1226	1781	41	5	16	19	31	39	3
Proxy2	7	11	0	17	2	2	0	0	1	3	0
JHotDraw					**Pattern Info**						
Factory Method	537	144	90	618	17	12	1	1	4	14	12
Prototype	1032	856	233	1122	66	18	25	16	13	32	17
Singleton	65	149	45	94	55	11	34	10	13	12	3
(Object)Adapter	1564	1127	308	1017	168	42	59	59	25	37	28
Composite	888	664	223	1082	23	8	6	5	5	19	7
Decorator	401	204	71	315	6	0	0	4	3	18	9
Observer	10	32	22	53	12	2	4	5	2	4	1
State	7338	2858	845	2643	433	72	209	144	81	91	51
Bridge	2255	880	309	1072	113	30	43	35	22	25	8
Template Method	178	589	136	355	112	31	48	25	15	27	5

Table 1: Summary of design patterns associated with grime and TD

In Table 2, we present a general statistical overview of grime and TD variables for each project that was analyzed in order to characterize our working dataset. We notice that the class grime metric, *cg-na*, shows significant variability between the projects. The maximum value for QuestDB is 3525, indicating a relatively higher presence of alien attributes compared to JHotDraw with a maximum value of 180. Similarly, the metric *cg-npm* also exhibits a difference between the projects. Namely, QuestDB has a higher maximum value of 20961, suggesting a larger number of alien public methods compared to JHotDraw with a maximum value of 1351. These results can indicate a sign of bad practices based on the fact that JHotDraw is considered a benchmark project that was designed for educational purposes whereas QuestDB is a byproduct of industry needs. Another reason can be the complexity of the software with QuestDB registering a substantially higher project size, number of grime instances, and TD accumulations than JHotDraw. In terms of *mg-ca* and *mg-ce*, QuestDB registered higher values than JHotDraw, indicating an increased level of incoming and outgoing dependencies. The *TotalBrokenRules* metric, representing the amount of technical debt, detected with SonarQube, has a higher accumulation in QuestDB (i.e., 379), implying a potentially larger amount of technical debt compared to JHotDraw (i.e., 105). Moreover, the metrics *MinorRules*, *MajorRules*, *CriticalRules* and *InfoRules* provide insights into the distribution of different types of TD and generally exhibit higher maximum values in QuestDB compared to JHotDraw, suggesting a potentially higher occurrence of rule violations. Furthermore, an interesting observation can be made regarding the violation of rules in both QuestDB and JHotDraw. Specifically, Table 2 reveals that QuestDB has a higher incidence of critical rule violations, indicating potential issues that significantly impact the system's maintenance. Conversely, JHotDraw exhibits a greater number of minor rule violations, suggesting a tendency towards less severe issues that may have a lower overall impact on the system. Furthermore, the mean score of SATD is relatively low, suggesting that the majority of examined comments do not explicitly mention the presence of technical debt as acknowledged by developers. However, it is important to note that specific areas within the codebase exhibit higher SATD scores, such as 50 in QuestDB and 20 in JHotDraw. These elevated scores signify the existence of significant technical debt in those particular areas. **To conclude, based on the aforementioned observations, it can be inferred that QuestDB demonstrates higher levels of both grime and (SA)TD in comparison to JHotDraw, which serves as a benchmark project.**

4.2 Grime-technical debt correlation

In this section, we present the results of two statistical methods described in Section 3.5 and utilized in the scope of two Java projects. Through a comprehensive analysis of the mean and correlation, we can address our research questions and gain a deeper understanding of the interplay between design pattern grime and technical debt. Thus, the results are organized into subsections that focus on different aspects of the analysis.

4.2.1 Analysis of the mean

The first subsection of the study explores the analysis of the mean. This involves using t-test to examine whether there are significant differences in means and distributions between various groups of grime metrics and technical debt. That being said, t-test was applied to three different datasets (i.e., we considered classes that have only TD or grime and both):

- grime and TD accumulations at the class level in both/QuestDB/JHotDraw projects combined
- grime and TD accumulations at the pattern instance level in both/QuestDB/JHotDraw projects
- grime and TD accumulations at the pattern level in both projects

Class level

Variable	Project	Minimum	Maximum	Mean	Std. Deviation
cg-na	QuestDB	0.00	3525.00	20.23	114.03
	JHotDraw	0.00	180.00	14.17	27.14
cg-npm	QuestDB	0.00	20961.00	52.00	628.50
	JHotDraw	0.00	1351.00	51.99	140.49
mg-ca	QuestDB	0.00	839.00	273.60	324.50
	JHotDraw	1.00	230.00	78.39	75.59
mg-ce	QuestDB	0.00	884.00	30.53	48.57
	JHotDraw	3.00	168.00	41.22	35.93
TotalBrokenRules	QuestDB	1.00	379.00	15.29	37.80
	JHotDraw	1.00	105.00	7.51	11.06
MinorRules	QuestDB	0.00	45.00	1.24	4.56
	JHotDraw	0.00	104.00	2.47	7.41
MajorRules	QuestDB	0.00	59.00	2.00	5.75
	JHotDraw	0.00	52.00	2.69	4.70
CriticalRules	QuestDB	0.00	377.00	11.87	34.13
	JHotDraw	0.00	23.00	2.20	3.28
InfoRules	QuestDB	0.00	36.00	0.13	1.67
	JHotDraw	0.00	4.00	0.02	0.27
SATD	QuestDB	0.00	50.00	0.12	1.13
	JHotDraw	0.00	20.00	0.43	1.25

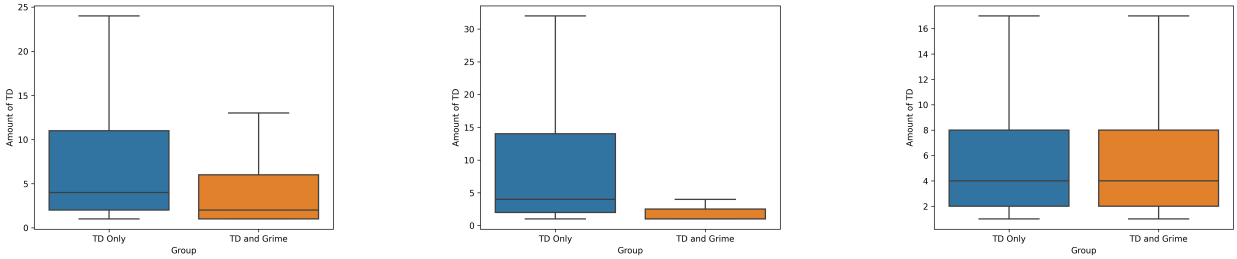
Table 2: Descriptive statistics per project.

Based on Table 3, in the combined JHotDraw and QuestDB datasets, we observe a significant difference in TD accumulation between the TD group and the TD-grime group. **Specifically, TD that is present in grime instances exhibits a lower mean accumulation compared to the group where only TD was found, indicating that the presence of grime might contribute to a decrease in TD.** Moreover, in the QuestDB project, we found a pronounced difference in TD accumulation between the TD group and the TD-grime group. The grime group exhibited a substantially lower mean TD accumulation compared to the TD group. This indicates that **the presence of class grime may significantly affect TD accumulations in QuestDB, as evidenced by the substantial differences in mean TD values** (i.e., from a mean of 17.21 to 2.88). Additionally, the low *p-value* (i.e., *p-value* < 0.001) and large t-statistic (i.e., 7.16) indicate a strong statistical significance, suggesting that the observed difference in means is unlikely to occur by chance alone. Moreover, the t-test results also reveal notable differences in the standard deviation of TD values between the TD group and the TD with grime group in QuestDB (i.e., 40.26 vs. 4.24). This implies that **the presence of grime may not only lead to lower mean TD values but also contributes to a decrease in the variability of TD accumulations.** The smaller standard deviation suggests a more consistent and controlled TD environment when grime is present in QuestDB.

In contrast, the t-test **results for the JHotDraw dataset reveal that the impact of grime on TD accumulations is not statistically significant** (i.e., 7.79 vs. 6.79, *p-value* = 0.41). Both groups exhibited a relatively similar mean of TD accumulations, implying that the presence of grime may not have a substantial impact on TD accumulation in this specific project. Moreover, the size of the JHotDraw project (i.e., 413 classes analyzed) which is considerably smaller than in QuestDB (i.e., 1724 classes analyzed), justifies the t-test results of significant difference in the means of TD obtained in the combined datasets which are depicted in Figure 4a. Hence, the QuestDB dataset holds greater significance in shaping the overall outcomes due to the heterogeneous nature (i.e., non-uniformity in terms of projects) of the combined dataset, which predominantly represents QuestDB. The discrepancy between these projects can also be inferred from the box plots in Figure 4b and 4c which provide a visualization of the TD spread in QuestDB and JHotDraw.

Dataset	Group	Mean	Std. Deviation	p-value	t-statistic	Dataset size
JhotDraw and QuestDB	TD	13.74	33.09			
	TD and grime	5.16	7.73	< 0.01	6.11	2137
QuestDB	TD	17.21	40.26			
	TD and grime	2.88	4.24	< 0.01	7.16	1724
JhotDraw	TD	7.79	11.70			
	TD and grime	6.79	9.15	0.41	0.83	413

Table 3: Impact of class grime on TD accumulation: summary of t-test results at class level



(a) TD in QuestDB and JHotDraw (b) TD in QuestDB (c) TD in JHotDraw

Figure 4: Distribution of TD in TD-only vs TD-grime dataset at class level

In Table 4, we present the results of the t-test analysis that showcase the mean variation of class grime in the presence and absence of TD. For JHotDraw, no fluctuations in the mean of both class grime metrics are observed. **In the case of QuestDB alone, t-test results show a significant difference in the mean number of alien attributes** (i.e., *cg-na*) with the presence of TD. The low *p-value* and the negative *t-statistic* indicate that the presence of TD is associated with a higher mean accumulation of *cg-na*. However, **there is no substantial difference observed in the mean value of the number of alien public methods** (i.e., *cg-npm*). Consequently, we can infer that the presence of **TD is associated with an increase in the number of alien attributes** in both the combined dataset and QuestDB. Furthermore, the visual representation of the distribution of grime metrics based on the combined datasets in Figure 5a indicate a high variability of *cg-na* and higher median in the presence of TD while in Figure 5b have an almost identical median of *cg-npm* distribution. Moreover, we included additional representations on the distribution of class grime per individual project in Figure 11.

(a) cg-na in QuestDB and JHotDraw (b) cg-npm in QuestDB and JHotDraw

Figure 5: Distribution of class grime in grime-only vs TD-grime dataset at class level

Dataset	Group	Mean	Std. Deviation	p-value	t-statistic	Dataset size
JhotDraw and QuestDB	cg-na	17.98	43.58			
	cg-na and TD	75.87	310.68	0.02	-2.29	2137
	cg-npm	52.20	638.85			
	cg-npm and TD	72.52	242.17	0.45	-0.76	
QuestDB	cg-na	18.03	43.45			
	cg-na and TD	153.32	460.59	0.02	-2.39	1724
	cg-npm	51.41	654.48			
	cg-npm and TD	104.72	318.10	0.23	-1.21	
JhotDraw	cg-na	17.03	22.74			
	cg-na and TD	16.43	31.87	0.90	0.12	413
	cg-npm	67.50	112.20			
	cg-npm and TD	49.10	164.90	0.40	0.80	

Table 4: Impact of TD on class grime accumulation: summary of t-test results at class level

Instance level

Table 5 presents the results of the t-test at the instance level, where each grime instance registers two values that describe modular grime: grime afferent coupling (i.e., *mg-ca*) and grime efferent coupling (i.e., *mg-ce*) introduced in Chapter 3. The results of the mean variation of these two modular grime metrics are based on datasets that contain only modular grime and datasets where modular grime is accompanied by TD. Thus, the aim is to determine if the presence of TD is associated with a change in the mean of modular grime. In the combined dataset (i.e., 1810 instances) we can notice that there is a decrease in the mean of *mg-ca* (i.e., from 304.54 to 129.78) in the presence of TD as the *p-value* is less than 0.01 and the *t-statistic* of 14.00 indicate a statistically significant difference between the means of the two groups. As instances of grime in QuestDB are considerably higher in number than in JHotDraw (i.e., 1628 vs 182), a similar decrease can be observed in the mean amount of grime afferent coupling in the presence of TD in QuestDB. These findings suggest that **the presence of TD might have a significant impact on the decrease in the number of incoming dependencies (i.e., afferent coupling) in QuestDB**.

Dataset	Group	Mean	Std. Deviation	p-value	t-statistic	Dataset size
JhotDraw and QuestDB	mg-ca	304.54	340.62			
	mg-ca and TD	129.78	184.47	< 0.01	14.00	1810
	mg-ce	22.44	19.66			
	mg-ce and TD	55.85	78.66	< 0.01	-9.50	
QuestDB	mg-ca	306.73	341.22			
	mg-ca and TD	154.42	215.13	< 0.01	10.14	1628
	mg-ce	22.32	19.50			
	mg-ce and TD	63.00	92.08	< 0.01	-8.12	
JhotDraw	mg-ca	50.18	60.60			
	mg-ca and TD	80.21	76.21	0.14	-1.57	182
	mg-ce	36.00	31.66			
	mg-ce and TD	41.56	36.24	0.59	-0.56	

Table 5: Impact of TD on modular grime: summary of t-test results at instance level

Similarly, for the *mg-ce* variable, the t-test results revealed a significant difference with the presence of TD

in the combined dataset (i.e., *p-value* < 0.01). The mean value of *mg-ce* is lower compared to the TD-grime group (55.85 in JhotDraw and QuestDB combined dataset, 63.00 in QuestDB alone). The negative *t-statistic* values, respectively, indicate a substantial increase in the means. Hence, these results suggest that **the presence of TD in modular grime efferent coupling is associated with an increase in the mean and variation (i.e., high standard deviation of 19.50 vs 92.08) in QuestDB.** In contrast, in the JhotDraw dataset, the t-test results did not show statistically significant differences between the TD and both of the modular grime metrics groups (i.e., *p-value* > 0.05) which can be justified by the small project size and diversity in comparison to QuestDB.

Pattern instance level

To answer RQ2, we perform a t-test analysis for each subset of design patterns at the instance level because each instance represents a violation of a given design pattern structure and captures both modular and class grime, thus providing a representative dataset of various grime patterns. Consequently, each instance in the dataset contains modular grime, at least one of the class grime metrics, and zero or more TD, hence, the t-test results presented in Table 6 shed light on the impact of TD on modular grime and Table 7 provides insights into the impact of TD on class grime at the level of design patterns.

Firstly, when considering the *mg-ce* variable, it can be noticed that the (Object)Adapter and State patterns exhibited significant differences between the TD and grime groups (i.e., *p-value* < 0.05 and *t-statistic* < 0.00). These findings suggest that **the presence of TD in (Object)Adapter and State pattern instances is associated with a higher occurrence of efferent coupling**, highlighting the importance of addressing TD in these patterns to potentially mitigate the accumulation of grime. Moreover, ***mg-ca*** **registered a decrease in the mean with respect to the presence of TD** (i.e., *p-value*< 0.01 and *t-statistic* > 0.00) in both patterns which explains the overall outcome of Table 5.

Secondly, the Bridge pattern also demonstrated significant differences in the *mg-ce* variable, however, there is no significant difference in the accumulation of *mg-ca* (i.e., *p-value* > 0.05). In contrast, the other patterns examined, including Factory Method, Singleton, Decorator, Template Method, and Proxy, did not exhibit significant differences in either *mg-ce* or *mg-ca* variables. These patterns showed t-test statistics and p-values that did not reach statistical significance, indicating that the presence of TD is not associated with the accumulation of grime in these particular patterns. One interesting thing to note regards the Singleton pattern which is the third most used design pattern in the dataset, however, there are no significant differences in the grime variations as observed in the Bridge pattern which is lower in representation. Thus, the main reason behind this effect concerns the complexity of the pattern itself.

Design Pattern	t-statistic (mg-ce)	p-value (mg-ce)	t-statistic (mg-ca)	p-value (mg-ca)	Dataset size
Factory Method	-1.262	0.246	-0.134	0.895	21
Singleton	-1.751	0.102	-1.425	0.176	130
(Object)Adapter	-8.852	< 0.01	3.435	< 0.01	494
Decorator	-0.371	0.714	-0.986	0.332	46
State	-9.010	< 0.01	16.989	< 0.01	960
Bridge	-2.878	0.007	1.322	0.193	59
Template Method	-2.759	0.011	-0.607	0.548	47
Proxy	-2.522	0.038	-0.624	0.545	31

Table 6: T-test results: the influence of TD on modular grime at pattern level

Among the design patterns analyzed, the (Object)Adapter pattern stands out with significant differences in both *cg-na* and *cg-npm* variables (i.e., *t-statistic*> 0 and *p-value* < 0.05). These results indicate a substantially high association between TD and the accumulation of class grime in (Object)Adapter pattern instances. The State

pattern also demonstrated a significant relationship with TD in terms of class grime. These findings suggest a strong association of TD with an increase in the accumulation of class grime. On the other hand, patterns such as Factory Method, Singleton, Decorator, Bridge, Template Method, and Proxy did not show statistically significant differences in either *cg-na* or *cg-npm* variables which can be a result of significantly lower instances of these patterns or the simplicity of the structure (e.g., Singleton pattern). It is worth noting that the Decorator pattern exhibited a significant decrease in the number of alien attributes, as indicated by the positive *t-statistic* and *p-value* of 0.001. However, there was no significant change in the *cg-npm* metric.

Design Pattern	**t-statistic (cg-na)**	**p-value (cg-na)**	**t-statistic (cg-npm)**	**p-value (cg-npm)**	**Dataset size**
Factory Method	-0.901	0.382	-1.401	0.202	21
Singleton	-1.714	0.109	-2.557	0.021	130
(Object)Adapter	-7.391	< 0.01	-4.983	< 0.01	494
Decorator	3.580	0.001	0.272	0.788	46
State	-6.776	< 0.01	-1.104	0.271	960
Bridge	-0.031	0.975	-1.700	0.095	59
Template Method	-0.772	0.448	-0.860	0.398	47
Proxy	-1.517	0.154	0.639	0.528	31

Table 7: T-test results for class grime based on TD at pattern level

4.2.2 Correlation analysis

The chi-square test was performed to analyze the relationship between the presence of TD and grime in two Java projects. As such, we created a couple of categorical datasets based on the presence or absence of TD, modular, and class grime to identify any associations or dependencies between these TD-grime variables. The chi-square test helps to determine whether the observed frequencies of the two examined variables differ significantly from what would be expected if they were independent and if there is evidence to suggest that the presence of TD is related to the presence of grime and vice versa. Hence, the purpose of conducting this statistical analysis is to reinforce the above findings and provide stronger evidence in response to RQ1, RSQ1, and RSQ2.

In Figure 6, the chi-square test provides a statistical measure to assess the relationship between TD and class grime. Overall, **there is a significant relationship between class grime and TD in both QuestDB and JHotDraw projects**. The chi-square statistics for both *cg-npm* and *cg-na* metrics are considerably large, indicating a strong association between class grime type and TD. This suggests that there is a statistically significant association between the presence of TD and the accumulation of grime attributes and public methods in both projects. When considering QuestDB individually, the chi-square statistics for class grime metrics are again highly significant ($p\text{-}value < 0.001$). Similarly, in the case of JHotDraw, the chi-square values for *cg-npm* and *cg-na* are statistically significant (i.e. $p\text{-}value < 0.001$). This implies that TD is significantly associated with the accumulation of grime in terms of both attributes and public methods within the QuestDB and JHotDraw project.

In Figure 7, when considering the combined dataset of QuestDB and JHotDraw, the chi-square test indicates a marginally significant relationship between TD and mg-ce (i.e., $p\text{-}value = 0.05$). This suggests that **the presence of TD may have some association with the accumulation of efferent coupling** in the context of modular grime. However, **for *mg-ca*, the chi-square test does not show a significant relationship with TD (i.e., $p\text{-}value > 0.05$)**. Focusing on the QuestDB project individually, the chi-square test results show that neither of the grime metrics has a significant relationship with TD. Similarly, in the case of JHotDraw, both mg-ce and mg-ca variables exhibit no significant relationship with TD based on the chi-square test (i.e. *p-*

value) = 1.0). This implies that the presence of TD does not have a notable association with the accumulation of modular grime in terms of efferent and afferent coupling within the QuestDB and JHotDraw project.

Project	grime type	chi-square	p-value
QuestDB and JHotDraw	cg-npm	1351.01	< 0.01
	cg-na	609.52	< 0.01
QuestDB	cg-npm	1211.20	< 0.01
	cg-na	502.00	< 0.01
JHotDraw	cg-npm	97.59	< 0.01
	cg-na	64.24	< 0.01

Figure 6: Chi-square results for class grime and TD

Project	grime type	chi-square	p-value
QuestDB and JHotDraw	mg-ce	3.88	0.05
	mg-ca	0.83	0.36
QuestDB	mg-ce	1.67	0.19
	mg-ca	0.37	0.54
JHotDraw	mg-ce	0.0	1.0
	mg-ca	0.0	1.0

Figure 7: Chi-square results for modular grime and TD

4.3 Grime-SATD correlation

To gain a greater understanding of SATD in the presence of pattern grime we chose to also analyze HBase[18] in addition to JHotDraw and QuestDB. This gave us over 8400 Java files at class level and around 2200 pattern instances to analyze.

4.3.1 Analysis of the mean

Class Level

Dataset	Group	Mean	Std. Deviation	p-value	t-statistic	Dataset size
JhotDraw and QuestDB	SATD	2.07	3.15			
and HBase	SATD and grime	2.19	2.67	0.47	-0.73	381
JhotDraw and QuestDB	SATD	2.07	3.84			
	SATD and grime	1.72	1.86	0.26	1.12	100
HBase	SATD	2.06	2.93			
	SATD and grime	2.36	2.90	0.14	-1.45	281
QuestDB	SATD	2.36	5.10			
	SATD and grime	1.83	2.225	0.36	0.91	55
JhotDraw	SATD	1.823	2.25			
	SATD and grime	1.58	1.28	0.38	0.87	45

Table 8: Impact of class grime on SATD accumulation: summary of t-test results at class level

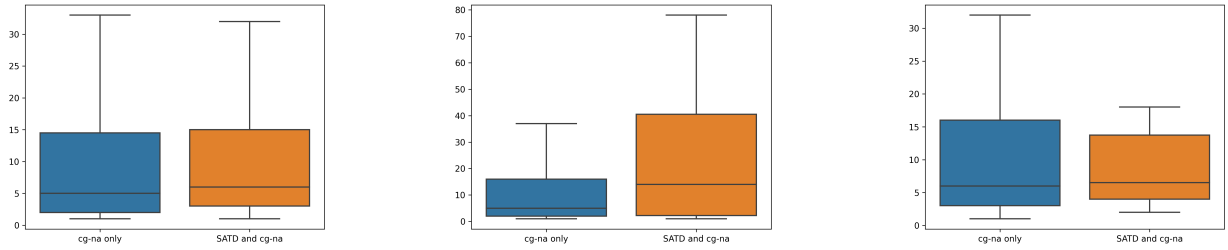
Based on Table 8 none of the results showcase a significant p-value. The T-Test was run by splitting our data into two partitions: 1) Data containing only SATD (i.e. no grime), and 2) data containing grime and SATD. This indicates that the presence of grime does not seem to increase the amount of SATD at class level.

[18] https://github.com/apache/hbase

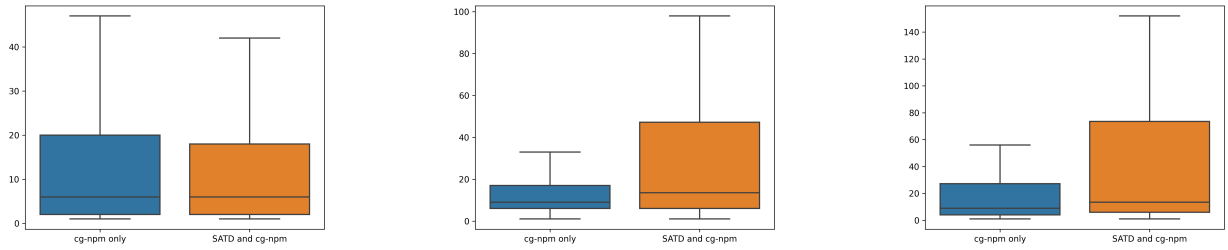
Dataset	Group	Mean	Std. Deviation	p-value	t-statistic	Dataset size
JhotDraw and QuestDB	cg-na	22.51	103.49			
and HBase	cg-na and SATD	32.92	142.52	0.21	-1.24	320
	cg-npm	43.09	506.04			
	cg-npm and SATD	68.09	290.09	0.19	-1.31	357
JhotDraw and QuestDB	cg-na	17.98	43.58			
	cg-na and SATD	34.59	92.29	0.40	-0.83	80
	cg-npm	50.38	621.48			
	cg-npm and SATD	111.32	369.46	0.15	-1.44	92
HBase	cg-na	17.99	50.57			
	cg-na and SATD	32.62	155.86	0.16	-1.40	240
	cg-npm	29.87	140.04			
	cg-npm and SATD	53.08	256.03	0.16	-1.40	265
QuestDB	cg-na	26.70	132.53			
	cg-na and SATD	42.76	120.25	0.40	-0.84	42
	cg-npm	50.76	647.75			
	cg-npm and SATD	132.02	473.97	0.23	-1.18	52
JhotDraw	cg-na	13.24	18.62			
	cg-na and SATD	25.55	44.74	0.11	-1.64	38
	cg-npm	46.25	142.20			
	cg-npm and SATD	84.40	153.05	0.17	-1.37	40

Table 9: Impact of SATD on class grime accumulation: summary of t-test results at class level

From Table 9 all of the results do not show significance either. In this instance, the partitions were flipped like such: 1) Only grime (i.e. no SATD) and, 2) grime and SATD. These results indicate that the presence of SATD has no effect on the presence of grime at class level.



(a) cg-na in HBase (b) cg-na in QuestDB (c) cg-na in JHotDraw



(d) cg-npm in HBase (e) cg-npm in QuestDB (f) cg-npm in JHotDraw

Figure 8: Distribution of grime metrics in grime only vs SATD-grime dataset at class level

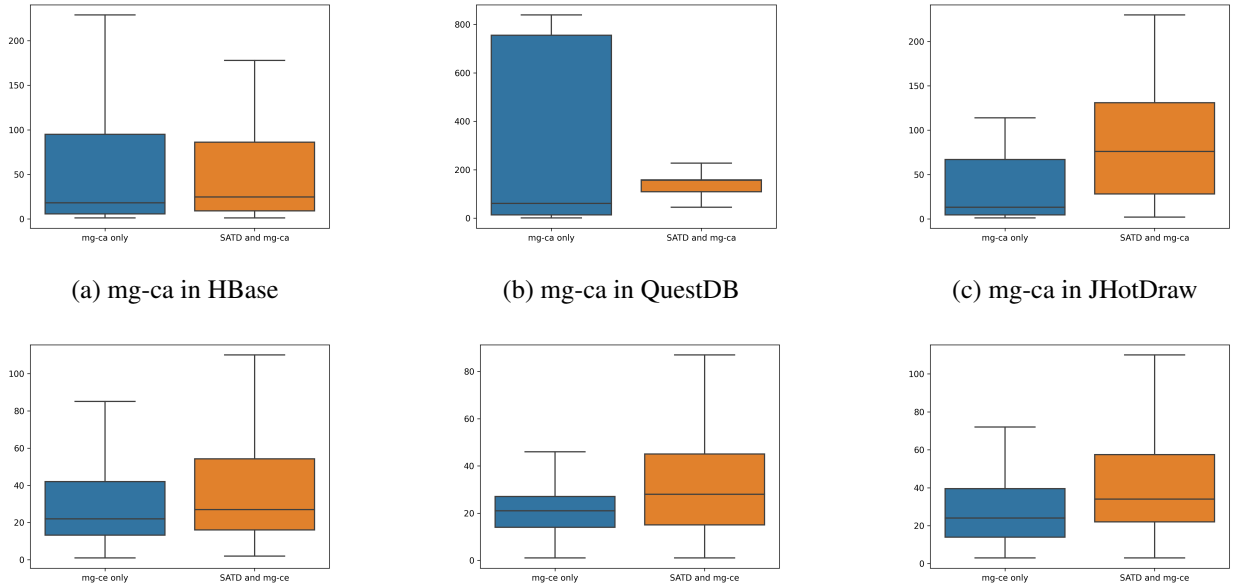
Instance Level

Dataset	Group	Mean	Std. Deviation	p-value	t-statistic	Dataset size
JhotDraw and QuestDB and HBase	mg-ca	274.49	339.18			
	mg-ca and SATD	125.86	163.85	< 0.001	13.71	577
	mg-ce	27.80	28.53			
	mg-ce and SATD	52.10	87.75	< 0.001	-6.53	
JhotDraw and QuestDB	mg-ca	292.33	343.05			
	mg-ca and SATD	122.82	102.55	< 0.001	16.16	401
	mg-ce	26.78	27.03			
	mg-ce and SATD	50.25	85.01	< 0.001	-5.44	
HBase	mg-ca	160.09	288.74			
	mg-ca and SATD	132.78	253.49	0.32	1.00	176
	mg-ce	34.20	35.95			
	mg-ce and SATD	56.28	93.73	0.003	-2.97	
QuestDB	mg-ca	304.01	346.95			
	mg-ca and SATD	135.96	110.35	< 0.001	14.59	286
	mg-ce	26.64	27.29			
	mg-ce and SATD	51.09	97.41	< 0.001	-4.19	
JhotDraw	mg-ca	58.19	80.41			
	mg-ca and SATD	90.16	70.31	0.007	-2.71	115
	mg-ce	29.29	21.49			
	mg-ce and SATD	48.17	40.62	< 0.001	-4.09	

Table 10: Impact of SATD on modular grime: summary of t-test results at instance level

Table 10 shows the T-Test results for SATD and pattern grime at instance level. While the results from class

level revealed no significant p-values this appears to be much different at instance level. Every p-value seems to point towards a significant difference in the means between modular grime and SATD apart from one. The only value that is not significant is when looking at HBase and considering afferent coupling with SATD, apart from that the rest all imply that SATD has a significant effect on the value of modular grime at instance level.



(a) mg-ca in HBase (b) mg-ca in QuestDB (c) mg-ca in JHotDraw

(d) mg-ce in HBase (e) mg-ce in QuestDB (f) mg-ce in JHotDraw

Figure 9: Distribution of grime metrics in grime only vs SATD-grime dataset at instance level

Pattern Level

Design Pattern	t-statistic (cg-na)	p-value (cg-na)	t-statistic (cg-npm)	p-value (cg-npm)	Dataset size
Factory Method	-0.98	0.83	-2.04	0.052	46
Singleton	-0.96	0.34	-1.49	0.15	168
(Object)Adapter	-1.13	0.26	-0.75	0.93	590
Decorator	-4.45	< 0.001	-4.01	< 0.001	65
State	-3.88	< 0.001	0.12	0.899	1106
Bridge	-2.20	0.03	-0.98	0.33	78
Template Method	-1.99	0.054	-3.34	< 0.001	95
Proxy	2.58	0.01	0.55	0.59	67

Table 11: T-test results for class grime based on SATD at pattern level

Conducting the T-Test analysis per pattern returned quite significant results. These tests were conducted at instance level, meaning that we calculated a sum of cg-na and sum of cg-npm per pattern instance. From Table 11 there are five patterns that indicate a significant result in relation to SATD. These include: 1) Decorator, 2) State, 3) Bridge, 4) Template Method, and 5) Proxy. The Decorator pattern was the only pattern that displayed significant results from both cg-na and cg-npm. The Bridge, Proxy and State patterns seem to show a correlation between SATD and the number of alien attributes in the pattern instance. Finally, the Template Method pattern's p-value is significant when considering the number of alien methods.

Design Pattern	t-statistic (mg-ce)	p-value (mg-ce)	t-statistic (mg-ca)	p-value (mg-ca)	Dataset size
Factory Method	-1.65	0.11	-2.38	0.02	46
Singleton	-1.90	0.07	-0.83	0.41	168
(Object)Adapter	-1.85	0.06	6.39	< 0.001	590
Decorator	-3.06	0.003	-2.21	0.03	65
State	-4.35	< 0.001	15.93	< 0.001	1106
Bridge	-2.02	0.049	2.45	0.01	78
Template Method	-2.38	0.02	-1.69	0.09	95
Proxy	-0.53	0.60	7.39	< 0.001	67

Table 12: T-test results: the influence of SATD on modular grime at pattern level

When it comes to modular grime, we notice that all patterns analyzed, apart from the Singleton pattern display significant results for the T-Test. From Table 12 the Decorator, State, and Bridge patterns return significant p-values in relation to both efferent and afferent coupling. The Factory Method, Object Adapter, and Proxy patterns are significant for afferent coupling, but not for efferent coupling. The Template Method pattern is the only pattern that has its only significant p-value in efferent coupling.

4.3.2 Correlation analysis

Project	grime type	p-value
QuestDB and Jhotdraw and HBase	cg-npm or cg-na	0.06
	cg-npm	0.04
	cg-na	0.04
QuestDB and JHotDraw	cg-npm or cg-na	< 0.001
	cg-npm	< 0.001
	cg-na	0.01
HBase	cg-npm or cg-na	< 0.001
	cg-npm	< 0.001
	cg-na	< 0.001
QuestDB	cg-npm or cg-na	0.008
	cg-npm	0.06
	cg-na	0.06
JHotDraw	cg-npm or cg-na	0.273
	cg-npm	0.463
	cg-na	0.364

Table 13: Chi-square results for class grime and SATD

The Chi-Squared results at class level returned varying results. The test was run in three ways for each project. It is important to know of the four categories that were used to run the tests. 1) HasSATD being all classes that contain at least one instance of SATD 2) HasGrime being all classes that contain either at least once instance of cg-npm or cg-na, 3) HasNA and, 4) HasNPM

1. By looking at HasSATD with HasGrime $\rightarrow$ Checking whether there is a correlation between SATD and both types of grime

2. By looking at HasSATD with HasNA $\rightarrow$ Checking whether there is a correlation between SATD and

alien attributes

3. By looking at HasSATD with HasNPM $\rightarrow$ Checking whether there is a correlation between SATD and alien methods

Looking at it project by project:

JHotDraw - All p-values returned are not significant, meaning that there is no correlation between SATD and grime. This result makes sense when you take into consideration that JHotDraw is viewed as a well-designed benchmark and contains low levels of both technical debt and pattern grime.

QuestDB - When looking at alien attributes and alien methods separately in isolation from one another the returned p-values are not significant, however, when comparing SATD to both of them then we discover that there seems to be a correlation between the pattern grime and SATD in QuestDB.

HBase - All p-values are < 0.001 implying that there is a correlation between the SATD instances and pattern grime in all cases.

JHotDraw and QuestDB - When the combined files of QuestDB and JHotDraw were being tested all of the results presented are significant.

All Three Projects - Testing all of the Java files that we were looking at returns somewhat surprising results. When looking at the grime metrics in isolation they returned significant results meaning there is a correlation between them and SATD. However, when both of them were considered at once then the p-value that was gained was not significant, implying that there is no correlation. However, as this p-value is at 0.06 we can say that there is a trend that is leaning towards significance and perhaps if given more data the results would be different.

5 Discussion and future work

This section delves into the findings of this research, which examines the relationship between design pattern grime and technical debt. In this section, we present a comprehensive analysis of the observed relationship, interpreting the results and exploring their implications for the field of software engineering.

5.1 Pattern grime and TD

To address the first research question (i.e., RQ1) and its associated sub-questions (i.e., RSQ1 and RSQ2), we examine the outcomes presented in Section 4.2 of this study. Additionally, we delve into the findings pertaining to various design patterns to address the second research question of this study (i.e., RQ2). Furthermore, it should be emphasized that the following analysis focuses specifically on technical debt detected via static code analysis when discussing the implications pertaining to the relationship between grime and TD.

TD and class grime

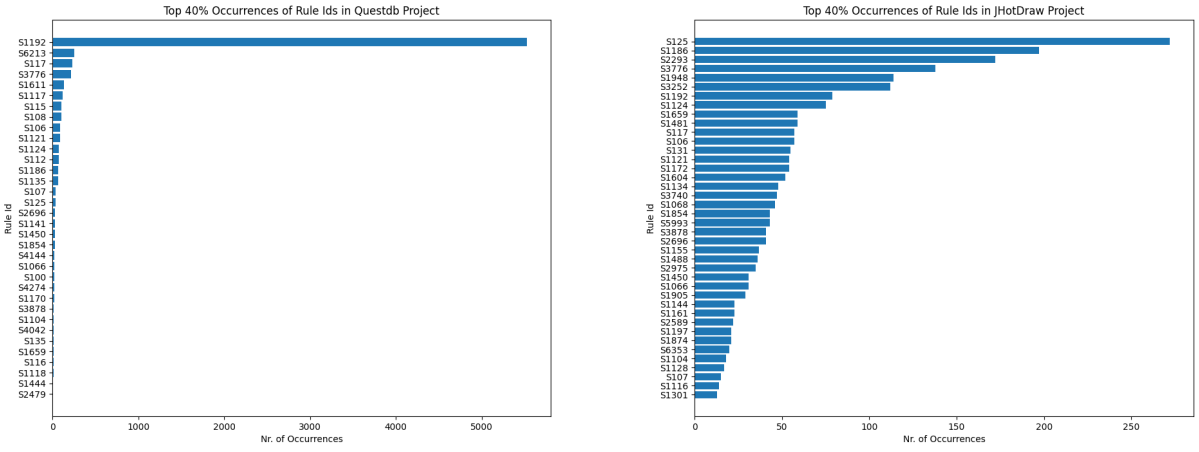
The results obtained from the t-test analysis, which examines the variation in TD accumulation in the presence of class grime, reveal that **the presence of class grime is associated with a decrease in the amount of TD accumulation**. This observation could potentially indicate that a meticulous design approach, which incorporates design patterns even if they contain grime, may contribute to the reduction of TD. This finding aligns with the notion that a well-structured design can potentially mitigate the accumulation of technical debt. Moreover, upon closer examination of the individual projects, it is evident that the impact of class grime on TD differs between the QuestDB and JHotDraw projects. In the case of the QuestDB project, the decrease in TD is more pronounced and statistically significant, indicating a stronger relationship between class grime and TD reduction. On the other hand, in the JHotDraw project, there was no significant difference in the mean TD accumulation between instances with and without class grime. Several factors may contribute to these divergent results. Firstly, it is important to consider the specific characteristics of each project, its complexity, and the overall development approach employed could influence the impact of class grime in reducing TD. For instance, in the case of QuestDB, developers might reduce the amount of TD in classes where design patterns and grime are introduced, as the classes become more complex, TD mitigation might be an effortless fix to do, specifically, Design and Code Debt receive more attention during refactoring [54]. However, in JHotDraw, the level of TD is maintained at a degree that does not drastically influence the quality of the project. This assumption can be strengthened by the findings of Tan et al. [54], which indicate that the *"fewer developers maintaining a file and the more changes made per file, the higher the chance of an issue being self-fixed"* which is the case of JHotDraw where only three developers have contributed to the repository over of a period of 23 years, while in QuestDB, 114 developers have contributed to the system over nine years. Therefore, the level of awareness and adherence to best practices within the development teams may play a crucial role in managing technical debt and enforcing clean design principles. An additional factor contributing to higher levels of TD in the absence of grime is the presence of test classes that incurred a significant amount of TD. Notably, classes such as `SampleByTest.java` with 379 TD items and `CastTest.java` with 129 TD items were particularly affected. These classes are predominantly utilized for unit testing purposes and do not participate in any instances of design patterns. Consequently, the absence of grime in these test classes and the lack of design pattern integration may contribute to the higher TD levels observed in Table 3.

Contrary to the observed relationship between class grime and TD, the analysis of the association of TD with the distribution of class grime yielded interesting results. Specifically, we discovered a **significant increase in class grime, particularly in the number of attributes that deviate from the expected pattern structure, in the presence of TD**. One possible explanation for the observed increase in alien attributes could be attributed to design flaws within the affected classes. Issues such as a lack of modularity, poor encapsulation, or insufficient separation of concerns are common flaws when deviating from the intended pattern structure.

Rule ID	Definition	Classification
S1192	String literals should not be duplicated	Design Debt
S6213	Restricted Identifiers should not be used as Identifiers	Design Debt
S117	Local variable and method parameter names should comply with a naming convention	Code Debt
S3776	Cognitive Complexity of methods should not be too high	Design Debt
S1611	Parentheses should be removed from a single lambda input parameter when its type is inferred	Code Debt
S1117	Local variables should not shadow class fields	Defect Debt
S115	Constant names should comply with a naming convention	Code Debt
S1948	Fields in a serializable class should either be transient or serializable	Defect Debt
S108	Nested blocks of code should not be left empty	Code Debt
S106	Standard outputs should not be used directly to log anything	Code Debt
S1121	Assignments should not be made from within sub-expressions	Code Debt
S1124	Modifiers should be declared in the correct order	Code Debt
S112	Generic exceptions should never be thrown	Defect Debt
S1186	Methods should not be empty	Defect Debt
S1135	Track uses of TODO tags	Documentation Debt
S107	Methods should not have too many parameters	Design Debt
S125	Sections of code should not be commented out	Code Debt

Table 14: Top 16 broken SonarQube rules in QuestDB and JhotDraw

For instance, the `io.questdb.cairo.TableWrite.java` class in QuestDB, which recorded the highest number of alien attributes (i.e., 3525), exhibited critical rule violations, including cognitive complexity in methods and code duplication, which contributed to the accumulation of technical debt. Additionally, in QuestDB, the most prominent broken rules, depicted in Figure 10a, are related to handling duplicated string literals (S1192), the inappropriate naming convention of attributes and methods (S6213, S117), and high cognitive complexity (S3775) and while in JHotDraw, Figure 10b, rules concerning empty methods (S1186) and commented-out code sections (S125), detailed in Table 14, are frequent. These observations align with those by Feitosa et al. [4], which state that classes that participate in grime instances are linked with a decrease in quality attributes such as correctness, performance, and security.



(a) Distribution of rules QuestDB

(b) Distribution of rules in JHotDraw

Figure 10: Top 40% most frequent SonarQube rules that introduced TD items.

Furthermore, the absence of a significant association between the presence of TD and the number of alien public methods in both projects might indicate that, unlike the effect on alien attributes, TD might not have a direct impact or association with alien public methods. TD is a low-level example that affects the quality of a system and has a higher chance of introducing complex methods (e.g., cyclomatic complexity increases the size of a method) rather than leading to an increase in the number of methods.

The chi-square analysis reveals a significant association between class grime and technical debt, which can be attributed to the shared areas of concern between the two. In particular, our findings highlight that the most frequently violated SonarQube rules pertain to attributes, naming conventions, and cognitive complexity. These rules are directly related to the presence of class grime and introduce technical debt issues. The identification of these common areas provides valuable insights for developers to proactively address class grime as a means of mitigating technical debt. By adopting cleaner coding practices, such as adhering to design principles and refactoring code to reduce grime, developers can effectively minimize the accumulation of technical debt in their software projects. It is crucial to emphasize that correlation does not imply causation. The observed association between TD and grime does not necessarily indicate a causal relationship where it leads to lower TD values or TD impacts directly the accumulation of grime. Other factors, such as quality attributes or the developer's coding practices and experience may contribute to the observed relationship.

TD and modular grime

Additionally, we looked at how pattern deviations, measured with modular grime metrics, vary in the presence or absence of TD. Thus, we noticed that **afferent coupling has a lower concentration in the presence of TD**. Notably, TD may discourage classes from establishing dependencies on pattern instances that exhibit grime, resulting in a lower afferent coupling. This could be attributed to developers prioritizing the mitigation of TD even in pattern instances with grime. As a result, pattern instances without TD are perceived as more reliable and more classes in the system may establish dependencies on these "improved" pattern instances. On the other hand, the results show that **TD is associated with higher efferent coupling**. This finding suggests that when a pattern instance contains technical debt, it tends to have a higher number of dependencies on other classes within the system. A possible explanation for this observation, if we assume causation, is that the presence of grime in a pattern instance may contribute to a more complex or convoluted design and introduce higher TD items. As a result, the pattern instance may need to rely on a larger number of classes to fulfill its functionality, leading to higher efferent coupling. Moreover, TD could also introduce additional dependencies or workarounds that require interactions with other classes. For example, an instance of the Factory Method pattern contains a high number of outgoing dependencies (i.e, 166 in total) and high accumulations of TD (i.e., 28 total broken rules), however, the class that is responsible for such complexity is the `io.questdb.cairo.TableWriter.java`[19]. Moreover, it is important to consider the role of a class in a system. For instance, the aforementioned class encapsulates the functionality related to managing the low-level details of writing data into the underlying storage engine, specifically handling the storage and organization of data in the database tables. Thus, such complex classes (i.e., with high efferent coupling), might justify the observed results.

Furthermore, the chi-square test revealed no significant correlation between efferent and afferent coupling and TD. Despite the contradictory findings, it is evident that coupling is a complex concept influenced by multiple factors. Therefore, we conclude that further research is needed to gain a comprehensive understanding of this relationship.

Analysis at pattern level

When looking at the t-test results of each design pattern instance, our observations indicate that design patterns with higher frequency values, (e.g., State, (object)Adapter, Decorator) significantly influence the overall statistical outcome. These design patterns, due to their larger representation in the dataset, carry more weight in

[19] https://javadoc.questdb.io/io.questdb/io/questdb/cairo/tablewriter

shaping the conclusions drawn from the analysis. Additionally, these design patterns, characterized by polymorphic calls, inherently introduce greater complexity into the codebase. The presence of polymorphism often leads to increased code size and more coupled interactions between components, making it more challenging to minimize grime and TD accumulations. This finding has been previously discussed by Feitosa et al. [4] which further strengthens the understanding that certain design patterns, particularly those involving polymorphism, can be prone to grime and TD and can shape the overall outcome of the relationship between these two concepts.

5.2 Pattern grime and SATD

In this study, we performed an analysis of self-admitted technical debt and pattern grime. Specifically, we were interested in discovering whether there is a relationship between the two. We chose to analyze three projects, namely HBase, QuestDB, and JHotDraw, which provided us with a diverse range of software repositories. The initial analysis of SATD and pattern grime at the class level using the t-test demonstrated no significant relationship between the presence of grime and SATD. However, when conducted at the instance level we noticed a shift in the results with a majority of the results indicating a significant relationship between the presence of SATD and modular grime, the only exception being afferent coupling with SATD in HBase.

It could be argued that as the larger pattern instances grow and develop more dependencies, they become complex. This would explain why developers are more likely to leave comments behind in the implementation as it grows to more than just one class.

As for the chi-squared results, we observed variances in each project. As JHotDraw is known as a well-developed benchmark project it displays low levels of both technical debt and pattern grime it displayed no correlation between SATD and grime. However, when looking at HBase and QuestDB the results were significant indicating a correlation in those projects. Additionally, as the projects were combined there also appears to be a trend leaning towards a correlation between pattern grime at class level and SATD.

Towards answering our research questions:

RQ1 **What is the relationship between the presence of grime and (self-admitted) technical debt?**

RSQ1 **Is the presence of grime associated with self-admitted technical debt and vice versa?**

From our results outlined above, it is possible to draw the conclusion that pattern grime has an association with self-admitted technical debt and vice versa. The T-Test and Chi-Squared results all lean towards an association between the two concepts.

RSQ2 **Which grime metrics demonstrate a more pronounced association with the presence of self-admitted technical debt?**

From the results of the t-test it appears that modular grime at instance level has a more pronounced association with SATD than grime at class level. However, the results of the chi-squared test also present values that are significant at class level and should not be ignored. Overall, it appears that all aspects of grime have an equal association with self-admitted technical debt.

RQ2 **What design patterns indicate a higher association between grime and self-admitted technical debt?**

From our results in the previous section it is clear that design patterns differ when it comes to their association with grime and SATD. This is true for when we are considering both class grime and modular grime at instance level. More patterns seemed to be affected by modular grime than class grime, which, once again could be attributed to the growth in complexity of the pattern instances as the classes grow in dependencies. Seven out of the eight design patterns we investigated returned significant values for at

least one grime metric.

The Singleton pattern was the only one that did not indicate a higher association between grime and SATD, it could be argued that the implementation of the Singleton pattern is widespread and commonplace these days, meaning that developers most likely do not struggle to implement the standard implementation and therefore do not introduce grime into the pattern instance. As the implementation is quite simple and well-documented, developers also most likely do not introduce high levels of self-admitted technical debt as implementing the pattern instance can be quite simple.

The Decorator pattern was the only pattern that displayed significant p-values for all grime metrics that we measured, indicating that it may have a higher association between grime and SATD.

5.3 Future Work

The current study on the relationship between design pattern grime and technical debt opens up several areas for future research. Firstly, expanding the analysis to include a larger number of projects, beyond those examined in this study, would provide a more comprehensive understanding of the association between grime and technical debt. Moreover, our current findings cannot be generalized to projects written in other programming languages. Thus, extending the analysis to projects written in different programming languages (e.g., Python, JavaScript) would provide insights into the language-specific aspects of grime and its relationship with technical debt. Lastly, to gain a broader perspective, it would be beneficial to have a higher representation of other design patterns, allowing for a more nuanced examination of how different patterns influence the study's outcomes.

Moreover, our study also offers insights into the associations between pattern grime and SATD at class and instance levels. As we have seen from our own results these associations can vary from project to project. Thus, it would be pivotal to examine this relationship in a larger variety of repositories to gain even greater insights into how exactly pattern grime and SATD are related. On top of this, we only conducted our analysis on SATD in source code, however, there is a large amount of data in commit messages, pull requests, and issue trackers, that is being omitted in this study. We believe that conducting an analysis on SATD and pattern grime from these sources could provide us with substantial information on how pattern grime and SATD interact in a project's life cycle. Lastly, we were only concerned with the binary relationship of SATD and non-SATD, running an analysis on the different classifications of SATD in relation to pattern grime would allow us to determine whether a certain type of SATD has a stronger or weaker association with pattern grime.

Furthermore, future research could consider conducting empirical studies that take into account factors such as project size, developer experience, and quality attributes, to explore how these variables influence the association between grime and technical debt. Additionally, longitudinal studies could track the evolution of grime and its impact on technical debt over time (e.g., across different versions of the system), providing insights into the long-term consequences and potential mitigation strategies. Lastly, conducting industry case studies that delve deeper into specific projects or companies would allow for more detailed and practical factors such as team dynamics, development practices, and experience level that might impact the relationship between grime and technical debt. Overall, these future research directions can contribute to a more comprehensive understanding of the dynamics between design pattern grime and technical debt.

6 Threats to validity

This section evaluates potential threats that could impact the accuracy and reliability of the findings in this study. Namely, we discuss *construct validity* which focuses on ensuring the accuracy of the measurement techniques utilized to capture the intended theoretical constructs. *Reliability* pertains to the consistency of the study's setup and data analysis phase and ensures the reproducibility of the study. *External validity* concerns the generalizability of the findings to other contexts or populations.

Concerning *construct validity*, we identified several threats that pertain to the methods in which we measured both grime and TD in our study. The first threat to construct validity is the accuracy of TD measurement achieved via static code analysis. Based on our knowledge, SonarQube is the only open-source tool that is widely used in both the industry and software engineering communities for identifying and measuring TD. Nevertheless, SonarQube might introduce false positives flagged in the codebase. Moreover, it may also miss certain instances of technical debt, resulting in false negatives. This, in turn, can be addressed by manually analyzing the codebase and verifying the relevancy of the flagged code smells. In this study, we did not focus on manually validating the output of the SonarQube analysis, however, all the projects underwent analysis using the default set of rules provided by SonarQube to minimize subjectivity.

Another threat to the construct validity is the tool used for the classification of SATD using natural language processing. From the paper by Li et al. [34] the tool achieved an F1-score of 0.611. To determine the validity of our results, we used *stratified random sampling* [55] on the comments extracted from pattern instances. From 119 pattern instances approximately 1500 comments were extracted and classified, and with a 50% population proportion one of the researchers verified the output of 750 comment classifications. The other researcher was given a 20% population proportion of the first researcher's validation for a further 150 comment classifications. This method ensures that we avoid bias as much as we can while validating the output of the tool. When considering the entire set of 750 comments from the initial sampling, there were 24 incorrect classifications. This results in an accuracy of 96.80%. When the tool output classified a comment as SATD there were 9 correct classifications and 6 false positives. Considering this, when the tool classified something as SATD it was accurate 60% of the time.

Yet another possible concern that could impact the construct validity regards the design pattern and grime detection tools which can be limited by false positives and negatives that can affect the accuracy of our findings. However, it is worth noting that these tools have demonstrated satisfactory performance as evidenced by previous research where they have been successfully validated [4]. Nevertheless, to mitigate this threat, we performed *stratified random sampling* on the QuestDB dataset and verified the output of 120 classes (i.e., 20% population proportion out of 344 classes that participate in unique grime instances). Furthermore, in order to ensure the accuracy of the validation process, a subsample of the dataset (i.e., 57 classes in total) was revalidated to increase the accuracy of the results. The outcome of our validation shows that out of the total 120 classes analyzed, the tool correctly identified the presence of grime in 115 classes (i.e., 4 false positives and 1 false negative), resulting in a validation accuracy of 95.8%. Thus, we can conclude that the *spoon-pttgrime* tool provides decent capabilities to accurately detect grime in software projects.

To address *reliability threats* and minimize potential biases in data collection, we manually checked the datasets at different stages in our research to ensure data consistency. For instance, we checked a subset of the data containing merged TD and grime accumulations per class to guarantee that the merging step was performed accordingly. Moreover, we employed Python scripts to automate some processes such as data parsing (i.e., extracting grime metrics from XML to CSV) and project configuration in the case of SonarQube analysis.

Concerning *external validity*, the main threat is the limited scope of our study, as only two projects were explored. This restricted dataset may not fully represent the diversity of software projects, thus limiting the

generalizability of our findings. Another aspect that impacts external validity is the focus on projects developed in Java. While Java is a widely used programming language, it may not capture the characteristics and nuances of projects developed in other languages. To address this limitation, additional analyses could be conducted on projects developed in different languages to provide a more comprehensive understanding of the relationship between grime and technical debt. Furthermore, we acknowledge that only a limited number of patterns were investigated. This narrow focus does not allow us to capture the full spectrum of design pattern grime and its relationship to technical debt. Including a wider range or at least a balanced representation of each design pattern in the analysis could yield more comprehensive insights. Lastly, to enhance the observations and accuracy of the study, future studies can analyze grime based on subtypes of grime [56]. This approach can provide a more nuanced understanding of the relationship between specific subtypes of grime and technical debt.

6.1 Reproducibility

To ensure the reproducibility of our analysis, we have taken several steps to provide transparency and accessibility to our research process and data. Firstly, we created a public GitHub repository [20] with instructions and scripts to replicate some aspects of our data collection and manipulation. For instance, we have developed a script, written and ran on a machine running Ubuntu 22.04 LTS, that automates the collection of TD based on a predefined list of repositories. This script ensures consistency in the data collection process and eliminates manual errors or biases. Additionally, we have created Python scripts for statistical analysis, which encompass the necessary statistical methods employed in our study. These scripts provide a clear and standardized methodology for conducting statistical tests, ensuring that the results can be replicated and validated by other researchers. Furthermore, we made our dataset readily available, including both the TD data and the grime data that was collected during this study. To facilitate the integration of the datasets, we have developed scripts that merge the TD and grime datasets. This ensures that the combined dataset is accurately consolidated and can be used for further analysis and interpretation.

[20] https://github.com/anaterna/BscThesis

7 Conclusion

In this research paper, we have conducted an extensive investigation into the relationship between design pattern grime and technical debt in Java projects. To accomplish this, we have considered four metrics related to class and modular grime, which have been employed to identify accumulations of grime. Furthermore, we have utilized two approaches to measure technical debt: static code analysis based on SonarQube rules and the analysis of self-admitted technical debt instances extracted from code comments.

Our findings reveal that the presence of class grime, specifically alien attributes, is associated with a decrease in the accumulation of technical debt. In contrast, the presence of technical debt leads to an increase in the number of alien attributes, while the number of alien public methods remains unaffected. We have also observed a decrease in afferent coupling and an increase in efferent coupling in the presence of technical debt which suggests that technical debt is linked to increased dependencies on external classes, while the dependencies on the affected class may decrease. Further investigation and analysis are required to fully understand the implications of coupling and its potential effects on software quality. Additionally, the analysis at the pattern level highlights that frequent design patterns, such as State, (Object) Adapter, and Decorator, significantly influence the overall outcome based on the aforementioned observations.

When it comes to self-admitted technical debt our analysis revealed that pattern grime appears to have the largest influence on SATD at instance level, specifically reporting a higher association with modular grime. This can be attributed to the growth in complexity as more dependencies are introduced into the pattern instance. The Chi-Squared results tell us that there is also an association between pattern grime and SATD at class level. However, this also varied between the projects so it would be interesting to see more research conducted on a diverse group of projects to see how that affects the analysis. Seven out of the eight patterns that were looked at displayed a relationship between pattern grime and SATD, the Singleton pattern being the only one that did not seem to have this association.

To ensure the validity of our study, we have validated the output of the pattern grime and self-admitted technical debt instances, yielding results indicating a satisfactory accuracy rate of over 90% for the utilized tools.
Building upon our findings, we propose several areas for future research. Firstly, expanding the scope of analysis to include a large number of projects also written in different programming languages would allow for a more comprehensive understanding of the relationship between grime and technical debt. Additionally, exploring a more balanced dataset of design patterns and investigating different subtypes of grime would provide accurate insights into how each design pattern contributes to the relationship between these two phenomena. Furthermore, incorporating alternative methods of detecting self-admitted technical debt, such as leveraging issue trackers, commit analysis, or pull requests, could enrich our understanding of the phenomenon. Lastly, conducting longitudinal studies, industrial case studies, and empirical studies would contribute to a more comprehensive exploration of the topic.

Bibliography

[1] W. Cunningham, "The wycash portfolio management system," *SIGPLAN OOPS Mess*, vol. 2, pp. 29–30, December 1992.

[2] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," *Dagstuhl Reports 6*, vol. 4, p. 120–121, 2016.

[3] N. Rios, R. Spínola, M. Mendonça, and C. Seaman, "The most common causes and effects of technical debt: first results from a global family of industrial surveys," *in: 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement – ESEM '18*, p. 1–10, 2018.

[4] D. Feitosa, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, "Correlating pattern grime and quality attributes," *IEEE Access*, vol. 6, pp. 23065–23078, 2018.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of reusable object-oriented software (addison-wesley professional computing series)," USA: Addison-Wesley, 1995.

[6] C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Softw. Quality J.*, vol. 21, no. 2, p. 289–323, Jun. 2013.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Elements of reusable object-oriented software (addison-wesley professional computing series)," pp. 233–242, USA: Addison-Wesley, 1995.

[8] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pp. 449–451, 2007.

[9] C. Izurieta and J. M. Bieman, "Testing consequences of grime buildup in object oriented design patterns," in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 171–179, 2008.

[10] D. Feitosa, P. Avgeriou, A. Ampatzoglou, and E. Y. Nakagawa, "The evolution of design pattern grime: An industrial case study," in *Product-Focused Software Process Improvement*, (Cham), pp. 165–181, Springer International Publishing, 2017.

[11] F. K. J. Radatz, A. Geraci, "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 10–15, 1990.

[12] F. A. Fontana, A. Caracciolo, and M. Zanoni, "Dpb: A benchmark for design pattern detection tools," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 235–244, 2012.

[13] B. Bafandeh Mayvan, A. Rasoolzadegan, and A. Ebrahimi, "A new benchmark for evaluating pattern mining methods based on the automatic generation of testbeds," *Information and Software Technology*, vol. 109, pp. 60–79, 2019.

[14] D. Shilintsev and G. Dlamini, "A study: Design patterns detection approaches and impact on software quality," in *Frontiers in Software Engineering* (G. Succi, P. Ciancarini, and A. Kruglov, eds.), (Cham), pp. 84–96, Springer International Publishing, 2021.

[15] N. Nazar, A. Aleti, and Y. Zheng, "Feature-based software design pattern detection," *Journal of Systems and Software*, vol. 185, p. 111179, 2022.

[16] R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero, "Geml: A grammar-based evolutionary machine learning approach for design-pattern detection," *Journal of Systems and Software*, vol. 175, p. 110919, 2021.

[17] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[18] J. Dong, Y. Sun, and Y. Zhao, "Design pattern detection by template matching," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, (New York, NY, USA), p. 765–769, Association for Computing Machinery, 2008.

[19] CAST, "Cast worldwide application software quality study: Summary of key findings," *CAST Research Labs*, 2010.

[20] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in *Proceedings of the 2nd Workshop on Managing Technical Debt*, (New York, NY, USA), p. 39–42, Association for Computing Machinery, 2011.

[21] V. Lenarduzzi, T. Besker, D. Taibi, A. Martini, and F. Arcelli Fontana, "A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools," *Journal of Systems and Software*, vol. 171, p. 110827, 2021.

[22] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015.

[23] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. Arcelli Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, A. Moschou, I. Pigazzini, N. Saarimaki, D. D. Sas, S. S. de Toledo, and A. A. Tsintzira, "An overview and comparison of technical debt measurement tools," *IEEE Software*, vol. 38, no. 3, pp. 61–71, 2021.

[24] D. Pina, A. Goldman, and C. Seaman, "Sonarlizer xplorer: a tool to mine github projects and identify technical debt items using sonarqube," in *2022 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 71–75, 2022.

[25] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 91–100, IEEE, 2014.

[26] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International conference on software maintenance*, pp. 516–519, IEEE, 2013.

[27] E. d. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pp. 9–15, 2015.

[28] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *2008 12th European conference on software maintenance and reengineering*, pp. 329–331, IEEE, 2008.

[29] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th international conference on mining software repositories*, pp. 315–326, 2016.

[30] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? an in-depth perspective," in *Proceedings of the 15th international conference on mining software repositories*, pp. 526–536, 2018.

[31] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Software Engineering*, vol. 23, pp. 418–451, 2018.

[32] E. d. S. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.

[33] M. A. de Freitas Farias, M. G. de Mendonça Neto, M. Kalinowski, and R. O. Spínola, "Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary," *Information and Software Technology*, vol. 121, p. 106270, 2020.

[34] Y. Li, M. Soliman, and P. Avgeriou, "Automatic identification of self-admitted technical debt from four different sources," *arXiv preprint arXiv:2202.02387*, 2022.

[35] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, 2017.

[36] A. F. Sabbah and A. A. Hanani, "Self-admitted technical debt classification using natural language processing word embeddings," *International Journal of Electrical and Computer Engineering*, vol. 13, no. 2, p. 2142, 2023.

[37] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[39] I. Santos, N. Nedjah, and L. de Macedo Mourelle, "Sentiment analysis using convolutional neural network with fasttext embeddings," in *2017 IEEE Latin American conference on computational intelligence (LA-CCI)*, pp. 1–5, IEEE, 2017.

[40] M. Pal, "Random forest classifier for remote sensing classification," *International journal of remote sensing*, vol. 26, no. 1, pp. 217–222, 2005.

[41] S. Suthaharan and S. Suthaharan, "Support vector machine," *Machine learning models and algorithms for big data classification: thinking with examples for effective learning*, pp. 207–235, 2016.

[42] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*, pp. 1–6, Ieee, 2017.

[43] Y. Li, M. Soliman, P. Avgeriou, and L. Somers, "Self-admitted technical debt in the embedded systems industry: An exploratory case study," *IEEE Transactions on Software Engineering*, 2022.

[44] M. R. Dale and C. Izurieta, "Impacts of design pattern decay on system quality," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, (New York, NY, USA), Association for Computing Machinery, 2014.

[45] I. D. Griffith, "Design pattern decay – a study of design pattern grime and its impact on quality and technical debt," 2021.

[46] I. Griffith, "Design pattern decay: An extended taxonomy and empirical study of grime and its impact on design pattern evolution," 2013.

[47] R. Alfayez, R. Winn, W. Alwehaibi, E. Venson, and B. Boehm, "How sonarqube-identified technical debt is prioritized: An exploratory case study," *Information and Software Technology*, vol. 156, 2023.

[48] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*. USA: Manning Publications Co., 1st ed., 2013.

[49] Y. Kim, "Convolutional neural networks for sentence classification," 2014.

[50] Y. Li, M. Soliman, and P. Avgeriou, "Identifying self-admitted technical debt in issue tracking systems using machine learning," *Empirical Software Engineering*, vol. 27, no. 6, p. 131, 2022.

[51] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, jul 2019.

[52] K. H. Yim, F. S. Nahm, K. A. Han, and S. Y. Park, "Analysis of statistical methods and errors in the articles published in the korean journal of pain," *The Korean journal of Pain, 23(1)*, pp. 35–41, 2010.

[53] A. Field in *Discovering statistics using IBM SPSS statistics (4th ed.)*, pp. 144–148, SAGE Publications, 2013.

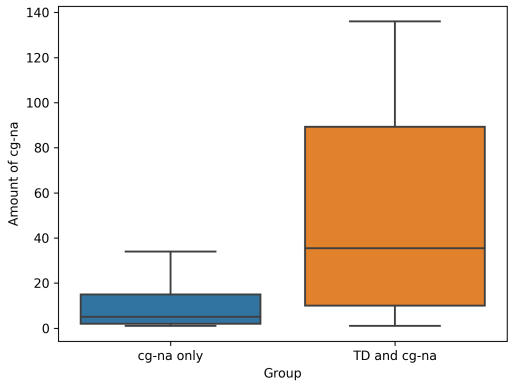
[54] J. Tan, D. Feitosa, and P. Avgeriou, "Does it matter who pays back technical debt? an empirical study of self-fixed td," *Information and Software Technology*, vol. 143, p. 106738, 2022.

[55] F. Shull, J. Singer, and D. I. Sjøberg in *Guide to Advanced Empirical Software Engineering*, p. 85, Springer, 2007.

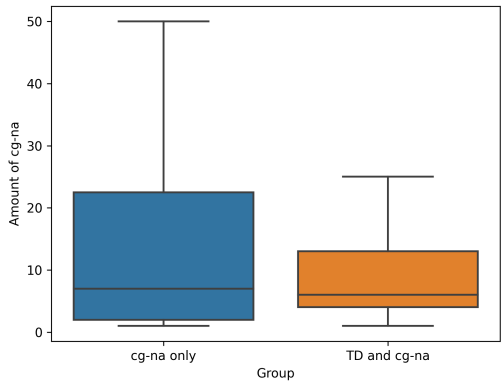
[56] T. Schanz and C. Izurieta, "Object oriented design pattern decay: A taxonomy," (New York, NY, USA), Association for Computing Machinery, 2010.

Appendix - Supplementary Material

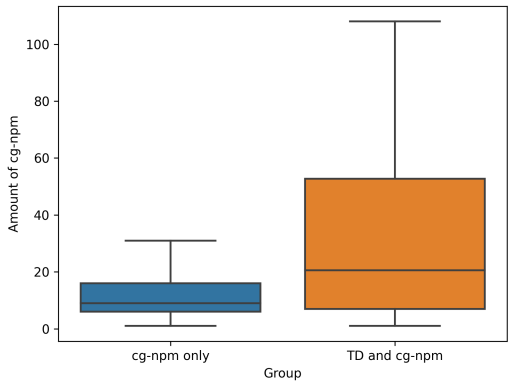
A Additional visualizations



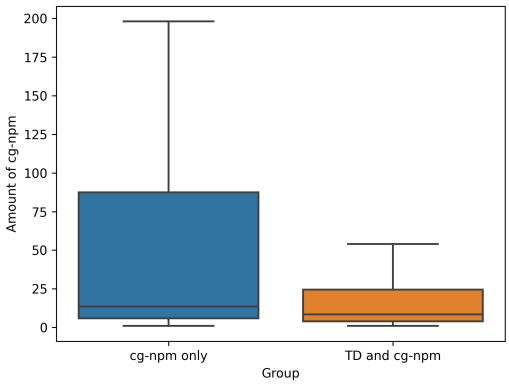
(a) cg-na in QuestDB



(b) cg-na in JHotDraw



(c) cg-npm in QuestDB



(d) cg-npm in JHotDraw

Figure 11: Distribution of class grime in grime-only vs TD-grime dataset