# THE IDENTIFICATION, CATEGORIZATION, AND EVALUATION OF MODEL-BASED BEHAVIORAL DECAY IN DESIGN PATTERNS

by

Derek Kristaps Reimanis

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

August 2019

# ACKNOWLEDGEMENTS

The finalization of this PhD represents the accomplishment of a major goal in my life. When I started, I was headstrong and overly confident to a fault. Yet as the years progressed, I grew humbler as I began to see the beauty in knowledge – I firmly believe that as one learns more about the universe, one begins to realize how much they truly do not know. To this end, a PhD is a project that no one can complete alone. I would like to acknowledge and thank the people and entities that were instrumental along the way.

I would like to acknowledge my family for instilling me with a strong work ethic, as well as the advice they provided at critical points in the process. Thank you.

This dissertation would be nothing with the help, advice, and mentorship of my advisor, Clem. I greatly appreciate your patience with me as I grew through this chapter in my life. Thank you.

I would like to thank Zoot Enterprises for their support of me through the majority of my PhD. In addition to financial support, Zoot provided me with a practical perspective that increased the value of my research. Thank you.

# DEDICATIONS

I dedicate this dissertation to my lovely wife, Rachael. She has been a true wonder and inspiration through the process of completing this PhD. She has never complained at the long hours I spent in the lab even though it meant I sacrificed time at home. She has always been willing to listen and provide valuable input to the problems encountered along the process. Thank you, Rachael.

TABLE OF CONTENTS

TABLE OF CONTENTS CONTINUED

TABLE OF CONTENTS CONTINUED

TABLE OF CONTENTS CONTINUED

# LIST OF TABLES

LIST OF TABLES CONTINUED

Table Page

# LIST OF FIGURES

LIST OF FIGURES CONTINUED

Figure Page

# ABSTRACT

Software quality assurance (QA) techniques seek to provide software developers and managers with the methods and tools necessary to monitor their software product to encourage fast, on-time, and bug-free releases for their clients. Ideally, QA methods and tools provide significant value and highly-specialized results to product stakeholders, while being fully incorporated into an organization's process and with actionable and easy-to-interpret outcomes. However, modern QA techniques fall short on these goals because they only feature structural analysis techniques, which do not fully illuminate all intricacies of a software product. Additionally, many modern QA methods are not capable of capturing domain-specific concerns, which suggests their results are not fulfilling their potential.

To assist in the remediation of these issues, we have performed a comprehensive study to explore an unexplored phenomenon in the field of QA, namely model-based behavioral analysis. In this sense, behavioral analysis refers to the mechanisms that occur in a software product as the product is executing its code, at system run-time. We approach this problem from a model-based perspective because models are not tied to program-specific behaviors, so findings are more generalizable. Our procedure follows an intuitive process, involving first the identification of model-based behavioral issues, then the classification and categorization of these behavioral issues into a taxonomy, and finally the evaluation of them in terms of their effect on software quality.

Our results include a taxonomy that captures and provides classifications for known model-based behavioral issues. We identified relationships between behavioral issues and existing structural issues to illustrate that the inclusion of behavioral analysis provides a new perspective into the inner mechanisms of software systems. We extended an existing state-of-the-art operational software quality measurement technique to incorporate these newfound behavioral issues. Finally, we used this quality extension to evaluate the effects of behavioral issues on system quality, and found that software quality has a strong inverse relationship with behavioral issues.

# CHAPTER ONE

# INTRODUCTION

## 1.0 Foreword

This chapter introduces several foundational concepts and provides motivation for the greater work in this dissertation. In section 1.1 we state the problem statement, followed by the general process employed to complete the body of work in section 1.2. We conclude Chapter 1 with an overview of each following chapter in section 1.3.

## 1.1 Problem Statement

Software quality assurance techniques provide software developers and managers with the methods and tools necessary to monitor their software product(s) to encourage fast, on-time, and bug-free releases for their clients. Ideal circumstances hold that the methods and tools of software quality assurance provide significant value and highly-specialized results to product stakeholders. Additionally, and with recent pushes towards process automation, ideally these methods and tools would be fully incorporated into an organization's continuous integration and continuous delivery process and with actionable and easy-to-interpret results. However, modern approaches fall short on these goals, and while many QA techniques exist that provide results to stakeholders, many times these results do not provide their stated value or are simply ignored. We claim this is due to two primary influences. First, current software QA approaches do not fully reveal all aspects of a software product in part because of their focus on static, or

structural analysis. By itself, static analysis is not an impairment, yet it fails to provide sufficient insight into a product's inner-workings to allow for a thorough analysis. Second, many QA techniques provide general packaged solutions, which fail to capture domain-specific concerns. Different software stakeholders have different expectations of quality, both from an end-user perspective and from an internal code quality perspective. Modern packaged solutions do not provide maximum value because they either do not allow for the ability to configure the solution to cater to specific needs, or the customizations they provide are difficult to implement because of the arbitrary process in which such a solution is calibrated. This logic forms the basis for our research, and a formal problem statement is presented:

> *Under ideal circumstances, software quality assurance efforts provide significant, highly-specialized, and immediate value to software product stakeholders. However, many modern approaches fall short of their goals, due to lack of models that fully capture the entities of a system, as well as models that fail to capture domain specific concerns.*

To assist in the remediation of these issues, we have committed to the exploration of model-based behavioral analysis techniques, which consider the mechanisms that occur as a product is executing its code at runtime from a modeling perspective. Specifically, we focus on design pattern evolution because of the known quality properties of design patterns, yet our methods are generalizable in context where product behavior from a modeling perspective is explicitly defined. The exploration of behavioral

analysis techniques complements existing structural analysis techniques, expanding upon the capabilities of state-of-the-art QA techniques. Furthermore, the manner in which we developed and evaluated these newfound capabilities, via extending an existing quality model that is highly-customizable yet easy-to-use and interpret, encourages a straightforward and non-arbitrary customization that fits all domains.

## 1.2 Solution Design

'Design Science' is a term that refers to the design and investigation of artifacts in a context to solve a problem [84]. Specifically, design science is concerned with solving problems by understanding the interactions between artifacts and contexts; artifacts and contexts exist as such, but to understand them fully researchers must understand the nature of the relationship between them, such as how the design of an artifact improves a context or how the context instigates the development of new artifacts. To this end, design science contains two kinds of research problems, *design problems* and *knowledge questions* [84]. Design problems are concerned with the design of a change in the real world, many times via an artifact, to solve a problem. Alternatively, knowledge questions are concerned with questioning the world as it is, many times via a propositional statement, such as 'Is *x* good enough?'. The interactions between design problems and knowledge questions forms a cycle; new design problems are created to solve knowledge questions, and knowledge questions provide inspiration for new design problems. Depending on the direction of the interaction, the cycle of knowledge acquisition is referred to as a *design cycle* or an *empirical cycle*. A design cycle captures the design

problem to knowledge question direction, and an empirical cycle captures to knowledge question to design problem direction. This greater process is generalized into a framework, which is presented in figure 1.1 [84].

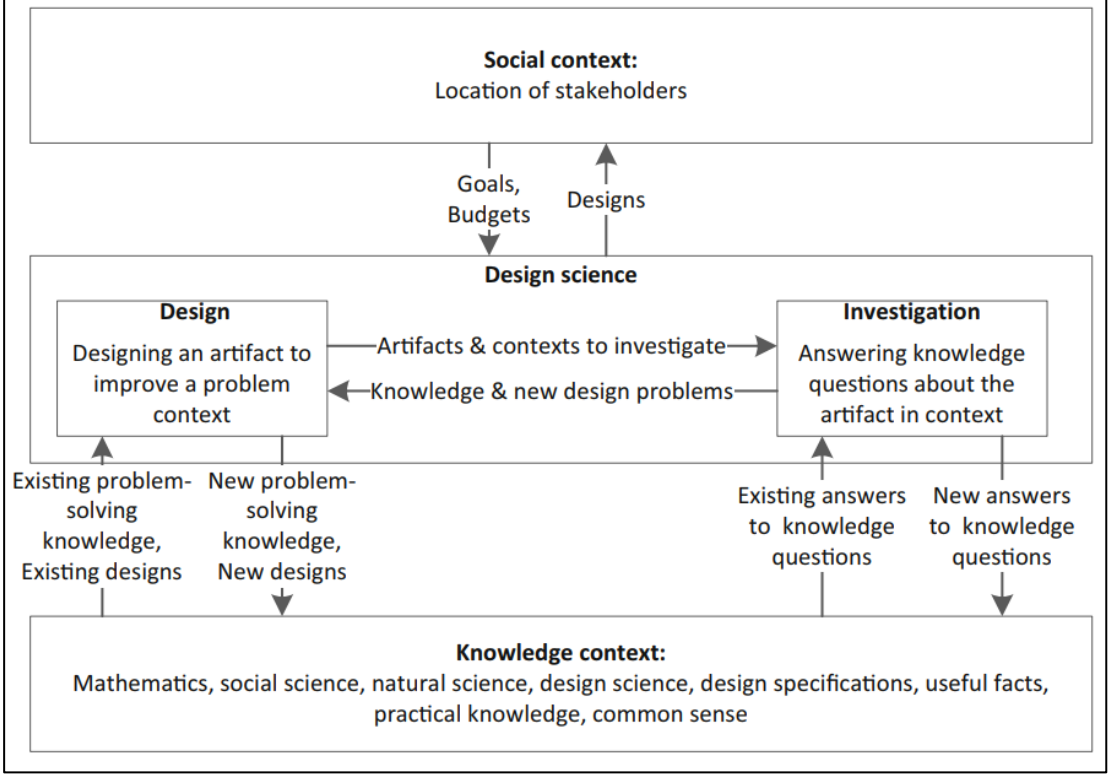


Figure 1.1: Framework for design science, image from Wieringa [84].

Figure 1.1 incorporates contexts into the interactions of design problems and knowledge questions [84]. The social context element at the top of figure 1.1 refers to the stakeholders of a particular project, many times being the fiscal beneficiaries of the development of a project. The knowledge context element at the bottom of figure 1.1. refers to the existing theories and what many times is considered the academic perspective of knowledge acquisition. We use this design science framework to help shape and direct the research presented herein.

Figure 1.2, inspired by [85], shows the problem decomposition overview of this research project, to address our problem statement. The orange box near the top of the figure presents the problem statement. The blue boxes on the left-hand side of the figure showcase the knowledge questions that we encountered, following the format of the design science framework [84]. The green boxes on the right-hand side of the figure are the empirical studies that were carried out to complete the empirical cycle, from a particular knowledge question. The chapter references for each empirical cycle are presented next to the green boxes. The figure is divided horizontally into three separate stages of research, part 1 pertaining to problem space identification, part 2 corresponding to a proposal of a solution that satisfies the problem, and part 3 corresponding to the studies that fulfilled the proposal. The remainder of this document follows the flow presented in this figure.

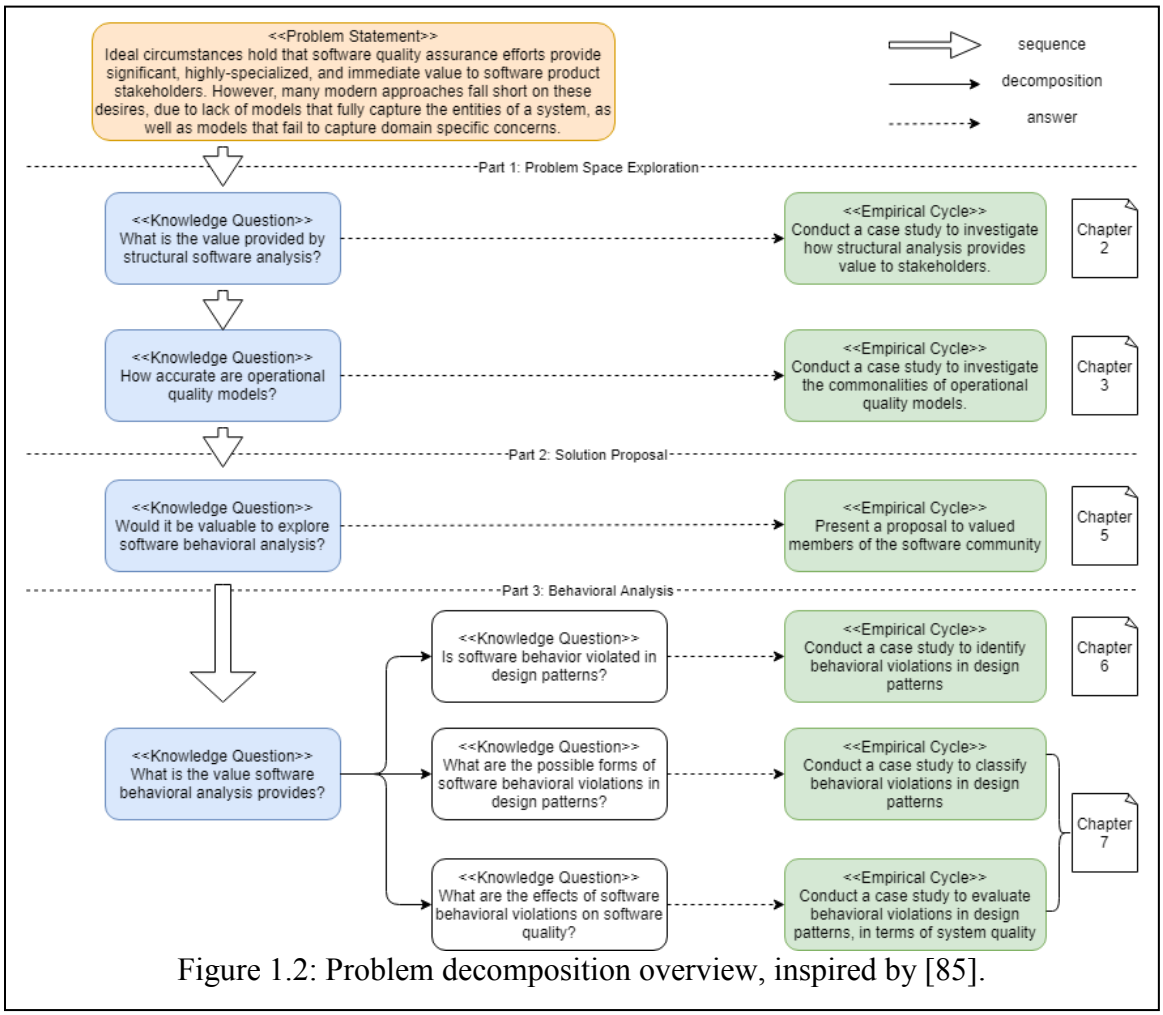

Figure 1.2: Problem decomposition overview, inspired by [85].

## 1.3 Chapter Overviews

Chapter 2 presents a replication case study performed in a commercial environment wherein a structural analysis technique, namely modularity violation detection, was applied to several versions of the software product. Modularity violations represent relationships between two or more files or classes in different modules that should not share a relationship. By themselves, relationships across modules are expected; such provide functionality to a project. However, when the relationship is not

expected, (i.e. a forgotten dependency), quality is sacrificed. In this study, we found that a select-few files contributed to the majority of modularity violations in the project. These files were also strongly correlated with modifications linked to bug-related fixes in **future** releases of the product. This suggests such files should be refactored to encourage less defects, good quality, subsequent quick releases of the product. However, the developers working on this product already knew about these files from their experience of the product, and had already begun efforts to improve the select few-files via refactorings. The research complemented developer intuition, suggesting that QA methods are capable of identifying issues that developers notice as well. Additionally, these results provided validity for developers, re-enforcing that their decision to refactor was a good decision.

Chapter 3 presents a case study that investigates the commonalities between five methods of estimating technical debt principal, compared to an external quality model. Both a correlation analysis and a regression analysis were performed to evaluate whether the technical debt estimation approaches can be related to the attributes of the QMOOD [7] quality model. Initially, the correlation analysis identified strong correlations between three of the estimate methods and system reusability and understandability. Though the further regression analysis yielded that with the exception of one technical debt estimation method (for flexibility and effectiveness) there was no observable relationship between the quality attributes and the technical debt estimates. This result suggests that state-of-the-art methods and tools for measuring software quality disagree on what they

consider to be quality factors, indicating that out-of-the-box implementations of quality models do not provide accurate estimates of quality.

Chapter 4 provides a brief interlude that synthesizes the results from Chapters 2 and 3 to identify a research gap. Chapter 2 illustrates the value that structural QA techniques provide to software stakeholders, and Chapter 3 suggests that out-of-the-box implementations of quality and TD analysis tools did not produce results that aligned with one another. This reveals a gap pertaining to behavioral analysis capabilities, which can complement existing structural approaches. Additionally, the introduction of domain-specific parameters into behavioral techniques will provide configurable tools that better estimate quality and TD.

Chapter 5 illustrates the results from a presentation [63] to the greater empirical software engineering community at the International Doctoral Symposium on Empirical Software Engineering (IDoESE'15), of a proposed plan of action to address this clear gap in the research. The paper describes a plan that explores behavioral deviations in the context of design pattern evolution, to ultimately supply practitioners and managers with more advanced and useful techniques to monitor and act on software QA. The feedback we received indicated that our four goals were ambitious, and it was suggested we reduce the scope of our work. We elected to remove requirements to predict behavioral deviations.

Chapter 6 describes a publication from the International Conference on Software Reuse (ICSR'19) conference [83], and a work-in-progress submission to the IEEE Transactions on Software Engineering. The paper presents the results from the remaining

two research goals. These goals are paraphrased as: (1) "*investigation of design pattern instances for the purpose of identifying and characterizing behavioral grime,*" and (2) "*quantify the impact of behavioral grime on quality and TD.*" To address the first goal, we constructed a taxonomy of design pattern behavioral grime that includes all known forms of behavioral grime, and is used as a complementary device to existing structural taxonomies. The taxonomy is published as part of a greater body of work [86]. We then evaluated the relationship between behavioral grime and structural grime, to illustrate how the two forms of analysis can complement one another. We found that strong relationships exists between five pairs of structural and behavioral grime, specifically TEER/PEE, PEER/TEE, PEAO/PI, PEAO/TEA, and PEAO/TI. To address the second goal, we extended an existing state-of-the-art operational quality model, QATCH [70], to incorporate model-based behavioral issues, and we used the extended model to evaluate the relationship between behavioral grime, quality and TD. We found that the presence of behavioral grime has a strong negative correlation with system quality, and a strong negative correlation with Maintainability, which serves as a surrogate measurement to TD.

# CHAPTER TWO

# A REPLICATION CASE STUDY TO MEASURE THE ARCHITECTURAL QUALITY OF A COMMERCIAL SYSTEM[1]

## 2.0 Abstract

**Context:** Long-term software management decisions are directly impacted by the quality of the software's architecture. **Goal:** Herein, we present a replication case study where structural information about a commercial software system is used in conjunction with bug-related change frequencies to measure and predict architecture quality. **Method:** Metrics describing history and structure were gathered and then correlated with future bug- related issues; the worst of which were visualized and presented to developers. **Results:** We identified dependencies between components that change together even though they belong to different architectural modules, and as a consequence are more prone to bugs. We validated these dependencies by presenting our results back to the developers. The developers did not identify any of these dependencies as unexpected, but rather architectural necessities. **Conclusions:** This replication study adds to the knowledge base of CLIO (a tool that detects architectural degradations) by incorporating a new programming language (C++) and by externally replicating a

[1] Based on:
Reimanis D., Izurieta C., Luhr R., Xiao L., Cai Y., Rudy G., "*A Replication Case Study to Measure the Architectural Quality of a Commercial System*," 8th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2014, Torino, Italy, September 2014.

previous case study on a separate commercial code base. Additionally, we provide lessons learned and suggestions for future applications of CLIO.

## 2.1 Introduction

Building confidence in previous results helps to increase the strength and the importance of findings. It is especially important to strive for external validation of results by independent researchers, as has been done by the replication study presented herein. To date, the field of Empirical Software Engineering lacks in the number of replication studies. Additionally, most of the existing guidelines found in the literature focus on formal experiments [8] [12] [40] [69]. In this paper, we present the findings of an external replication *case-study*. We present our results by borrowing from the existing experimentation terminology and we have structured our findings consistent with expected sections as delineated by Wohlin et al. [78].

The motivation behind this study stems from a desire to see if the techniques used by Schwanke et al. [68] to uncover architecture- related risks in a Java agile development environment (using architecture and history measures) can also be applied to a commercial C++ development environment. This is important because we wanted to evaluate the deployment of this technology in an industrial setting of a successful company with strict quality controls. We were also interested to see if the observations we make can be used to build consensus in explaining a form of architectural decay, where decay is defined as the structural breakdown of agreed upon solutions [39].

We applied CLIO [79], a tool designed to uncover modularity violations, to a commercial software system developed by a local bioinformatics company –Golden Helix[2]. The latter allowed us access to their software code base to investigate potential architectural disharmonies.

This chapter is organized as follows: Section 2.2 discusses background and related work; Section 2.3 explains the importance of replication in empirical software engineering and our approach to classifying this study; Section 2.4 discusses the method followed by our replication; Section 2.5 explores how the method was carried out, including deviations and challenges encountered from the baseline method, results and developer feedback; Section 2.6 discusses the relation of our results to the baseline study. Section 2.7 discusses the threats to validity in our study; and Section 2.8 concludes this chapter with lessons learned from this study and suggestions of future work.

## 2.2 Background and Related Work

### 2.2.1 Modularity Violations

Baldwin and Clark [6] define a module as "a unit whose structural elements are powerfully connected among themselves and relatively weakly connected to elements in other units." Identifying violations in modules (hereafter referred to as modularity violations) is important because it allows developers to find code that exhibits bad structural design. Identifying such violations early in the lifecycle leads to proactive module refactoring. However, early detection of modularity violations is difficult because

[2] Golden Helix Inc.; http://www.goldenhelix.com

they do not always exhibit negative influences on the functionality of the software system. It is entirely possible for a system to function as intended, yet still contain modularity violations. If these violations are left uncorrected, they can lead to architectural decay, which would slowly cripple production.

Zazworka et al. [82] used the modularity violations findings from a CLIO case study and compared them to three other technical debt identification approaches. They found that modularity violations contribute to technical debt in the Hadoop open source software system. Technical debt [19] is a well-known metaphor that describes the tradeoffs between making short term decisions (i.e., time to market) at the expense of long term but high software quality (i.e., low coupling). The debt incurred during the lifetime of a software system can be measured as a function of cost (monetary or effort) with added interest. Often, debt happens because of quick and dirty implementation decisions –usually occurring when a development team is trying to meet a deadline. Technical debt is dangerous if not managed because it can result in a costly refactoring process. Techniques to slow down the accumulation of technical debt can benefit from early detection of modularity violations.

### 2.2.2 CLIO

CLIO was developed by Wong et al. [79] as a means to identify modularity violations in code. Wong et al. evaluated CLIO by running it on two different open source Java projects, Eclipse JDT[3] and Hadoop Common[4]. The results showed that hundreds of

[3] The Eclipse Project; http://www.eclipse.org
[4] Apache Hadoop Common; http://hadoop.apache.org

violations identified by CLIO were fixed in later versions of the software. CLIO finds violations within modules by looking not only at the source code of a project, but also at its version history. It helps developers identify unknown modular level violations in software. Although developers will identify some violations, specifically if the violations prove to be bothersome, the difficulty of finding all modularity violations is quite great. CLIO validates that its reports are useful by confirming that previously detected violations are indeed fixed in later versions of the software. The results that Wong et al. [79] obtained showed that CLIO could detect these modularity violations much earlier than developers who were manually checking for them. This means that CLIO can be used in software systems to identify modularity violations early in the development process to save time and money by not having to check for them manually.

Schwanke et al. [68] expanded upon this work by using CLIO on an agile industrial software development project. They looked specifically at the architectural quality of the software. They used a clustering algorithm to observe how files changed together without developer knowledge, and the impact that change had on the quality of the architecture, as measured by source code changes because of bugs. They reported several modularity violations to developers. The developers issued a refactoring because the modularity violations were (1) unexpected and (2) possibly harmful to their system. CLIO allowed them to see the exact number of files that were dependent on one another, and how those changes were affecting the structure of their project.

## 2.3 Replication in Software Engineering

Literature in the field concerning guidelines of replication studies only addresses experimental replication, not case study replication [12] [69]. Therefore, we have borrowed terminology from this literature to inform our work.

### 2.3.1 Importance of Replicating Case Studies

Experiment replication plays a key role in empirical software engineering [12] [69]. While many other domains construct hypotheses *in vitro*, software engineers are generally not granted that luxury. Empirical software engineering frequently involves humans, directly or indirectly, as experimental subjects, and human behavior is unpredictable and not repeatable in a laboratory setting. Coupled with the prohibitive costs of formal experimentation, software engineering empiricists must look for alternatives. Instead, we must rely on repeated case studies in various contexts to construct a knowledge base suitable for a scientific hypothesis. This process, while requiring exhaustive work, allows for consensus building that can provide the necessary support to generate scientific claims.

### 2.3.2 Categories of Replication

Shull et al. [69] discuss two primary types of replications; *exact replications* and *conceptual replications*. Exact replications are concerned with repeating the procedure of a baseline experiment as closely as possible. Conceptual replications, alternatively, attempt to use a different experimental procedure to answer the same questions as the

baseline experiment. The study presented in this paper utilizes an exact replication method.

Shull et al. [69] divide exact replications into two categories: *dependent replications* and *independent replications*. In dependent replications, researchers keep all elements of the baseline study the same. In independent replications, researchers may alter elements of the original study. An independent replication follows the same procedure as the original study, but tweaks experimental treatments to come to the same or a different result. If treatments are changed and the same result is found, researchers can conclude that the treatment in question probably has little or no effect on the outcome. However, if changing a treatment leads to different results, that treatment needs to be explored further.

Using Shull's terminology, we categorized this study as an independent replication, with five major treatment differences from what would be considered a dependent replication. These differences are illustrated in table 2.1. First, the baseline study used a software project written in Java as their only treatment to the programming language factor. In our case, the treatment is the C++ programming language. In other words, our study lies in the context of a C++ programming language, which may provide different results from the baseline. Second, the comparative sizes of the development groups differed. The baseline study featured a development group of up to 20 developers working on the project at any given point in time [68]. The C++ system analyzed in this paper has had a total of eleven contributing developers in its four year lifetime. Third, the software project in the baseline study had been in development for two years, while the

project covered in our study has been in development for four years. Finally, the project in the baseline study features 300 kilo-source lines of code (KSLOC) in 900 Java files. The project in our study has 1300 KSLOC across 3903 source files, of which 1836 have a .cpp/.c extension, and 2067 are header files. Surprisingly, both projects have a similar ratio of LOC per source file (333 LOC per source file).

Table 2.1: Summary of different treatments between case studies

| *Factor* | *Baseline Project* | *Our Project* |
|---|---|---|
| Programming Language | Java | C++ |
| # of Developers | Up to 20 | Up to 11 |
| Project Lifetime | 2 years | 4 years |
| # Source Files | 900 | 3903 (1569 C++, 267 C, 2067 h) |
| KSLOC | 300 | 1300 |

### 2.3.3 Replication Baseline

In the selected baseline study, Schwanke et al. [68] reported on a case study that measured architecture quality and discovered architecture issues by combining the analysis of software structure and change history. They studied three structured measures (file size, fan-in, and fan-out) and four history measures (file change frequency, file ticket frequency, file bug frequency, and pair of file change frequency). Their study included two parts: 1) Exploring different software measures; and 2) Uncovering architecture issues using those measures.

1) *Exploring different software measures*: First, they explored the relationship between each pair of measures (structure and history) using Kendall's *tau-b* rank correlation [41], which showed the extent to which any two measures rank the same data in the same order. This study provided an initial insight on whether those measures were indicative of software quality, which was approximated by the surrogate file bug frequency. Then they studied how predictive those measures were of software faults. The data they used spanned two development cycles of the subject system, release 1 (R1) and release 2 (R2). They illustrated how predictive the calculated measures from R1 were for faults that appeared in R2 using Alberg diagrams [56].

2) *Uncovering architecture issues*: After validating the measures, they were used to discover architecture issues using three separate approaches. First, Schwanke et al. [68] ranked all files by different measures –worst first. They found that the top ranked files (outliers) were quite consistent for different measures. They showed those outliers to the developers to obtain feedback about potential architecture issues; however, the developers gave little response because they could not visualize these issues. To generate responses from developers, they used a static analysis tool named *Understand*™[5] to visualize the position of those outliers in the architecture. Using this method, they were able to discuss many of the outlier files with the developers. In some cases, the developers pointed out how severe the problems were. Finally, they used CLIO to investigate the structure and history of pairs of files and grouped structurally distant yet historical coupled files into clusters. For each cluster, its structure was visualized using *Understand*™ in a structure diagram,

[5] Understand; http://www.scitools.com

which illustrated how clusters which cross-cut different architecture layers could be severe, and gave hints about why they were coupled in history.

### 2.3.4 Major Findings of the Baseline

Schwanke et al. [68] found that by using CLIO they could identify, predict, and communicate certain architectural issues in the system. They found that a few key interface files contributed to the majority of faults in the software. Additionally, they discovered that the file size and fan-out metrics are good predictors of future fault-proneness. In the absence of historical artifacts, files that contain high measures of these metrics typically have a higher number of architectural violations. Finally, unknown to the developers, some of these files violated modularity in the system by creating unwanted connections between layers. These violations were visualized and presented to the developers who issued a refactoring thereafter.

## 2.4 Procedure

Following the procedure outlined in [68], our case study consisted of the following steps:

1) Data collection: The source code, version control history, and ticket tracking history of the software system in question were gathered.
2) Structure and history measurements: Measurements for common metrics were computed/collected across all versions of the software.
3) Validation: Measurements from the second-most recent release are correlated with fault measurements from the most recent release.

4) Prediction: Measurements from the most recent release are used to predict faults in upcoming future releases of the project.
5) Uncovering architecture problems: Measurements were sorted according to future fault impact and visualized. Outlier measurements present the most concern to system architecture quality, and were selected for further exploration.
6) Present findings to developers: Visualizations of the architecture of outlier modules were presented to developers with the intent of helping to realize the architectural quality of the system.

## 2.5 Case Study

### 2.5.1 Setting

The project analyzed in this case study is named SNP & Variation Suite (SVS), and is the primary product of the bioinformatics company Golden Helix. We analyzed seven major releases of SVS.

SVS features 1.3 million lines of C++ source code spread out across 3903 source files. The project's structure is spread out across a total of 22 directories. In this study, we have chosen to define module as a directory, based on Parnas el al.'s definition [60]. We use the term directory and module interchangeably.

Eleven developers have contributed to this project over its four- year lifetime. The organization of the development group has an interesting hierarchy. The lead developer is also the Vice President of Product Development at Golden Helix. He plays a major role in not only developing SVS, but also in managing product development from a financial perspective. This means he has comprehensive knowledge of the software system when he

makes management-related decisions, and therefore, is more aware of the technical debt present in the software than business- oriented managers.

### 2.5.2 Motivation

This project was chosen for three reasons. First, Golden Helix is a local software company with its developing team in close proximity to the authors, and is well known for their generous contributions to the community. The process presented in this study is a great opportunity to inform Golden Helix of the architectural quality of their flagship software. Second, applying the CLIO tool in different commercial settings will help future applications of CLIO. By clearly outlining the strengths, weaknesses, and lessons learned at the end of the study, we hope to improve future applications of CLIO. Finally, no previous study that follows this methodology to detect modularity violations has considered a C++ project. Previous studies such as [79] [82] only looked at non-commercial Java projects. Using the C++ programming language as a treatment in this sense builds on the knowledge base of CLIO, extending what we know about this method.

### 2.5.3 Data Collection

Golden Helix strongly encourages developers to commit often, and keep commits localized to their section of change. These commits are stored in a Mercurial[6] repository, and the FogBugz[7] tool is used to track issues. Golden Helix switched repositories, from Apache Subversion (SVN)[8] to Mercurial, and ticket tracking tools, from Trac[9] to

[6] Mercurial SCM; http://mercurial.selenic.com/
[7] FogBugz Bug Tracking; https://www.fogcreek.com/fogbugz/
[8] Apache Subversion; http://subversion.apache.org/
[9] Trac; http://trac.edgewall.org/

FogBugz, during the lifetime of SVS. Because this study focuses on the entirety of the project's lifetime, both the SVN repository and Trac ticket logs have been recovered and treated in the same manner as the current system. Each developer is expected to include references to ticket cases in their commits.

Similar to [68], the repository logs and issue tracking logs were extracted into a PostgreSQL[10] database. This allowed us to search for historical data using simple SQL queries. We have grouped C/C++ source files and header source files together in this study. That is, for each C/C++ source file and its corresponding header file(s), the files are considered one and the same. For the remainder of this case study, we refer to the C/C++ source and corresponding header file pairs as a *file pair*. Measurements made in both files are aggregated together. There is a reason for doing this. Developers of SVS demand that source files and their corresponding header files be kept together in the same directory. When either a source file or a header file changes, the developers are expected to update the signatures in the corresponding file. This implies that any changes made to the latter are expected and hence do not constitute modularity violations. Our study is concerned with locating *unexpected* changes in modules of code. Therefore, including any information about header/source pairs changing together will lead to useless information.

### 2.5.4 Structure and History Metrics

Following the work of Schwanke et al. [68], the following metrics were gathered for all file pairs (*u)* across all seven versions of the software:

[10] PostgreSQL; http://www.postgresql.org/

1. *File size:* The aggregated file size on disk of both elements in *u*, measured in bytes.
2. *Fan-in:* Within a project, *fan-in* of *u* is the sum of the number of references from any *v* (where *v* is defined identically similarly to *u*) pointing to *u*.
3. *Fan-out:* Within a project, *fan-out* of *u* is the sum of the number of references from *u* that point to any *v* (where *v* is defined identically similarly to *u*).
4. *Change frequency:* The number of times that any element in *u* is changed, according to the commit log. Commits where both elements of *u* are changed are only counted once.
5. *Ticket frequency*: The number of different FogBugz or Trac issue tickets referenced for which either element in *u* is modified. If both elements in *u* are modified with a reference to the same issue ticket, it is only counted once.
6. *Bug change frequency*: The number of different FogBugz or Trac **bug** issue tickets referenced for which either element in *u* is modified. If both elements in *u* are modified with a reference to the same **bug** issue ticket, it is only counted once.
7. *Pair change frequency*: For each file pair, *v*, in the project, the number of times in which *u* and *v* are modified in the same commit.

### 2.5.5 Validation

In an effort to validate the significance of our metric choices, several exploratory data analysis techniques were utilized. These include histogram inspection, scatter plot analysis, and correlation analysis. Although the system in question has gone through seven releases, in this paper we only present the results from the most recent release

(release 7.5) and the release immediately preceding the most recent release (release 7). Hereafter, we refer to release 7.5 as the *present* state of the software, and release 7 as the *past*.

Similar to the baseline study, we found that data analysis across all other releases showed very similar results. The baseline study chose to focus their work on the most recent releases, because it is more representative of the system in the present time, and may provide better predictive power. We have followed suit because of the same reasons.

*1)Histogram analysis*

Histograms were generated for each metric in question. We focused on identifying distributions of each metric across releases. From the distributions, we identified outlier file pairs which Schwanke et al. [68] states are more prone to unexpected changes. For example, figure 2.1 illustrates the change frequency metric across all releases of the software. The y-axis is shown as a logarithmic scale in base 4 to preserve column space. There is a typical exponential decay curve, suggesting that the majority of file pairs experienced few changes. However, there exist outliers with more than 180 changes per file (not shown, but aggregated to form the bin at x=180). This suggests that a surprising number of pairs (about 60) experience more than 180 changes.

This is congruent with findings from [68] and their histogram analysis.

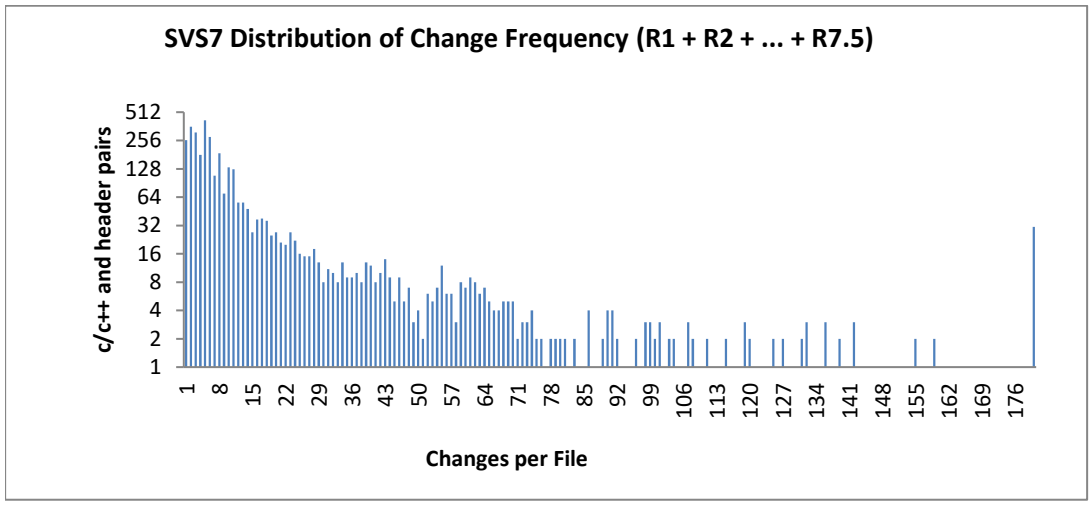


Figure 2.1: Histogram of change frequency across all releases. The x-axis shows change frequency. The y-axis shows a count of the number of pairs. Any pair with 180 or more changes was considered to be an outlier, and likely to contribute to many unexpected dependencies

*2)Scatter Plot Analysis*

Scatter plots were constructed for each metric gathered. When constructing scatter plots, we plotted the measure in release 7.5 on the y-axis and the measure of other metrics from release 7 on the x-axis. This gave us the opportunity to identify a possible relationship between past and present measurements. Figure 2.2 shows a scatter plot of change frequency in release 7.5 versus fan-out in release 7. There appears to be a slight linear correlation between the two, suggesting that change frequency in future releases can be predicted from fan-out in current or past releases.

This graph suggests that the fan-out of current or past file pairs may be used to predict the change frequency of the pair in the future. Our scatter plot analysis provided

similar results as the baseline study by Schwanke et al [68].

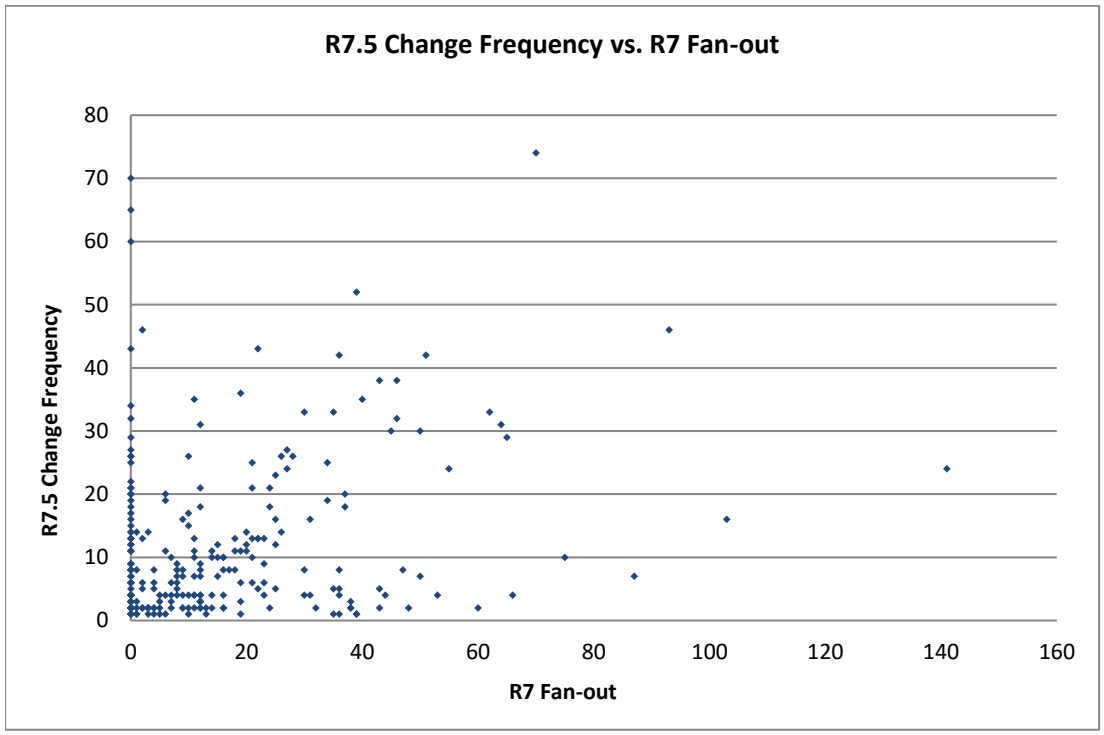


Figure 2.2: Scatter plot of release 7.5 change frequency and release 7 fan-out. Each data points represents a C/C++ and header pair. The x-axis plots the fan-out of pairs in release 7. The y-axis plots the change frequency of each pair in release 7.5. There appears to be a linear correlation between the two, suggesting that change frequency in future releases can be predicted from fan-out in current or past releases.

*3)Correlation Analysis*

Rank-based correlation analysis was performed on the data to identify possible relationships between measurements in one release and fault measurements in a future release. Per the baseline study, we used the Kendall's *tau-b* rank correlation measure [41]. This non-parametric test was chosen instead of a Spearman or the parametric Pearson test because many of the values fall near zero. The Ordinary Least Squares (OLS) method of Spearman or Pearson performs poorly when many values fall near zero.

Kendall's *tau-b* value is found in a two-step process. First, the measurements taken from two metrics are ordered according to their values. Second, a calculation is performed which counts the number of values which appear in the same order. The calculation is shown below:

$$\tau_B(F,G) = \frac{concord(F,G) - discord(F,G)}{concord(F,G) + discord(F,G)}$$

Where *F* and *G* are two orderings of values taken from a file pair. *concord(F,G)* is a count of the number of times values appear in the same order. Alternatively, *discord(F,G)* is a count of the number of times values appear in different order. For this test, values of 0 in either *F* or *G* are ignored; that is, they are not counted by either *concord* or *discord*. The value produced falls in range [-1, 1], corresponding to the correlation between the orderings. A value of 1 indicates a perfect linear correlation. For the purpose of this study, and in agreement with [68], we consider values at 0.6 or greater to be strong. Because this is a non-parametric statistical test, we cannot assume a normal distribution fits the data. Therefore, we cannot find an associated *p*-value for each *tau-b* value.

Table 2.2 shows the *tau-b* value calculated for each metric pair in release 7 and release 7.5. Each cell corresponds to the *tau-b* value as found by the previously described equation. The table is symmetric because the comparison of two ranked metric values is a symmetric property. Highlighted cells indicate a strong correlation.

Table 2.2: Tau-b values for metric pairs

| *Tau-b* table of metrics for svs7 + svs7.5 | | | | | | |
|---|---|---|---|---|---|---|
| r7+r7.5 | fan-in | fan-out | file size | changes | tickets | bugs |
| Fan-in | 1 | 0.257 | 0.301 | 0.331 | 0.328 | 0.464 |
| Fan-out | 0.257 | 1 | 0.441 | 0.417 | 0.416 | 0.637 |
| size | 0.301 | 0.441 | 1 | 0.293 | 0.273 | 0.510 |
| changes | 0.331 | 0.417 | 0.293 | 1 | 0.972 | 0.858 |
| tickets | 0.328 | 0.416 | 0.273 | 0.972 | 1 | 0.857 |
| bugs | 0.463 | 0.637 | 0.510 | 0.858 | 0.857 | 1 |

The highlighted values in the bottom right quadrant of the table are expected correlations. The values report that, for example, as ticket frequency increases, bug change frequency increases as well. This is logically consistent because as developers add more tickets to their commits, more of these tickets will contain bug references. However, the correlation value for bugs vs. fan-out is an unexpected result. This number tells us that as the fan-out of a file pair increases, the number of bugs associated with that pair increases as well. Similar results were found by [68], adding more power to hypothesis that fan-out and number of bugs increase together.

Using these three methods of exploratory data analysis, we identified likely correlations between metrics. In the validation step we analyze these correlations to see if they are indicative of bug-related changes in the future.

### 2.5.6 Prediction

Ostrand and Weyuker [58] introduced *accuracy*, *precision*, and *recall* measures from the information retrieval domain. We use various *recall* metrics to validate our prediction of future bugs. Recall is defined as the percentage of faulty files that are

correctly identified as faulty files. As in the baseline case study, we calculate recall in three different ways. For every file pair *u,*

*Faulty file recall:* An instance occurs when either element in *u* is changed at least once in the release representing the future due to any bug ticket.

*Fault recall*: An instance is a tuple defined as <*u*, bug ticket reference>, where *u* is changed at least once due to the same bug ticket.

*Fault impact recall*: An instance is a triple defined as <*u*, commit number in the source control logs where *u* is changed, bug ticket reference> where the bug ticket is referenced in the same commit where *u* is changed in.

These three recall measures apply different emphasis to future fault prediction. *Faulty file recall* emphasizes future fault prediction least, because it treats all future bug-related changes to *u*, regardless of the number of instances, as one. This fails to capture instances where *u* is associated with more than one bug ticket. However, *Fault recall* does take this into account, because it considers multiple bug ticket references in an instance. Furthermore, *Fault impact recall* provides the highest granularity to allow for future fault prediction because it takes into account all changes *u* goes through. All three recall measures form an implied subsumption hierarchy.

Using these recall measures, we use Alberg diagrams [56] to plot release 7 measurements vs. release 7.5 faults. Alberg diagrams are based on the pareto principle, that roughly 20% of the files in a system are responsible for 80% of the faults. In this context, we use this same principle to estimate the accuracy of prediction models [56].

Figure 2.3 illustrates one Alberg diagram for this system. The x-axis shows 60 C/C++ source and header pairs, *u*, ordered in descending order according to their metric values from release 7. These 60 file pairs are selected based on their contribution to bug-related changes in release 7.5. The bug change frequency for *u* in release 7.5 is plotted on the y-axis. Any given point on the curve represents a C/C++ source and header pair. The oracle curve is a perfect predictor of release 7.5 bug change frequency for all *u*. As other curves get nearer to the oracle curve, their accuracy for predicting release 7.5 bug change frequency increases.

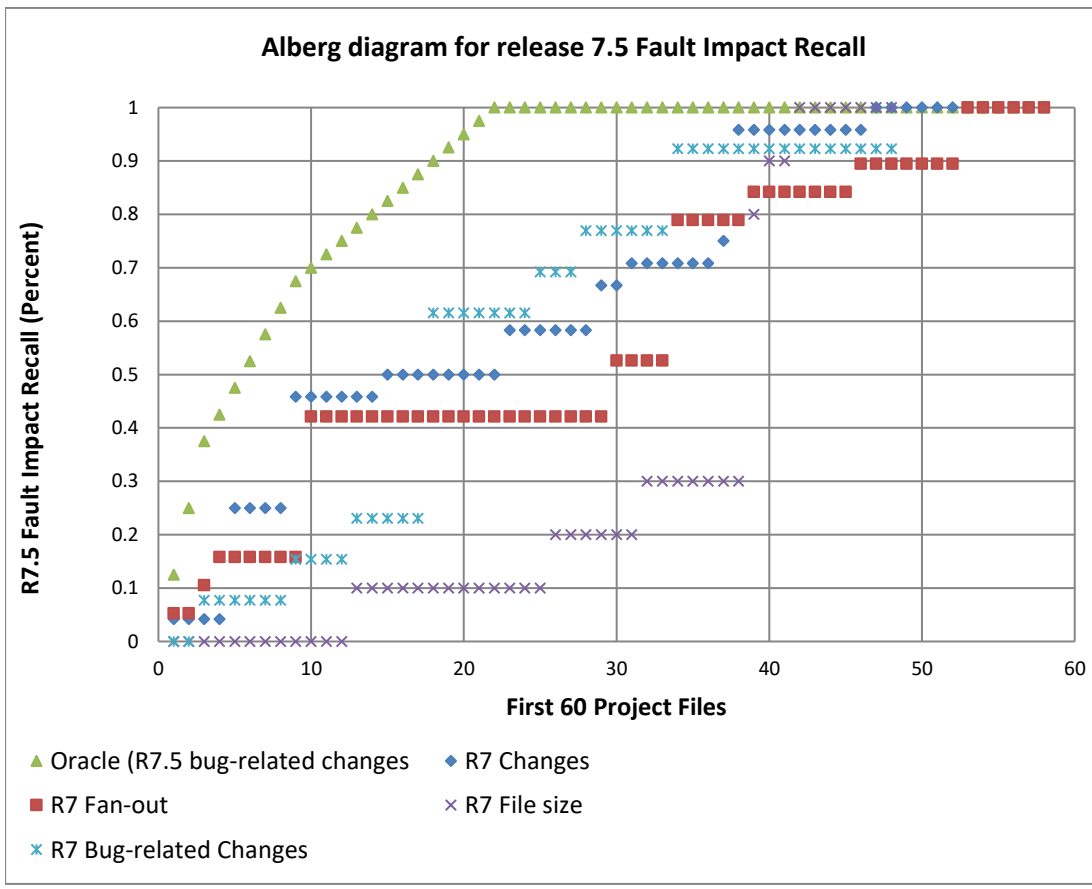


Figure 2.3: Alberg diagram of release 7.5 Fault Impact Recall. Each data point is a C/C++ source and header pair. The x-axis represents the rank of a data point in release 7, sorted according to its type. The y-axis represents the percentage of bugs in release 7.5.

The oracle curve from this Alberg diagram states that roughly 20% (actually 23.3%) of C/C++ source and header pairs contribute to 80% of bug change frequency in release 7.5. The values of fan-out and change frequency in release 7 for these pairs contributed from 40% to 50% of bug changes in release 7.5. These findings are slightly less than Schwanke et al.'s findings [68], yet are still noteworthy. This validates that selected metrics from earlier releases can be used to predict bug change frequency in future releases.

### 2.5.7 Uncovering and Visualizing Architecture Problems

Once these measures have been validated as capable of predicting future faults, the problem of identifying file pairs which are more prone to unexpected changes arises. Next, we study the extent to which these pair affect other quality measures.

We utilized the static code analysis tool *Understand*™ to visualize graphs of interdependent components. *Understand*™ is a commercial product developed by Scientific Tools, Inc.[7] *Understand*™ can find many structural features of code, including dependency listings of how pairs of C++ files depend on one another. Through visualization, we can analyze the extent to which these dependencies affect other pairs in the software system.

These graphs help differentiate expected and unexpected dependencies. If dependencies occur between two pairs that are in the same module, we treat them as *expected* dependencies, consistent with the baseline study. This is based on the assumption that developers group files or classes together based on similar functionality. *Unexpected* dependencies are treated as dependencies that occur across different

modules, also consistent with our baseline study. Our definitions of expected and unexpected dependencies were validated by the developers at Golden Helix.

Because we are concerned with how these dependencies are changing together, we define a "distant" and "local" change pair. Using Schwanke et al.'s [68] definitions, a pair of file pairs that change together, *change pair, <u,v>* is local if (1) *u* directly depends on *v*, (2) *v* directly depends on *u*, or (3) *u* and *v* belong to the same module. Any change pair which does not fit under this definition is a *distant change pair*.

Figure 2.4 illustrates a high level view of the dependencies between modules in SVS. Nodes in the graph represent modules, and edges represent dependencies between modules. The number on the edge refers to the exact number of dependencies. The modules shown contain the ten most frequent distant change pairs. This graph is nearly a complete graph, suggesting that modules have high coupling when distant change pair frequency is high.

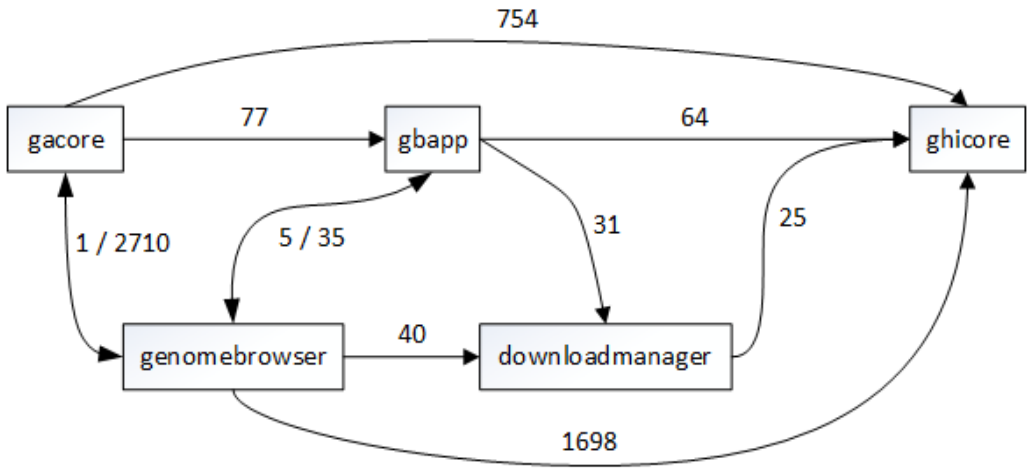


Figure 2.4: High level view of the dependencies between modules containing the ten most frequent distant change pairs. The numbers on the edges represent the number of times all file pairs in a module depend on another module.

Once change pairs have been classified as either local or distant, CLIO is used to

(1) identify change pairs which historically have changed together frequently, and (2) cluster these pairs according to the scope of their change pair (local or distant). To identify frequent historic change pairs, we mine the PostgreSQL database built in the procedure described by section 5.1. To cluster the pairs, a "single link" clustering algorithm is used [68].

The clustering algorithm groups distant change pairs as follows: For each frequent, distant change pair <*u*, *v*>, cluster *u* and *v* together. Then, add all the local dependencies which contain either *u* or *v* to the cluster. We generated visualizations of these clusters that illustrate the number of dependencies across distant change pairs and presented these visualizations to developers.

### 2.5.8 Presenting Results to developers

Visualizing architectural dependencies with graphs provided us with a convenient and intuitive medium that could be validated with developers. We presented all our data to the lead developer at Golden Helix. In summary, the lead developer at Golden Helix was not surprised by our findings. He indicated that several outlier file pairs were contributing to the majority of modularity violations in the code base. It was these pairs that also contributed to a large number of bugs in the most current releases. The lead developer was well aware of this, and more or less the extent to which this affected other files.

The majority of modularity violations and bugs occurred in packages representing highly customizable components of the SVS executable. These packages include the UI component, the core component, and a component that is concerned with reading in a

large variety of complex file formats. We noticed that file pairs in these packages both heavily depend on and were depended upon by many others (i.e., they have high efferent and afferent coupling). However, the structure observed was the choice of the developers. The developers utilized these pairs as access points, or common files to reference when one component needed to be used. When these access point pairs were changed, they incurred a slew of changes in other modules in the system because of numerous, propagating dependencies. The developers saw this method as a necessary step in their development lifecycle.

## 2.6 Discussion

The process of using CLIO to detect and measure architectural quality of software needs to be matured further. Developers were not surprised by the findings of CLIO, primarily because the findings pointed out known problems. Many of these problems are due to the many connections that exist between modules. From an academic sense this is a problem, because it is preferable to have few connection points between modules (coupling). Lower coupling between modules is indicative of better design, and helps localize possible future changes as well as allows for increased quality attributes (such as understandability) [7]. However, from the developers' perspective, familiarity with the code base was more important than traditional good design. The developers are content leaving the coupling between modules as is, because it makes the most sense for the SVS system. This finding is very interesting because it gives the impression that the results from tools such as CLIO should be system dependent. That is, although the results may

appear useful, nothing can be learned unless an in-depth assessment of the software system in question has been made. These conclusions cannot be reached without evaluating and deploying laboratory tools in commercial grade environments.

We did find very similar results to the baseline, which is promising in helping extend power of the hypothesis that certain metrics can be used as better predictors of software quality. We found that a select few files contributed to many modularity violations, and greatly influenced the number of bugs. While in our case the developers were not surprised by the results, the results are promising in that they clearly identify problem files in code. The baseline found that developers were not always aware of these modularity violations. In cases where developers may not be fully familiar with the structural connections across modules in their code base, this procedure provided significant insights.

We also identified and validated cases where structural metrics can be used as quality predictors for future releases. Both this study and Schwanke et al. [68] concur that the fan-out metric is a good predictor of future faults, as verified by correlation analysis and Alberg diagrams.

## 2.7 Threats to Validity

There are several threats that threaten the validity of this study. One developer brought up the argument that, “If a developer prefers to commit files more frequently than other developers, it would show up in the commit logs as having few change pairs. This would give misleading results because it would provide cases where too few files

are being committed to account for changes across modules, or too many files are being committed which would make it appear that more dependencies exist." This is a direct threat to the construct validity of our study. Although the developer's observation is correct, it did not have a large impact on our results. We identify files showing up in the commit logs together with a high frequency, and ignore cases where paired changes happen infrequently. This reinforces that such cases as described by the developer are unlikely to occur often. Regardless, the observation does shed light into a situation that will be mitigated in future studies.

A second threat to the construct validity is the fact that we grouped C/C++ source file and corresponding header files together. These file pairs consist of the aggregated information from their combined elements. Although a threat, it is mitigated by the following reason. The developers brought to our attention that both elements in the file pair are expected to belong to the same package, and are expected to change together. That is, if a C++ source file is updated, the developers expect to make changes to the signature of the header source file as well. Because both of these cases are expected changes, including both files separately in the study would be spurious information. Thus, we chose to group every C++ source and corresponding header file together.

A third threat to the construct validity of this study is the assumption that developers tag bugs correctly in the commit messages. As an external observer, the only method we have of identifying past-bugs in the software project is through analyzing historical artifacts. Therefore, we need to rely on the discipline of developers to (1) tag

the bugs they focused on in a commit and (2) tag the bugs correctly. We have no way of knowing if either of these two conditions is not met.

External validity represents the ability to generalize from the results of a study. In this instance, we cannot generalize the results we found to other contexts. In other words, the results found in this study and the baseline only hold true for our specific contexts, however they helped in building consensus around our findings across different programming languages in commercial agile development environments. More replication studies are necessary to increase the power of these results.

## 2.8 Conclusion

This replication case study was performed to help us analyze how structural file metrics could be correlated with system quality, and to help us comprehend if similar observations performed in a Java commercial product could also be observed in its C++ counterpart. We have gathered structural metrics and identified correlations between them and future bug problems. We identified a select few outlier files which contribute to the majority of future bug problems. From these, we collected dependencies and visualized how extensively problems may propagate. We showed this information to the developers of Golden Helix and they were not surprised by the results. Rather than attempt to entirely eliminate distant-modules with frequently-changing dependencies, the developers preferred to keep a select-few files as connection points. When asked why, the lead developer explained that these connection points offer a single point of entry into a module. Any changes between modules would be reflected in the connection points only.

The developers would rather be aware of a few files that are frequently problematic than issue a refactoring.

## 2.9 Challenges

Herein we describe some of the challenges we encountered while trying to perform this study.

1) *Specific Tools*: The baseline study featured the use of the commercial tool *Understand*™ for static analysis of code to gather metrics as well as to visualize results. Although the static analysis and visualizations provided high quality analysis, it is nearly impossible to replicate this case study without the use of this specific tool. Alternatives were considered, but the mechanistic formula used for analyzing files needed to be used as is, as other approaches would have constituted (in the opinion of the authors) a significantly large deviation from the baseline method that we would not have been able to call this a replication study.
2) *Understanding the System:* While we hope that manually performing the CLIO process eventually leads to an automated approach, this study suggests that such a hope may be far-fetched. Ultimately, a complete understanding of the system in question is necessary before any significant value can be taken from this tool. Our results mean very little unless the developers actually make use of them.
3) *Literature Coverage*: The majority (entirety) of literature covering replications in Empirical Software Engineering refers to formal experiments, not case studies. We have borrowed the terminology from such literature in this study. This situation is not ideal

because case studies have less power than formal experiments and therefore should be approached differently. Peer reviewed literature needs to be published which outlines case study replication guidelines.

# CHAPTER THREE

# THE CORRESPONDENCE BETWEEN SOFTWARE QUALITY MODELS AND TECHNICAL DEBT ESTIMATION APPROACHES[11]

## 3.0 Abstract

This Chapter summarizes a report that identified a gap in the capabilities of modern QA research tools. In this motivational research, we performed a case study that analyzed the similarities between results of state-of-the-art operational quality and TD measurement tools. We estimated quality and TD across 10 releases of 10 open source systems and found that only one TD estimation technique had a strong correlation to the quality attributes of reusability and understandability. In a multiple linear regression analysis, we also found that a single different TD estimation technique had a significant relationship to the quality attributes of effectiveness and functionality. These results indicate that a gap exists within the state-of-the-art; specifically that the results of operational quality and TD estimation tools disagree.

[11] Based on:
Griffith I., Reimanis D., Izurieta C., Codabux Z., Deo A., Williams B., "The Correspondence between Software Quality Models and Technical Debt Estimation Approaches," IEEE ACM MTD 2014 6th International Workshop on Managing Technical Debt. In association with the 30th International Conference on Software Maintenance and Evolution, ICSME, Victoria, British Columbia, Canada, September 30, 2014.

## 3.1 Introduction

The desire to measure the quality of a software product has existed nearly as long as software engineering itself [27]. Because of this, several operational models that estimate software quality have surfaced in industry. Largely, these models perform static analysis of a code-base to identify the degree to which code aligns to quality goals, such as ‘Security’ or ‘Maintainability’. Complementary to software quality is Technical Debt (TD), which is a metaphor established by Ward Cunningham to describe the gap between the current state of a software system and the ideal state [20]. In essence, TD captures the effects of decisions that sacrifice good design principles for on-time delivery of software. Many times these decisions take the form of shortcuts or workarounds in code that complete the task at hand, but at the expense of decreased quality. TD is analogous to financial debt in that some debt is beneficial, because it facilitates growth, but too much debt becomes a burden because of the need to repay it at the expense of valuable resources. Drawing parallels from financial debt, principal and interest are two attributes of TD. Given a task to implement, principal refers to the cost in effort to complete the task. Interest refers to the gap between maintenance costs under ideal conditions versus conditions where maintenance is higher due to accrued debt from tasks where TD is not repaid. Effectively managing TD is multifaceted problem, because the need the need to implement new features must be leveraged with the need to refactor to cleanse the code-base.

This motivational research poses the following question: *What does the estimate of technical debt provided by approach X mean in the context of quality model Y?* In

other words, how can we evaluate the accuracy of technical debt estimation approaches in the context of an external quality model? Capturing the relationship between TD estimation methods and software quality will reveal the accuracy of the various TD estimation approaches.

To perform this study, we considered three TD estimation approaches. These approaches are the SonarQubeTM1 TD-Plugin [32], CAST's method of technical debt estimation identified by Curtis, Sappidi, and Szynkarski [20][21], and Marinescu's method of technical debt estimation using design disharmonies [49]. We used three versions of the CAST TD estimation method, each version capturing different high-level goals from an organization's perspective. In total, we evaluated five TD estimation approaches against the QMOOD quality model [7].

## 3.2 Background and Related Work

### 3.2.1 TD Estimation Techniques

The first TD estimation method we implemented was the SonarQube™ TD-Plugin [32]. This method uses the following formula to calculate the technical debt value [32]:

$$Debt = duplication + violations + comments + coverage + complexity + design \quad (1)$$

$$duplication = cost\_to\_fix\_one\_block * duplicated\_blocks \quad (2)$$

$$violations = cost\_to\_fix\_one\_violation * mandatory\_violations \quad (3)$$

$$comments = cost\ to\ comment\ on\ API * public\_undocumented\_API \quad (4)$$

$$coverage = cost\_to\_cover\_one\_of\_complexity * uncovered\_complexity\_by\_tests \quad (5)$$

$$design = cost\_to\_cut\_an\_edge\_between\_two\_files * package\_edges\_weight \quad (6)$$

$$complexity = cost\_to\_split\_a\_method * (function\_complexity\_distribution \geq 8) + cost\_to\_split\_a\_class * (class\_complexity\_distribution \geq 60) \quad (7)$$

Where *duplication, violations, comments, coverage, complexity* and *cycles* secondary formulas are each measured in man-days. Each of the costs used in the secondary formulas can be set as parameters. We used the default values as described by table 3.1. *Duplication* refers to the estimated effort associated with the removal of duplicated code in the system. *Violations* is the estimated effort associated with the removal of violations in the system. *Coverages* represents the estimated effort required to bring test coverage up to 80%. *Complexity* is the total estimated effort required to split every method and every class (of those requiring such a split). *Comments* refers to the estimated effort associated with documenting the undocumented portions of the API. *Design* refers to the estimated effort associated with cutting all existing edges between files. Each of the cost (estimated effort) (table 3.1) are defined in man-hours, in order to convert this to man-days for the debt calculation, the default value of 8 hours per day is used. A final calculation is then performed to evaluate the cost per man-day of technical debt using a default value of $500.

Table 3.1 Default cost values used in the calculation of Technical Debt in the SonarQube TD-Plugin [4]

| Cost | Default Value (in man-hours) |
|---|---|
| *cost_to_fix_one_block* | 2 |
| *cost_to_fix_one_violation* | 0.1 |
| *cost_to_comment_one_API* | 0.2 |
| *cost_to_cover_one_of_complexity* | 0.2 |
| *cost to split a method* | 0.5 |
| *cost_to_split_a_class* | 8 |
| *cost_to_cut_an_edge_between_two_files* | 4 |

The second TD estimation method we chose was developed by Curtis, Sappidi, and Szynkarski [20][21], which estimates technical debt principal using a cost model based on detected violations. This method uses estimates of *time to fix* and *cost to fix* in order to connect these identified violations to a monetary value. The following equation is proposed as a means to measure the technical debt principal:

$$\begin{aligned} TDE = {} & (\Sigma HS * \%HS * \overline{HS_f} * HS_{cost}) \\ & + (\Sigma MS * \%MS * \overline{MS_f} * MS_{cost}) \\ & + (\Sigma LS * \%LS * \overline{LS_f} * LS_{cost}) \end{aligned} \quad (8)$$

Where $\Sigma HS$, $\Sigma MS$, and $\Sigma LS$ are the count of *high severity*, *medium severity*, and *low severity* violations respectively. The values for *%HS*, *%MS*, and *%LS* represent the percentages of high, medium, and low severity violations intended to be fixed. The values of $\overline{HS_f}$, $\overline{MS_f}$, and $\overline{LS_f}$ represent the average time (in hours) required to fix per instance of each severity level. Finally, the values of $HS_{cost}$, $MS_{cost}$, and $LS_{cost}$ represent the cost in monetary value per hour to perform the work. Curtis, Sappidi, and Szynkarski, provide three estimates for technical debt (see table 3.2).

Table 3.2 Values for estimates of TDE as proposed by Curtis, Sappidi, and Szynkarski [21].

| | Violation Severity Level | Estimate 1 | Estimate 2 | Estimate 3 |
|---|---|---|---|---|
| *Percent of Violations to be Fixed* | High Severity | 50% | 100% | 100% |
| | Medium Severity | 25% | 50% | – |
| | Low Severity | 10% | – | – |
| *Hours to Fix* | High Severity | 1hr | 2.5 hrs | 10% - 1 hr<br>20% - 2 hrs<br>40% - 4 hrs<br>15% - 6 hrs<br>10% - 8 hrs<br>5% - 16 hrs |
| | Medium Severity | 1 hr | 1 hr | – |
| | Low Severity | 1 hr | – | – |
| *Cost per Hour* | All Severity Levels | $75 | $75 | $75 |

The final TD estimation method we chose was developed by Marinescu [49]. This method utilizes design disharmonies in the software to derive an index of the underlying issues in quality. Marinescu proposes that we measure the impact of these design disharmonies based on how they influence the underlying design, the level of granularity at which they manifest themselves (class or method) and the underlying severity of the disharmony based on the amount of code it impacts. Here the influence, $I_{disharmony}$, is one of the following values: high=2.0, medium = 1.0, and low = 0.5. The granularity, $G_{disharmony}$, is either of the following values: method=1.0 or class=3.0. Finally, the severity, $S_{instance}$, is based on how much a disharmony violates a given metrics threshold. The impact score of a given instance of a disharmony is calculated using the following formula [49]:

$$IS_{instance} = I_{disharmony} * G_{disharmony} * S_{instance} \qquad (9)$$

Once the impact score is computed the overall debt symptoms index (DSI) can be evaluated using the following equation [49]:

$$DSI = \frac{\sum_{all\ instances} IS_{instance}}{KLOC}$$

Where KLOC is the number of thousands of lines of code for the software system under consideration. Marinescu indicates that this index value acts as a surrogate measure for the technical debt level of a software system.

### 3.2.2 Quality Estimation

We used the QMOOD [7] quality model to evaluate the quality of each project. The QMOOD quality model is based on the ISO 9126 specification [35] and uses of a combination of design metrics to indicate changes in system quality. Each of the QMOOD quality aspects is measured using a combination of metrics as identified in [7] (see table 3.3). The model is composed of the following six quality attributes: *reusability, understandability, flexibility, effectiveness, functionality,* and *extendibility.* The calculation of each of the quality attributes from the metrics listed in table 3.3 is provided in table 3.4. In order to measure these metrics, we used the tool Understand™. The QMOOD quality aspects and their relationships are provided in table 3.4.

Table 3.3 QMOOD [7] metric measurements using Understand.

| Metric | Measurement |
|---|---|
| Design Size (DSC) | Total number of classes defined in a system. |
| Hierarchies (NOH) | The count of the number of classes in a system where *MaxInheritanceTree(class) = 0*. |
| Abstraction (ANA) | The maximum length of a given class inheritance tree. |
| Encapsulation (DAM) | Ratio of the number of private and protected variables declared in a class to the number of total variables declared in the class. |
| Coupling (DCC) | A count of the number of couplings a class has. |
| Cohesion (CAM) | $\frac{100 - PercentLackOfCohesion}{100}$ |
| Composition (MOA) | A count of the number of attributes which whose type is an object defined within the scope of the system. |
| Inheritance (MFA) | A count of the methods inherited by the class divided by the total number of methods available to the classs. |
| Polymorphism (NOP) | All methods excluding final, static, and private methods. |
| Messaging (CIS) | A count of the number of public methods declared in a class. |
| Complexity (NOM) | A count of all methods declared in a class. |

Table 3.4 QMOOD quality attribute equations [49].

| Quality Attribute | Calculation |
|---|---|
| *Reusability* | -0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size |
| *Flexibility* | 0.25 * Encapsulation – 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism |
| *Understand-ability* | -0.33 * Abstraction + 0.33 * Encapsulation – 0.33 * Coupling + 0.33 * Cohesion – 0.33 * Polymorphism – 0.33 * Complexity – 0.33 * Design Size |
| *Functionality* | 0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * Design Size + 0.22 * Hierarchies |
| *Extendibility* | 0.5 * Abstraction – 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism |
| *Effectiveness* | 0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism |

## 3.3 Summary of Results

In terms of presentation of the results, the label ‘TDE1’ refers to the SonarQube TD-plugin estimation technique, ‘TDE2a’ refers to the first estimation configuration described in table 3.2, ‘TDE2b’ refers to the second one, and ‘TDE2c’ refers to the third one. Finally, ‘TDE3’ refers to Marinescu’s method of TD estimation.

We calculated Kendall’s Tau correlation coefficient between each TD estimation technique and each quality attribute value. These are shown in table 3.5 and figure 3.1. We tested for correlation between each paired sample using $p < 0.05$ as a significance level, and significant correlations are shown in bold in table 3.5. The associated scatterplots for the correlations are displayed in figure 3.1. Figure 3.1 can be read by finding the pair of variables along the diagonal finding either the scatterplot (below the

diagonal) or correlation value (above the diagonal) where the rows and columns of the variables intersect.

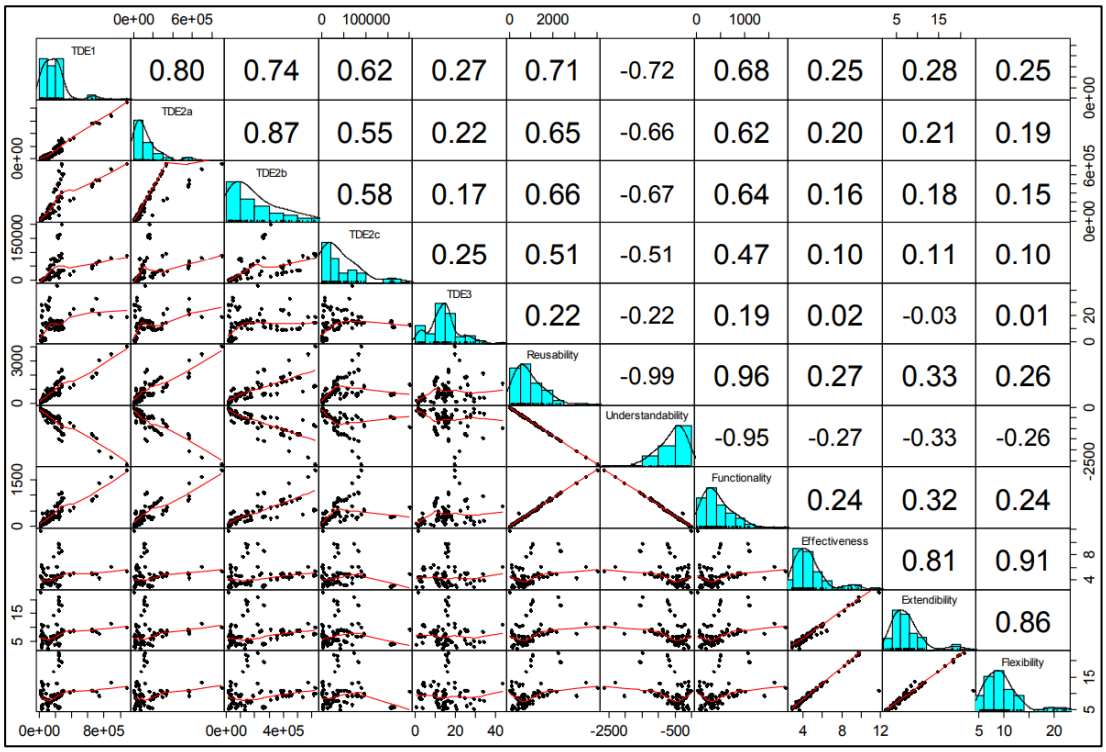


Figure 3.1 Scatterplot and correlation matrix for TDE1, TDE2a, TDE2b, TDE2c, TDE3, Reusability, Understandability, Functionality, Effectiveness, Extendibility, and Flexibility

As can be seen in table 3.5, in all cases TDE3 shows weak correlation (< 0.45) (or no significant correlation) to each of the quality attributes. For reusability, understandability, and functionality there is moderate to strong correlation to TDE1, TDE2a, TDE2b, and TDE2c as shown in table 3.5. Although these results are somewhat promising, they do not take into account the differences in size between the different systems nor the changes in size between releases of a system. To alleviate this threat, we also developed a multiple linear regression model which compensates for these issues.

Table 3.5 Correlation between TD estimates and quality attributes

| | Technical Debt Estimates | | | | |
|---|---|---|---|---|---|
| **Quality** | *TDE1* | *TDE2a* | *TDE2b* | *TDE2c* | *TDE3* |
| *Reusa-bility* | 0.7146 | 0.6483 | 0.6636 | 0.5059 | 0.2206 |
| *Flexib-ility* | 0.2538 | 0.1908 | 0.1481 | **0.0955** | **0.012** |
| *Under-stand ability* | -0.715 | -0.658 | -0.673 | -0.506 | -0.219 |
| *Effect-iveness* | 0.2522 | 0.1964 | 0.1554 | **0.0988** | **0.0156** |
| *Function -ality* | 0.6846 | 0.6175 | 0.6363 | 0.4748 | 0.1948 |
| *Extend-ibility* | 0.2805 | 0.211 | 0.1805 | 0.1105 | **-0.03** |

The significance of the results from the multiple linear regression analysis are displayed in table 3.6. For each of the technical debt estimation approaches we found that there was little to no evidence suggesting that the selected technical debt estimates have a relationship to reusability, understandability, functionality, and extendibility (as defined by the QMOOD quality model), while controlling for LOC and the number of releases in systems. With the exclusion of one TD estimation technique (TDE2c), each of the remaining technical debt estimation techniques show little to no evidence of a relationship to flexibility or effectiveness.

Table 3.6 Indication of a relationship between each of the technical debt estimates and each of the QMOOD quality attributes. An X indicates no relationship and a check indicates a relationship

| | Technical Debt Estimate | | | | |
|---|---|---|---|---|---|
| **Quality** | *TDE1* | *TDE2a* | *TDE2b* | *TDE2c* | *TDE3* |
| *Reusability* | ✗ | ✗ | ✗ | ✗ | ✗ |
| *Flexibility* | ✗ | ✗ | ✗ | ✓ | ✗ |
| *Understandability* | ✗ | ✗ | ✗ | ✗ | ✗ |
| *Effectiveness* | ✗ | ✗ | ✗ | ✓ | ✗ |
| *Functionality* | ✗ | ✗ | ✗ | ✗ | ✗ |
| *Extendibility* | ✗ | ✗ | ✗ | ✗ | ✗ |

The significance of the results from the multiple linear regression analysis are displayed in table 3.6. For each of the technical debt estimation approaches we found that there was little to no evidence suggesting that the selected technical debt estimates have a relationship to reusability, understandability, functionality, and extendibility (as defined by the QMOOD quality model), while controlling for LOC and the number of releases in systems. With the exclusion of TDE2c, each of the remaining technical debt estimation techniques show little to no evidence of a relationship to flexibility or effectiveness.

In summary, as seen in table 3.6, it appears that for all technical debt estimates excluding TDE2c they appear to have no relation to the QMOOD quality model, regardless of the correlation analysis shown in table 3.5 and figure 3.1. We have demonstrated here, there is no evidence to suggest that these estimates of technical debt reflect the expected relationship to quality.

## 3.4 Conclusions

This motivational study investigated the level of agreement from 5 methods of estimating technical debt principal to an external quality model. To address agreement, we conducted both a correlation analysis and a regression analysis. The results of this analysis showed that with the exception of one estimation method (for flexibility and effectiveness) there was no observable relationship between the quality attributes and the technical debt estimates. Additionally, given prior research showing that technical debt impacts both reusability and understandability of a software system, we found that for these quality attributes none of the technical debt principal estimates showed any relationship when taking size into consideration. These results illuminate a clear gap in the state-of-the-art, specifically that modern quality and TD estimation techniques do not provide results that align with one another.

# CHAPTER FOUR

# INTERLUDE

A synthesis of the results from our studies in Chapters 2 and 3 reveals several integral findings which provide motivation for the remainder of this work and the greater contributions to the field. The findings presented in Chapter 2, specifically that the file pairs we identified as having a large number of modularity violations also contributed to a large number of bugs in future releases, validated the intuition of the developers where the study was conducted (i.e., Golden Helix). The lead developer stated that the findings of the research aligned with his intuition of the 'problem areas' within the code-base. From an empirical perspective, our results provide the data to back the lead developer's intuition, which is a valuable result. Developers at Golden Helix had already begun refactoring the problem areas of the code-base prior to our study, yet their decision to do so was based entirely on their intuition. Our methods and tools provided validation for Golden Helix developers, suggesting that their decision to refactor was a good decision, as well as provided construct validity for our work as QA researchers.

The findings from the work in Chapter 3 revealed a potential flaw with the state-of-the-art methods and tools used to measure quality and TD in a system. Specifically, there was no relationship between quality attributes and TD estimators (with the exception of one estimation method for flexibility and effectiveness) after accounting for system size. In and of itself, this finding is surprising, as one would expect some tools to agree that TD does affect quality. In the case of our case study however, this

disagreement poses a problem because each of these tools and methods claim to provide meaningful estimates of software quality to stakeholders, yet there is no agreement on what they consider to be important quality features. Practitioners expect methods and tools that indicate areas of concerns and thus allow for the improvement of software quality. To support practitioners, tools should provide results that agree given similar systems.

A synthesis of these results reveals two research gaps. First, the form of analysis presented in both studies is restricted to static analysis, which only considers the structural aspects of a code-base. The structure of a system includes the classes, class members (variables, functions, etc.), and relationships between classes. While structural analysis provides an important perspective into a software system, it does not capture every aspect of best practice violations. Complementary to structural analysis techniques is behavioral analysis, which entails identifying violations of best practices due to unexpected runtime behaviors. This can be achieved either by normal runtime behavior or simulation of cases that execute chunks of the code-base, similar to Unit Testing and Integration Testing methods from the software testing domain. The key difference between the behavioral analysis presented in this research and Unit Testing is that our goal is to identify violations of best practices within program behaviors, while Unit Testing seeks to identify correct program behaviors and outcomes. A second research gap revealed by our studies presented in Chapters 2 and 3, is the lack of domain-specific parameters in the models. Domain-specific parameters refers to terms within a model that capture variables pertaining to the domain space, which grant the ability to configure the

model to allow for different generation and/or interpretation of results. The work with CLIO (c.f. Chapter 2) found that simple intuition instigated changes in the process, and our methods and tools validated those decisions. In Chapter 3 we identified that the out-of-the-box implementations of quality and TD analysis tools did not produce results that aligned with one another. While a user of these tools can technically configure them to modify the results, as practitioners do in domain-specific contexts, the tools provide no indication of an empirical process of calibration. Currently, selecting quality attribute weight values is entirely arbitrary, based on perceived importance of each quality attribute. This is not an empirical and data-driven way of approaching the QA process, and requires improvement.

Ultimately, our goal as software quality assurance researchers is to provide software stakeholders with better methods and tools to measure and monitor the quality of their products. Providing behavioral analysis capabilities will help with this goal, because it provides a new perspective that will complement existing structural approaches. Furthermore, the introduction of domain-specific parameters into these models will proactively supply stakeholders with configurable solutions that cater towards their quality concerns. These two points motivate the remainder of this research.

We frame our research into behavioral analysis by considering design pattern evolution. This decision is made because design patterns are considered micro-architectures of good design and have well-known expectations of structure and behavior. The work presented in this document encompasses design pattern evolution, yet the methods and tools can be generalized to consider any setting.

# CHAPTER FIVE

# A RESEARCH PLAN TO CHARACTERIZE, EVALUATE, AND PREDICT THE IMPACTS OF BEHAVIORAL DECAY IN DESIGN PATTERNS[12]

## 5.0 Abstract

We propose a research plan to further the understanding of design pattern evolution. Current research into design pattern evolution focuses on the structural elements of decay, which is realized as *structural grime*. We plan to expand the current state of research by introducing the notion of *behavioral grime*, or unwanted artifacts that appear at run-time in a pattern. This form of grime may be transparent to the current analysis models. We seek to classify types of grime into taxonomy, evaluate each type in terms of impacts on technical debt and quality in the pattern and system as a whole, and predict future occurrences of behavioral grime. Studies are designed for each of these respective goals. The results of this research will further the understanding of design patterns, assisting practitioners and researchers alike.

## 5.1 Introduction

Design patterns embody recurring solutions to common object-oriented problems in software development. Patterns are design decisions that are reusable, maintainable,

[12] Based on:
Reimanis D., Izurieta C., "A Research Plan to Characterize, Evaluate, and Predict the Impacts of Behavioral Decay in Design Patterns," IEEE ACM IDoESE, 13th International Doctoral Symposium on Empirical Software Engineering, Beijing, China, October 19, 2015.

and attempt to minimize re-design in the future [31]. However, the evolution of design patterns is controversial. The original intent of the pattern may become obscured for many reasons, including new developers contributing to a pattern, or the unforeseen changes to elements participating in the pattern. Empirical work has shown that the structure of a pattern has the potential to decay as the pattern ages [33] [34] [37] [38] [39]. Furthermore, research has shown that the structural decay of patterns results in decreased system quality and increased technical debt [22].

Although significant work has been made towards understanding design pattern structural decay, little work has been made towards understanding behavioral decay. Behavioral decay refers to the deterioration of the runtime design of a system. Behavioral decay is complementary to structural decay, yet a large gap and dearth of research is evident. The exploration of behavioral decay in design patterns will yield greater insights into the benefits and detriments of utilizing design patterns.

This chapter is organized as follows: Section 5.2 discusses related work. Section 5.3 outlines the current challenges in the field, including research gaps and relevant problems. Section 5.4 outlines research objectives. Section 5.5 describes the approach. Section 5.6 identifies the threats to the validity of the proposed study, and section 5.7 provides concluding remarks.

## 5.2 Background and Related Work

### 4.2.1 Technical Debt

Technical debt (TD) is a metaphor coined by Ward Cunningham to describe the gap between the current state of a software system and the ideal state [19]. TD captures the effects of decisions that sacrifice good design principles for on-time delivery. Many times these decisions take the form of shortcuts or workarounds in code that complete the task at hand, but at the expense of decreased quality. *Principal* and *interest* are two attributes of TD. Given a task to implement, principal refers to the cost in effort to complete the task. Interest refers to the gap between maintenance costs under ideal conditions versus conditions where maintenance is higher due to accrued debt from tasks where TD is not repaid. Effectively managing TD is multi-faceted problem, where the need to implement new features must be leveraged with the need to refactor.

Tom et al. performed a systematic literature review of the current state of TD in academic literature [74]. The study reports that many of the difficulties of managing TD are a result of poor problem definition and representative models. As an outcome of this study, Tom et al. propose a fundamental framework of TD; this work follows this framework.

Tom et al.'s framework identifies architectural technical debt (ATD) as a specific type of TD that focuses on items originating from the design or architecture of a software project. These are items such as modularity violations [79], architecture dependency issues [66], and design pattern decay [11] [33] [34] [37] [38] [39]. Several operational models for estimating TD have recently surfaced in the field [20] [32] [46] [49] [54],

however no single method has surfaced as a clear better approach, possibly because they fail to capture domain specific information in a system.

### 5.2.2 Software Quality

Software quality has been categorized into a set of characteristics, each of which is composed of related sub-characteristics. The ISO-IEC 25010 Software Quality Specification formalizes a set of eight characteristics to form an abstract model for measuring quality [36]. These characteristics, or attributes, are evaluated to the extent to which a system realizes that characteristic. Several domain-agnostic quality models that realize this specification have been developed. Two quality models, QMOOD and a robust alternative QUAMOCO, have surfaced as operational quality models [7] [76].

### 5.2.3 Software Behavior

Preliminary research reveals that software behavior can be of two types; *internal* and *external*. Internal behavior refers to the interior mechanisms and API calls that occur during system runtime. Internal behaviors are not necessarily seen except at the point in time in which they are executing. In this manner, internal behaviors are more a temporary artifact that exists only for the duration of their execution. External behavior refers to the external and observable result that the system produces. These may be represented as system goals and are the consequences of internal behaviors. That is, internal behaviors cause external behaviors.

5.2.4 Software Decay

Code decay is a term that refers to the case where code is “harder to change than it should be” [24]. Similarly, software decay refers to software that is more difficult to change than it should. Several types of software decay have been identified, including code smells, anti-patterns, and design pattern decay [11] [28] [37] [38] [39]. Design pattern decay refers to implementations of design patterns that gain undesired elements or lose desired elements as they evolve. In this sense, the benefits that the pattern offers are lost as its design becomes obfuscated. Studies have found that design pattern decay negatively impacts testability and understandability of systems [11] [37].

Previous work in design pattern decay has focused on the structure of patterns [22] [33] [34] [37] [38] [39]. These are realized as unwanted or missing artifacts that do not follow the structural specification of the pattern. When these artifacts obscure the implementation of a pattern while still maintaining some of the integrity of the original pattern, they are referred to as *design pattern grime*. Alternatively, when these artifacts obscure an implementation of a pattern to such an extent that the integrity of the pattern is entirely lost, they are referred to as *design pattern rot*. Empirical studies have only confirmed the existence of pattern grime.

Further work has classified the types of design pattern grime into three disjoint categories: *class grime, modular grime,* and *organizational grime* [34] [37] [38] [39]. Of these, Schanz and Izurieta expanded the modular grime category, identifying *strength, scope,* and *direction* as attributes of modular grime [67]. Additionally, Griffith and

Izurieta expanded the class grime category, identifying *strength, scope,* and *direction/context* as attributes of class grime [34].

5.2.4.1 Design Pattern Specification. The process of identifying pattern grime consists of recognizing differences between a pattern instance and a pattern's specification. A common language used to specify patterns is the Role-Based Meta-Modeling Language (RBML) [43]. RBML is realized in the Unified Modeling Language (UML 2.0)[13] and is an abstract language that generalizes each actor in a pattern to a single common role. Depending on the type of pattern, there will be a number of possible roles. For example, the Observer pattern has a *Subject* role and an *Observer* role. Observer pattern *instances have classes that fulfill both these roles.*

Dae-Kyoo Kim has shown that RBML alone is not sufficient for specifying patterns because it lacks constraint templates that limit the capabilities of roles [42]. In order to combat this, the Object-Constraint Language (OCL) is used to provide necessary constraints to RBML models.

## 5.3 Current Research Challenges

### 5.3.1 Research Gaps

The current knowledge base of design pattern grime features only structure-based disconformities, or grime that is captured from a static snapshot of a pattern instance. This works seeks to extend the knowledge base of pattern grime by considering behavior-based disconformities, or grime that is captured during the runtime execution of a design

[13] http://www.uml.org/

pattern. In an effort to achieve this goal, the authors have identified the following research gaps.

1. **Characterization of Behavioral Grime**: Structural grime is incapable of capturing whether or not a design pattern is behaving as intended. A pattern instance may have no structural grime, but the runtime execution of the pattern may not match the expected runtime execution of the pattern. Cases such as this are not captured by the current knowledge base of pattern grime. This notion forms the basis for this research. Given this, the characterization of behavioral grime is a gap that needs clear definitions.
2. **Behavioral Grime Taxonomy**: To the best knowledge of the authors, no attempt has been made at categorizing the types of behavioral grime in the context of design patterns.
3. **Impacts on Quality**: Previous studies have identified the impact of structure-based grime on quality attributes, showing that testability and maintainability are negatively impacted from structural grime [34] [39]. However, no attempt has been made at quantifying the impact of behavioral grime on these quality attributes and the additional quality attributes featured in the ISO 25010 software quality specification.
4. **Impacts on Technical Debt**: Dale and Izurieta showed that the injection of modular grime into patterns increases the technical debt of the pattern [22]. No work has sought to capture the impact of behavioral grime on technical debt.

5. **Relationships between Behavioral and Structural Grime**: Several questions arise that are concerned with the relationships between behavioral and structural grime. For example: How are structural grime and behavioral grime related? Is the appearance of structural grime causal to the existence of behavioral grime? Is the reverse true? Are there cases where structural grime exists but behavioral grime does not?
6. **Tool Support**: Currently, there is no known tool support to operationalize behavioral concepts. Implementing a tool is an important contribution to the community.
7. **Predicting Pattern Decay:** No research has looked into predicting when a pattern is prone to decaying, or even if certain patterns are more prone to decay. Bridges to these two research gaps would give valuable insight to developers regarding the implementation of patterns, and even when to be aware that a pattern might be near decaying/rotting.

### 5.3.2 Operational Gaps

A pilot study was performed, in the form of a controlled experiment; in which realizations of observer patterns were studied. We created three instances of the observer pattern; one instance behaved as defined, one instance featured *Subjects* that waited a significant amount of time before updating their *Observers* when their state changed, and the final instance featured *Subjects* that did NOT update their *Observers* when their state changed. These three instances exemplify cases where, respectively, (1) a pattern behaves properly, (2) a pattern behaves properly but a disharmony exists during its lifetime, and

(3) a pattern behaves significantly different from its intended usage. The SonarQube [32] tool, used to estimate Technical Debt, and the inCode tool[14], used to identify design flaws, were run across the pattern instances. Neither of these tools identified a major difference between the three pattern instances, suggesting that state-of-the-art tools used to identify issues are not capable of detecting problems concerning design pattern behavior. This experiment highlights the need to explore this area further.

### 5.3.3 Proposed Contributions

To address current gaps, the following contributions are proposed:

1. The formal characterization of behavioral grime in design patterns
2. The development of taxonomy to classify behavioral grime
3. The development of empirical studies to capture the impacts of grime on TD and quality
4. The identification of patterns that are prone to behavioral grime
5. The creation of a tool that aids in the detection of behavioral grime
6. The development of a method that allows predictive capabilities for recognizing grime

### 5.3.4 IDoESE Feedback Sought

Advice on the following topics is sought:

1. **Overall Scope**: Whilst all topics presented in this paper are interesting and necessary research items, advice on the estimation of work and its feasibility is

[14] https://www.intooitus.com/products/incode

sought. For the scope of a doctoral-level degree, is this plan too ambitious? If so, what parts should be prioritized?

2. **Automation**: Currently, there is very little automation of these processes. This is a result of exploring a new area of research. To what extent should we focus on operationalizing behavioral detection and quantification?
3. **Pattern Dataset**: The only available dataset of design pattern instances is the Perceron's dataset [2]. This dataset only features instances of 10 unique pattern types, all from the Java programming language. This means that this research has limited generalizability. Is it necessary or worth the effort to look at more pattern types and/or patterns instances from other languages?

## 5.4 Objectives

### 5.4.1 Research Objectives

**RG1:** *Investigate* design pattern instances *for the purpose of* identifying and characterizing internal and external behavioral grime *with respect to* proper pattern behavior as defined by the design pattern specification *from the perspective of* the software system *in the context of* design patterns in open source and commercial software.

**RQ1.1:** Does the behavior of a design pattern instance deviate from the expected behavior of that pattern type?

**Rationale:** This is the basic question of this research. If it is possible to identify design pattern instances where the actual behavior deviates from expected behavior, then the need to further explore this phenomenon is apparent.

**RQ1.2:** Do common types of behavioral grime exist within multiple instances of a single pattern type?

**Rationale:** If common grime types can be identified within a specific pattern, other instances of that pattern may be circumspect to the same type of grime.

**RQ1.3:** Do common types of behavioral grime exist across multiple instances of different pattern types?

**Rationale:** If common types of behavioral grime exist across different types of patterns, we will have attained some level of generalizability that applies to a larger set of pattern types.

**RG2:** *Express* the difference between structural and behavioral grime *for the purpose of* illustrating the importance of studying behavioral grime *with respect to* design pattern instances *from the perspective of* design pattern instances *in the context of* open source and commercial software.

**RQ2.1:** To what extent can patterns have both structural and behavioral grime?

**Rationale:** Consider the grime quadrant in table 5.1. Columns indicate whether structural grime exists in a pattern, and rows indicate whether behavioral grime exists in the same pattern. Current research has identified design patterns with grime, but those patterns are constrained by cases A and B. This research needs to be expanded to discover patterns that fall in cases C and D. This will illustrate that this work is novel.

Table 5.1 Grime quadrant of possible grime types. For a given pattern, rows correspond to at least once instance of behavioral grime existing in the pattern, and columns correspond to at least one case of structural grime existing in the pattern.

| | Structural grime does not exist | Structural grime exists |
|---|---|---|
| Behavioral grime does not exist | Case A | Case B |
| Behavioral grime exists | Case C | Case D |

**RQ2.2:** Does the current knowledge base of structural grime instances include cases of behavioral grime?

**Rationale:** There may be behavioral grime in many of the patterns that exhibit structural grime.

**RQ2.3:** What is the relationship between behavioral grime and structural grime?

**Rationale:** Intuitively, it appears a relationship exists between behavioral and structural grime. Discovering the precise nature of this relationship will help developers understand pattern decay in the future.

**RG3:** *Quantify* the impact of *grime* in internal and external design pattern behavior *for the purpose of* capturing the effects on system quality and TD *with respect to* proper pattern behavior as defined by the design pattern specification *from the perspective of* the software system *in the context of* design patterns in open source and commercial software.

**RQ3.1:** To what extent does behavioral grime affect the quality attributes of a design pattern?

**Rationale:** This research question seeks to quantify the impact behavioral grime has on the quality of the pattern.

**RQ3.2**: Is the quality of certain types of behavioral grime worse than other types?

**Rationale**: This question attempts to identify the forms of behavioral grime that are worse than others.

**RQ3.3:** To what extent does behavioral grime affect the TD of a software project?

**Rationale:** In essence, TD captures the financial impact of behavioral grime. Understanding this impact is crucial for developers and project managers alike so decisions regarding release timelines or refactorings can be made.

**RQ3.4**: Is the TD of certain types of behavioral grime worse than other types?

**Rationale**: This question attempts to identify the forms of behavioral grime that are worse than others.

**RQ3.5:** Are the current TD estimation and quality measurement tools capable of capturing behavioral grime?

**Rationale:** Behavioral grime may have an impact on the TD estimate and quality of the pattern. If the current tools are not sufficient in capturing these impacts, then the tools need to be extended in order to reflect the impact.

**RG4:** *Investigate* the evolution of internal and external behavior in design patterns *for the purpose of* capturing trends of behavioral grime over time *with respect to* proper pattern behavior *from the perspective of* the software system *in the context of* pattern in open source and commercial software.

**RQ4.1:** Can common trends of behavioral grime be captured as a pattern evolves?

**Rationale:** This question identifies if patterns are more prone to certain behavioral grime types. If we can predict which patterns tend towards building behavioral grime, then development efforts can be more pro-active in addressing pattern evolution.

**RQ4.2:** Can behavioral grime be predicted?

**Rationale:** This question focuses on the possibility that underlying mechanisms may exist that allow us to predict when a pattern will accumulate behavioral grime in the future.

### 5.4.2 Research Metrics

Following the GQM approach [9], several metrics are identified that will be used to answer the research questions.

M1: *Structural Grime Count (SGC)* – The total amount of grime accumulated in a single pattern realization that is identified from structural models. This metric will be used to answer RQs 2-4.

M2: *Behavioral Grime Count (BGC)* -- The total amount of grime accumulated in a single pattern realization that is identified from behavioral models. This metric will be used to answer RQs 2-4.

M3: *Technical Debt Principal (TDP)* – A measure of the cost required to complete a task. This metric will be used to answer RQ 3.

M4: *Technical Debt Interest (TDI)* – A measure of differences in cost required to complete tasks under ideal conditions versus the current condition of the system. This metric will be used to answer RQ 3.

M5: *Pattern Quality (PQ)* – An aggregated measure of the eight quality characteristics featured in the ISO 25010 software quality specification [36]. Each quality characteristic is further broken down into a number of (sub)-characteristics. This metric reflects an aggregation of the (sub)-characteristics. This metric will be used to answer RQ 3.

M6: *Probability to Deviate (PD)* – The probability that a pattern will accumulate grime in the future, given its pattern type, past and current SGC, BGC, TDP, TDI, and PQ. This metric will be used to answer RQ 4.

### 5.4.3 Working Hypotheses

**H1:** There exist instances of behavioral grime that are not captured by current structural grime models.

**H2:** Common forms of behavioral grime exist within the same pattern type.

**H3:** Common forms of behav**i**oral grime exist across different pattern types.

**H4:** Including behavioral grime in the current grime models will allow the detection of pattern rot.

**H5:** Quality and TD

**H5.1:** Behavioral grime has a negative effect on the quality of the (a) pattern realization, and (b) software system as a whole.

**H5.2:** Behavioral grime has a negative effect on the TD calculation of the (a) pattern realization, and (b) software system as a whole.

**H6:** Given the pattern type, and past and current measurements of SGC, BGC, TDP, TDI, and PQ, it is possible to predict whether a pattern will accumulate grime in the future, with a degree of uncertainty.

## 5.5 Approach

### 5.5.1 Data Collection

Design pattern instances will be collected across a variety of open source and commercial software systems. The Perceron's dataset features 4500 pattern instances from Java open source software systems [2]. The patterns featured in this database will be downloaded to provide an initial set of design pattern instances. Additionally, design patterns will be manually extracted from a commercial software system owned by a local firm with an established relationship.

Models of each design pattern instance will be captured using UML class diagrams and UML sequence diagrams[15]. Class diagrams capture the structural elements of the pattern instance, and sequence diagrams capture the behavioral elements of the pattern instance. Additionally, pattern specifications for each pattern type will be captured in UML class and sequence diagrams, using RBML and OCL.

The PQ, TDI, and TDP of each pattern instance will be calculated. These metrics will be calculated for both individual pattern instances and the entire software system that the pattern originates from. This data will be stored in a relational database.

[15] http://www.uml.org/

5.5.2 Research Approach

Once the data collection process is complete, a variety of case studies and experiments will be used to answer the research questions. Juristo and Moreno's guide on experimentation in software engineering will be used to initially construct experiments [40].

RQ1.1-3 will be evaluated using a case study, wherein the taxonomy of design pattern grime will be extended to incorporate behavioral grime types. All pattern instances will be categorized according to their behavioral and structural conformance from the grime quadrant of table 5.1. We will manually sort through each category, identifying design pattern violations. Violations that share similarities (OCL or RBML) will be grouped.

RQ2.1-3 will be evaluated using a case study. Conformance checking algorithms will be implemented that validate the structural conformance and behavioral conformance according to the work done by [42] [71]. All available pattern instances will be categorized into one of the four groups defined in table 4.1. A binomial regression model will be fitted from the sample in order to answer RQ2.3.

RQ3.1-5 will be evaluated using a controlled experiment. Patterns will be blocked according to pattern type and then randomly selected from the available dataset. Patterns will be evaluated for TD and quality using a suite of static and dynamic analysis tools, as discussed in section 4.2. After measurements are recorded, forms of grime will be randomly selected and injected into patterns. After injecting, we will re-evaluate the TD and quality measurements. To analyze the data, two ANOVA tests will be utilized.

RQ3.1-4 will be answered by fitting a two mean model, containing a mean for non-injected patterns and a mean for injected patterns. That is, the respective TD and quality measurements from all tools that analyzed non-injected patterns will be averaged. Respectively, the same analysis will be done for injected patterns. RQ3.5 will be answered by fitting a separate means model; that is, each quality analysis tool will have a mean. Variance will be measured over all the analysis tools, for each of non-injected and injected patterns.

RQ4.1-2 will be evaluated using an observational study. Patterns will be divided by pattern type and assessed for the existence of grime across their lifetime in terms of project releases. For each release, a record will exist documenting whether that pattern instance has grime or not. Further, an ARIMA analysis will be performed. This will give an indication into the tendencies of a pattern to collect grime as it ages.

## 5.6 Threats to Validity

There exist several threats to the validity of this study. Internal validity refers to the ability to recognize a causative relationship in the study, and not as a result of confounding variables. Internal validity is threatened because other design defects may exist alongside grime in a pattern; thus design defects are a confounding variable in this study. To attempt to remove the effect of design defects, we utilize a large number of pattern instances in the analysis and block across pattern type. This mitigates the chance that a design defect will affect the results of the study.

External validity refers to the ability to generalize from the results of the study. External validity is threatened because of the limited datasets of design pattern instances.

To combat this threat, we have utilized the Perceron's dataset, which is the only publically available dataset of patterns that features a large number of instances (over 4500), and pattern instances from a local commercial software firm. Patterns from both these datasets are implemented in Java, and the Perceron's dataset features only open source patterns. Therefore, the ability to generalize the results is limited to the population of patterns in this study.

## 5.7 Conclusions

We have outlined the work that will result in a doctoral dissertation in hopes that we can receive feedback on the merit of this research. Research gaps are presented and studies are designed that fill them. We intend to contribute novel research that strengthens the current state of empirical software engineering.

This research is in its early stages. Currently, preliminary research has been performed, for the purpose of illustrating the research gaps. This research includes generating pattern instances and manually injecting grime into them, as described in section 5.2. Additionally, two potential forms of behavioral grime have been identified. Next steps call for the analysis of a larger number of pattern instances that expand the taxonomy of behavioral grime.

# CHAPTER SIX

## EVALUATIONS OF BEHAVIORAL TECHNICAL DEBT IN DESIGN PATTERNS: A MULTIPLE LONGITUDINAL CASE STUDY

### Contribution of Authors and Co-Authors

Manuscript(s) in Chapter(s) 6

Author: Derek Reimanis

Contributions: Selection of research questions and design of study, the development of the methods and tools to answer the research questions, the analysis of the results including discussions, and the initial drafts of the manuscript.

Co-Author: Clemente Izurieta

Contributions: Kept motivations and scope grounded and achievable, provided substantial edits to the manuscripts, and provided valuable advice for the analysis and presentation of figures.

Manuscript Information

Derek Reimanis, Clemente Izurieta

IEEE Transactions on Software Engineering

___X_ Prepared for submission to a peer-reviewed journal
____ Officially submitted to a peer-reviewed journal
____ Accepted by a peer-reviewed journal
____ Published in a peer-reviewed journal

IEEE Computer Society

# CHAPTER SIX

# EVALUATIONS OF BEHAVIORAL TECHNICAL DEBT IN DESIGN PATTERNS: A MULTIPLE LONGITUDINAL CASE STUDY[16]

## 6.0 Abstract

Design patterns represent a means of communicating reusable solutions to common problems, provided they are implemented and maintained correctly. However, many design pattern instances erode as they age, sacrificing qualities they once provided. Identifying instances of pattern decay, or pattern grime, is valuable because it allows for proactive attempts to extend the longevity and reuse of pattern components. Apart from structural decay, design patterns can exhibit symptoms of behavioral decay. We constructed a taxonomy that characterizes these negative behaviors and designed a case study wherein we measured structural and behavioral grime, as well as pattern quality and size, across pattern evolutions pertaining to four design pattern types. We evaluated the relationships between structural and behavioral grime and found statistically significant cases of strong correlations between specific types of structural and behavioral grime. Furthermore, we identified the rates at which behavioral grime accumulates in pattern instances using multiple linear regression analysis. We extended the QATCH software quality model to incorporate design pattern grime, and measured and correlated

[16] Based on:
Reimanis D., Izurieta,C, "Behavioral Evolution of Design Patterns: Understanding Software Reuse through the Evolution of Pattern Behavior," 18th International Conference on Software Systems and Reuse, ICSR 2019. In: Peng X., Ampatzoglou A., Bhowmik T. (eds) Reuse in the Big Data Era. Vol 11602, Springer Cham. https://doi.org/10.1007/978-3-030-22888-0_6 Cincinnati, OH, June 26-28 2019.

software quality to the presence of behavioral grime in software systems. Our results suggest a strong inverse relationship between software quality and behavioral grime.

## 6.1 Introduction

Software products have evolved rapidly over the last several decades. Increasingly complex software requirements from customers have prompted advances in software development practices and automation across all disciplines. These circumstances have helped create an ecosystem where the expectations of software products is significantly higher, and where once minor upgrades were sufficient, now fully functional and highly specialized products are expected. To cope with higher expectations and complex requirements, software quality assurance is becoming a mainstream approach to meet those needs.

The deployment of complex products with multiple components does not come without its drawbacks, however. The expectation that multi-component complex systems are delivered on time and within budget, require the adoption of robust processes to accommodate all phases of the product's software life-cycle. One such process is software quality assurance (QA); which seeks to measure and monitor all aspects of software quality over the entire lifetime of a software product. In fact, traditional software testing is no longer enough, and more advanced QA methods, such as continuous integration, are necessary approaches to ensure the quality of every component at all stages of the product's life-cycle. Software design is one phase in the software life-cycle where QA techniques are necessary. Software design represents the

vision of a software solution, considering current and potential future requirements. Designs must be flexible enough to accommodate change, facilitate extensibility, and promote the ease of interchangeable and reusable software components, while still maintaining a high level of quality. One common strategy to assist with this balance is to use design patterns.

Design patterns embody recurring and reusable solutions to common problems encountered in the software development process [31]. Design patterns capture experience reuse and represent decisions that are made in the design phases of software life-cycles. They have the properties of being reusable, maintainable, and easy to extend in future versions. The choice to utilize design patterns in a project comes with the understanding of an important assumption– specifically, that the initial implementation of a pattern instance may take longer than a non-pattern implementation, but future revisions and maintenance efforts will be faster and therefore cheaper if a pattern is present. This assumption holds true in a theoretical sense, yet is controversial in a practical sense. Historically, design pattern realizations have been found to deviate or drift from their initial intent, thus eliminating many of the beneficial qualities the pattern offers in the first place. Such a deviation may occur if a new developer is unfamiliar with a code-base, or if pressure from management to ship a product requires ’quick-and-dirty’ extensions of the pattern. Such a phenomenon is referred to as technical debt (TD) [4], and the existence and extent of TD are not fully explored; for example, it is not known whether the presence of such a deviation within a design pattern provides more harm to a software product than choosing not to utilize a design pattern in the first place.

### 6.1.1 Research Problem

Previous research efforts have both explored the existence and measured the effects of design pattern deviations from only a structural perspective. The structural perspective of a design pattern refers to the class members of the pattern, including the operations and attributes of the pattern's classes, as well as the relationships between class members. This research has found that such deviations do exist within a design pattern's evolution, and that these deviations have a negative effect on software quality. However, the structural perspective is one of many perspectives into a design pattern. Another perspective necessary to understand design patterns is the behavioral perspective, or the events that occur as a design pattern instance is operating at program run-time, which are not visible from a structural perspective. A behavioral perspective offers additional insights into a design pattern and its evolution, thus refining existing scientific models and taxonomies [67] [34] [83] that capture design pattern evolution.

### 6.1.2 Research Objective

The goal of this research is to expand the body of knowledge surrounding software quality and technical debt, as it pertains to design pattern evolution, from a behavioral perspective. Four specific activities aligned with our overarching goal are identified. First, the identification of design pattern deviations from a behavioral perspective. Second, the characterization of behavioral deviations into a structured organizational scheme, a taxonomy. Third, the comparison of behavioral grime to current grime models, specifically structural grime. Fourth, the evaluation of the effects that

behavioral deviations have on software quality and technical debt. Meeting these objectives complements structural approaches, and provides software stakeholders with more advanced techniques and tools to monitor software quality, so that important decisions pertaining to software products can be made with increased certainty.

### 6.1.3 Contributions

The contributions of this work include:

- A taxonomy that captures behavioral grime in design pattern instances.
- Evaluation of the relationships between structural grime and behavioral grime.
- Analyses illustrating the rate at which behavioral grime accumulates in a pattern instance.
- Extension of the QATCH [70] quality model to include design pattern evolution quality properties.
- Evaluations of the relationships between behavioral grime and software quality.

## 6.2 Background and Related Work

In the following subsections we discuss relevant background and research, which can be broadly labeled as software quality assurance. We also provide definitions for key terms, and follow by detailing the process we employed to identify important research topics aligned with our goal.

6.2.1 Software Quality

Software product quality, hereafter referred to as software quality, is broadly defined by the ISO-25010 specification as the *degree to which a software product satisfies its various needs* [36]. This general definition can be applied to any and all software products. To aid in the operationalization of such an abstract concept, a hierarchy of software quality characteristics and attributes are provided by the specification. At the second-highest level in the hierarchy, software quality is divided into eight characteristics, which themselves consist of multiple sub-characteristics or properties. An illustration of the software product quality model is presented in figure 6.1. Largely, every quality sub-characteristic or quality property is defined as *the degree to which a software product satisfies* it.

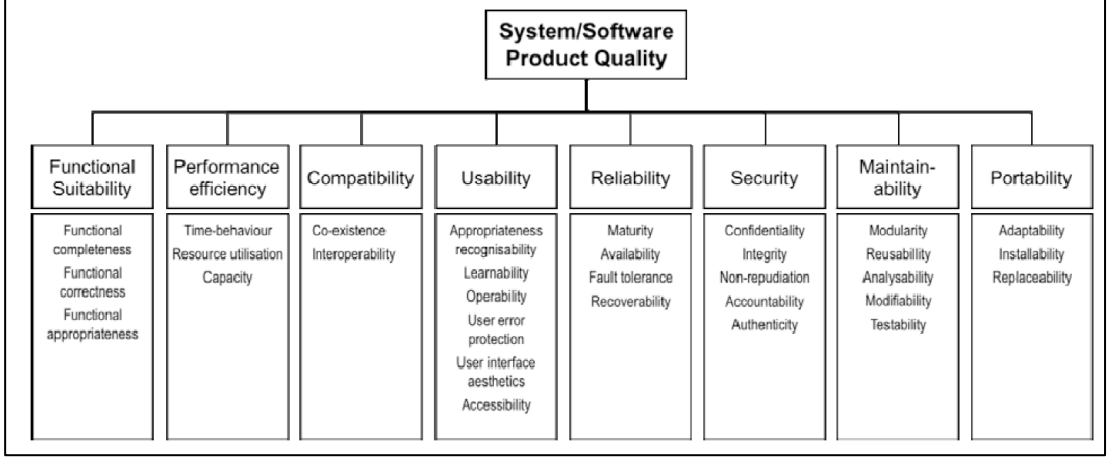


Figure 6.1 The ISO-25010 software product quality model [36]. Software product quality is divided into eight primary quality characteristics, which each have multiple quality sub-characteristics or properties themselves.

Because of its abstract nature, the ISO-25010 quality model can be viewed more as an academic, or research tool, not a practitioner's tool. This conflicts with the goal of the quality model, because its purpose is to facilitate operationalization, and ideally help

quantify or qualify software products with the intention that practitioners use the results to better understand their software. As the model is defined, it does not immediately provide practitioners with guidance for measuring each of the characteristics. Fortunately, several research projects have taken steps to improve the model's usefulness by operationalizing the model with concrete measurements of each quality characteristic or property, at the code level. Such operational models perform static analysis on a product's code-base, identifying violations of coding best practices or suggestions for improvements (such as insufficient code commenting). Following the code-level analysis, all findings are aggregated and mapped to corollary characteristics or properties within the quality model. This mapping effectively connects the abstract levels of the quality model with its operationalized counterparts, thereby providing practitioners with meaningful and actionable methods to measure software quality.

### 6.2.2 Technical Debt

TD is a metaphor coined by Ward Cunningham in the seminal Wycash Portfolio Management System report [19]. TD captures the trade-offs between spending time to follow good design and development practices versus rushing a product to market to secure a market niche before competitors. If more time is spent on product quality, the product may never be released in a timely manner. Yet if more time is spent on shipping a product to market, the quality of the underlying design and code may suffer, meaning future changes may be more difficult to make. Contrary to intuition, TD is not a minimization or maximization problem, but rather a portfolio management problem. With the understanding that it is very difficult or, in many cases impossible, to predict the

future direction of a product, it is preferable to provide stakeholders with a TD report and allow them to make an informed decision regarding the state of TD in their product. Some scenarios may encourage TD remediation efforts, while some scenarios may encourage the push to production. Additionally, the domain of the company, and subsequent products, may have an effect on the decision to re-mediate TD items. Recent work from a Dagstuhl[17] seminar narrowed the scope of the TD field to consider only internal code issues, stemming from the quality characteristic of Maintainability [4]. This narrowing of scope was necessary as a means to identify the search space of TD, in order to effectively measure TD in a software system.

### 6.2.3 Design Pattern Formalization

Design patterns can be formally specified using a combination of the Role-Based Meta-Modeling Language (RBML) [42] and the Object Constraint Language (OCL) [77]. RBML specializes the Unified Modeling Language (UML) [65] meta-model and captures key elements of a design pattern, based on specific roles that participants in that design pattern may take. A design pattern specification consists of two sub-specifications, the Structural Pattern Specification (SPS) and the Interaction Pattern Specification (IPS) [42]. An SPS characterizes the structural elements of a pattern, including the class members, attributes, operation signatures, and relationships. An IPS characterizes the behavioral elements of a pattern, referring to the flow of information that occurs when a design pattern is in operation, at program run-time. SPSs are analogous to UML class diagrams, whereas IPSs are analogous to UML sequence diagrams at the M2 level of

[17] https://www.dagstuhl.de/en/

design specification. For example, consider the Observer pattern instance illustrated in figure 6.2. In this example inspired by [29], we consider an Observer pattern implementation that controls the operation of a kiln system. Two kiln classes, CeramicKiln and SteelKiln are Subjects in this example, and each kiln is monitored by two Observers, an LEDObserver and a RemoteObserver. The UML class diagram of this system is shown on the left half of the figure, and the corresponding UML sequence diagram of the system is shown on the right half. Both of these diagrams are depicted as M1 level specifications. The diagram also shows the respective M2 level SPS (top-left) and IPS (top-right) specifications of the Observer pattern. Dotted lines capture mappings from individual elements of the kiln system to the corresponding design pattern role characterized by the specification. The arrows represent conformance to the intended design of the pattern. To improve clarity, individual mappings for the operations and attributes are not shown, yet they are considered in the actual mapping process. This example is naive in the sense that this kiln system nearly perfectly aligns to the Observer pattern SPS and IPS; such close alignments are unlikely in practical systems. However, the example serves as a visual representation of mappings from design pattern instance to design pattern specification.

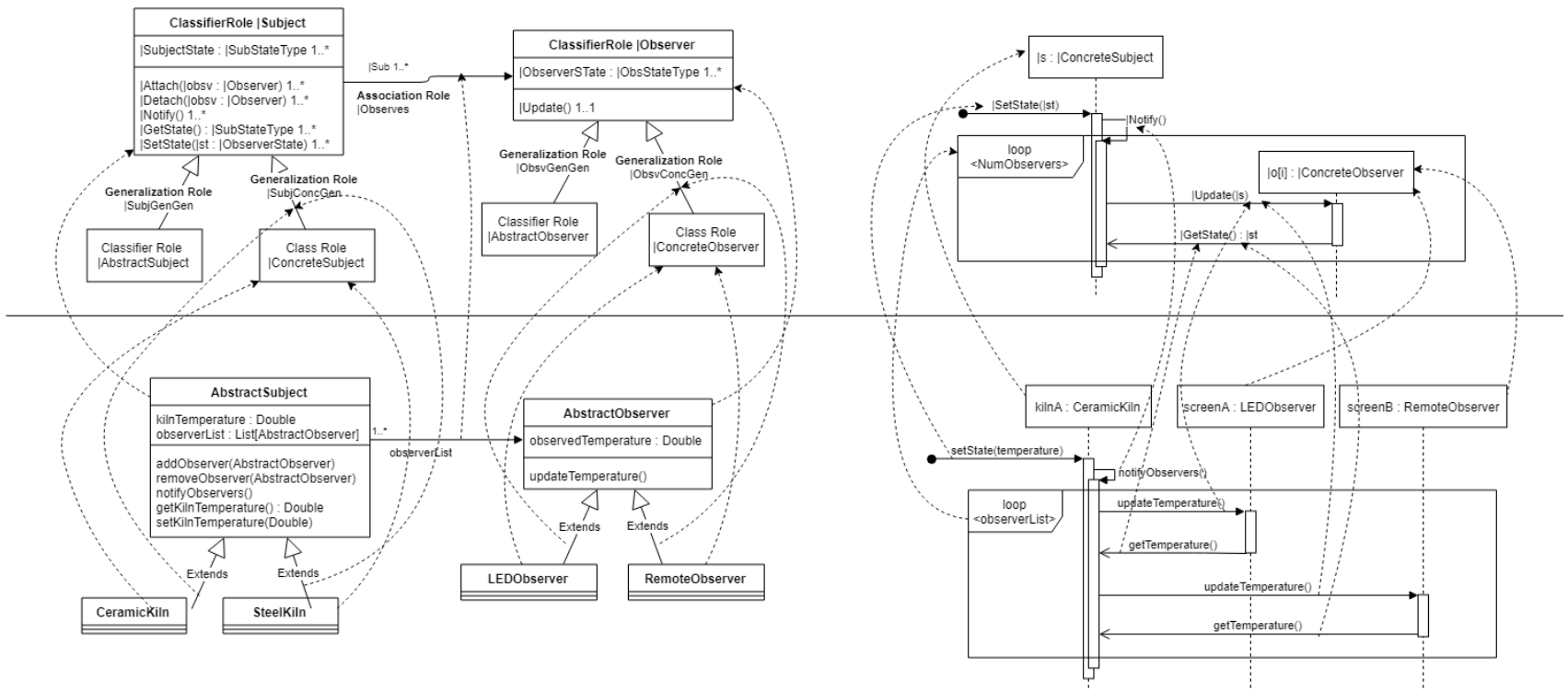


Figure 6.2 Example Observer pattern instance of a kiln system. The class and sequence diagrams for the kiln system are shown on the bottom half, and the SPS/IPS for the Observer pattern are shown, using dotted arrows, from concrete instance to SPS (on the left), and to IPS (on the right)

### 6.2.4 Design Pattern Decay

Software applications are used every day, yet they do not 'wear out' over extended use periods in the classical sense, as physical objects would. Instead, software is subject to a different type of wear, related to the maintenance of the underlying design and code. Over time, many factors such as unforeseen changing requirements, developer turnover, legacy code dependencies, and others, will contribute to the degradation of software quality. This phenomenon is captured by the terms *software decay* and *code decay*. Software and code are deemed *decayed* if they are *harder to change than they should be* [24]. A specific form of software decay is *design pattern decay*. Design pattern decay refers to the addition of undesired elements or loss of desired elements in a design pattern instance, over the lifetime of the design pattern [37] [38]. Design pattern decay is considered a sub-domain of design decay, which is analogous to code decay with the exception that the decay occurs in the design level of a software project instead of at the code level. Design pattern decay consists of two categories; *design pattern grime* and *design pattern rot* [38]. Design pattern grime, hereafter referred to as *grime*, is defined as the build-up of unintended artifacts, or elements, over the lifetime of a design pattern instance. These artifacts do not contribute to the pattern's intended role in the overall software project, detracting from the beneficial qualities the pattern would otherwise provide. A key distinction exists between grime and elements that are necessary for the implementation of the design pattern in the system; specifically that grime considers the evolution of the design pattern instance, illustrating elements that appear over time that do not reflect an initial and clean version of the design.

Previous work has shown that the presence of grime is associated with decreases in testability and adaptability, as well as the presence of anti-patterns [39]. Additionally, recent work has shown that the presence of grime is related to the depreciation of system correctness, system performance, and system security [26]. Furthermore, Feitosa et al. has found that grime has a tendency to accumulate linearly, suggesting the quality of a pattern worsens as the grime of that pattern increases [25]. Design pattern rot, hereafter referred to as *rot*, is defined as the removal of key elements of the pattern such that the pattern no longer retains its core elements. A pattern that has succumbed to rot no longer identifies as such; instances of rot in software projects has eluded researchers because of the difficulty in identifying it. From a formal perspective, a pattern has succumbed to rot when it no longer conforms to, or successfully maps to, a pattern's SPS or IPS. The degree to which a pattern instance conforms to its intended design is a research topic that has not been explored.

### 6.2.5 Literature Review

In an effort to identify important and relevant research topics, we utilized a Systematic Mapping Study (SMS) as outlined by Peterson et al. [61]. A SMS seeks to provide an organized overview of a research area, categorizing the quantity and type of research performed by various research groups. Within our SMS, we employed Budgen's protocol for identifying research 'gaps' as well as 'clusters' using mapping studies [14]. Gaps present research areas that have little exploration, where new or improved primary studies are required. Alternatively, clusters indicate areas that already have been explored, where more complete Systematic Literature Reviews (SLRs) may be

undertaken. After finding several clusters in software quality assurance, with respect to software quality and TD, we performed a formal literature review with a focus on identifying research goals with respect to evident gaps. The following section details this process by introducing research accomplishments in software quality assurance with focus on software maintainability and technical debt. Along with these definitions of software quality and technical debt, we briefly describe several state-of-the-art operational research-based quality assurance tools and discuss their impact on open source and commercial software projects. Following, we draw logical connections across research groups as well as operational tools to highlight gaps in the research area.

Table 6.1 presents the results from our SMS using Budgen's formatting [14]. We have elected to remove the column titled 'Period Searched' because the scope of focus for this study includes all publications from the listed author(s) until the current date. Synthesizing the results, we identify an immediate research gap in behavioral TD analysis. Every research cluster presented in the table utilizes some form of structural analysis to perform their research, yet none has considered the behavioral aspects, or the intricacies of how software quality and TD are affected by the run time attributes and properties of code. More specifically, behavioral aspects in the context of design patterns, of which we have quite formal specifications for understand that deviations away from the formal specification is a form of TD [82]. This clear gap provides the need and basis for the extent of this study.

Table 6.1 Results from our Systematic Mapping Study, following the format of Budgen [14].

| Author(s) | Years Published | Topic | No. of Studies | Form of Research | Sources Searched |
|---|---|---|---|---|---|
| Seaman, Shull, Guo | 2006-2018 | TD Management | 26 | TD Management; TD identification and decision-making frameworks | Google Scholar & dblp database |
| Cai, Wong, Kazman, Xiao | 2007-2016 | Architectural TD | 24 | Version Control/Ticket System Analysis & Tool development & Case Studies | Google Scholar & dblp database & ACM library |
| Morisio, Vetro, Torchiano | 2004-2017 | Automatic Static Analysis Issues | 10 | Model and Evaluation | Google Scholar & dblp database & ACM library |
| Fontana, Zanoni, Roveda | 2011-2018 | Code Smells | 19 | Code smell identification, Code smell evaluation | dblp database |
| Kim, France, Bieman | 2002-2018 | Design Pattern Formalization | 33 | Model formalization, Model checking, conformance | dblp database |
| Izurieta, Griffith, Reimanis | 2007-2016 | Design Pattern Evolution | 24 | TD Identification, TD evaluation, TD injection, TD evolution | Google Scholar & ACM library |
| Avgeriou, Ampatzoglou, Chatzigeorgiou, Feitosa | 2003-2019 | Software Architecture Evolution | 31 | TD management, TD evaluation, TD evolution | Google Scholar & dblp database |

## 6.3 Research Approach

In an effort to expand on software quality assurance, as it pertains to design pattern evolution from a behavioral perspective, the strategy employed in this research has three-steps; first, the identification and detection of unintended behavioral items, as they appear in the code of design pattern instances. Second, the characterization of unintended behavioral items into categories that simplify the remediation effort. Third, the measurement of severity of unintended behavioral items so that remediation efforts can be prioritized. The third step has three sub-steps, involving first the comparison of behavioral items to existing structural items, second the exploration of how unwanted behavioral items come to appear in design pattern instances, and third, the evaluation of behavioral items as they affect software quality and TD.

### 6.3.1 GQM

We use Basili's Goal-Question-Metric (GQM) approach [10] as a guide for this research. The GQM approach dictates an outline of high-level research goals (RG) supplemented with questions (RQ) and metrics (M) that guide the research. The GQM for this research is listed below:

**RG1**: *Explore* design pattern instances *for the purpose of* identifying and characterizing behavioral deviations *with respect to* proper pattern behaviors as defined by the design pattern's specification *from the perspective of* the software system *in the context of* design patterns in open source software systems.

**RQ1:** How does the behavior of a design pattern instance deviate from the expected behavior of that pattern type?

**RQ2:** Is there evidence to suggest that behavioral grime is present in pattern instances of a single pattern type?

**RQ3:** Is there evidence to suggest that behavioral grime is present in pattern instances across different pattern types?

**RQ4:** To what extent can a pattern instance have both structural and behavioral grime?

**RG2**: *Evaluate* design pattern behavioral grime *for the purpose of* understanding the value of behavioral grime *with respect to* structural grime and software evolution *from the perspective of* the software system *in the context of* design patterns in open source software systems.

**RQ5:** What is the relationship between structural and behavioral grime?

**RQ6:** Is the size of a design pattern instance related to the amount of behavioral grime in that pattern instance?

**RQ7:** What is the rate at which patterns accumulate behavioral grime?

**RG3**: *Quantify* the impact of behavioral grime *for the purpose* of capturing the effect of behavioral grime on patterns *with respect to* quality of pattern implementation and greater software system *from the perspective of* the software system *in the context of* design patterns in open source software systems.

**RQ8:** Are state of the art software quality analysis tools capable of identifying behavioral grime?

**RQ9:** How can the ISO 25010 software quality specification be implemented to consider design pattern grime?

**RQ10:** What is the relationship between behavioral grime and system quality and TD?

**Metrics**: Several metrics are outlined that will aid in answering the questions.

- **M1**: Structural Conformance
- **M2**: Behavioral Conformance
- **M3**: Structural Grime
- **M4**: Behavioral Grime
- **M5**: Pattern Integrity
- **M6**: Pattern Instability
- **M7**: Pattern Size
- **M8**: Pattern Age
- **M9**: Pattern Quality

Table 6.2 describes the formulations for each metric, with respect to a pattern instance *P*.

Table 6.2 Summary of the metrics selected for this analysis.

| Metric Name | Description |
|---|---|
| Structural Conformance (M1) | The percentage of structural roles in *P* that conform to at least one structural role from *P*'s SPS. |
| Behavioral Conformance (M2) | The percentage of behavioral roles in *P* that conform to at least one behavioral role from *P*'s IPS. |
| Structural Grime (M3) | A tuple <$SG_{\Sigma}$, $SG_{\delta+}$, $SG_{\delta-}$>, referring to the <count, number of additional elements, and number of removed elements>, respectively, that constitute structural grime in a single pattern *P*, in a single version. |
| Behavioral Grime (M4) | A tuple < $BG_{\Sigma}$, $BG_{\delta+}$, $BG_{\delta-}$ >, referring to the: <count, number of additional elements, and number of removed elements>, respectively, that constitute behavioral grime in a single pattern *P*, in a single version. |
| Pattern Integrity (M5) | $\frac{M1 + M2}{2}$ |
| Pattern Instability (M6) | Adopted from Martin's Instability metric (*I*) [50], the afferent coupling of *P* divided by the sum of the efferent coupling of *P* and the afferent coupling of *P*. $\frac{Ce(P)}{Ce(P) + Ca(P)}$ |
| Pattern Size (M7) | Adopted from Li and Henry's Size2 metric (*size2*) [48], the sum of attributes and methods across all classes in *P*. |
| Pattern Age (M8) | Age of a design pattern instance, calculated as a count of the number of software versions one design pattern instance appears in. |
| Software Quality (M9) | Scores of quality and all eight quality characteristics across an entire software project at a single version, derived from the QATCH toolchain [70] |

6.3.2 Study Design

The study design for this research is depicted in figure 6.3. To begin, we selected several software projects to study according to the selection process presented in the paragraph below. From these software projects, we identified design pattern instances using the design pattern detection tool described by Tsantalis et al. in [75]. We chose this tool because it is based on strong theory and claims little to no false positives in practice. Additionally, we used the tool SrcML [18] to assist in the source code parsing process. We chose this tool because it offers a translation from language-specific source code to standard format XML, meaning this process becomes language-agnostic. Following XML generation, we reverse-engineered the UML class and sequence diagrams of the entire software project. The entire software project's UML class and sequence diagrams need to be reverse-engineered, not just a subset, because the behavioral aspects that we wish to study require a holistic view of the software project, so that function calls and data types are assigned correctly. Once we had reverse-engineered the UML class and sequence diagrams, we generated a UML representation of the design pattern by combining the design pattern's detection results with the corresponding UML diagrams. Next, we subjected each design pattern instance to a process of coalescence. The process of pattern coalescence involves identifying members of the design pattern not captured by the design pattern detection tool. Such members may be sub-classes, super-classes, or pattern-methods within a pattern class that the design pattern detection tool may have missed. Following coalescence, we extracted the evolution of each pattern instance by tracking and connecting contributing roles of patterns across software versions.

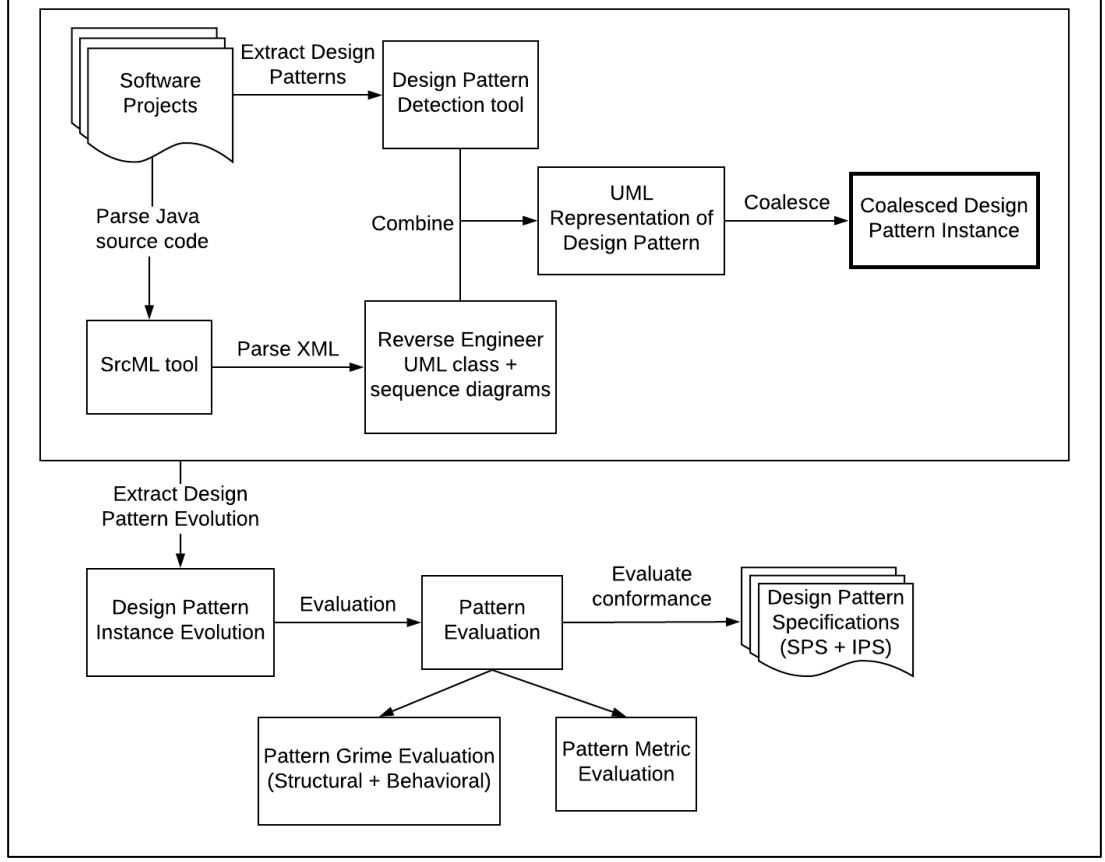


Figure 6.3 Summary of study design. Design pattern instances are extracted from software projects, and the associated UML is reverse-engineered from source code. The evolution of each pattern instance is generated, and evaluations for conformance, grime, and metrics are found across each pattern instance evolution.

Once pattern instance evolutions were generated, we entered the evaluation stage wherein we evaluated pattern conformance, pattern grime, and pattern quality/size for each version (pattern instance) in the pattern instance evolution. Regarding pattern conformance, we chose to evaluate each pattern instance to the pattern's SPS and IPS presented in the RBML specification [29]. While any user can modify a given pattern's SPS or IPS based on expectations for the pattern instance, utilizing the SPSes and IPSes presented in the specification offer a general solution that caters to domain differences,

even though such SPSes and IPSes may be considered too formal for many developers. Regarding pattern grime, there is the question of whether a non-pattern element is considered essential for the application, even if it might not align with the pattern's specifications. Such elements are not considered grime because the design pattern definition is flexible enough to allow for non-pattern elements to be present in the pattern instance while still allowing complete conformance. However, pattern grime has been shown to be present, and therefore we need to differentiate between non-pattern members that represent grime or not. To alleviate this differentiation, we made the assumption that each pattern is allowed one incoming non-pattern element and one outgoing non-pattern element. This assumption is based on usage of patterns; ideally a pattern will have one client, and we allow it to use up to one non-pattern class. Anything else is considered grime. While these precise values are configurable based on application, we chose such strict allowances for this study to model applications where program conformance is a necessity. The complete tool-chain is available under the MIT license at the following GitHub repository[18].

The process of selecting experimental units, or software projects, is as follows. In an effort to increase generalizability of results, we chose to analyze ten projects in total. To ensure relevancy, projects were selected based on their popularity ranking on the online code repository GitHub[19]. Specifically, we ranked all projects according to their 'number of stars', which is synonymous with a favorite or bookmark, and selected the first ten projects such that each project had at least 2,000 commits, 20 releases, and 100

[18] https://github.com/MSUSEL/msusel-pattern-behavior
[19] www.github.com

unique contributors. In most cases, all projects had significantly more than the minimum required filters; for example, the Selenium project features 23,550 commits, 116 releases, and 424 contributors. From each project, we selected 20 minor releases evenly divided between the oldest release and most recent release, under the assumption that each project followed traditional notation for release numbers, which is: [major.minor.bug_fix]. As an example, if a project had releases labeled v2.0 through v2.40, in which 40 minor releases existed between v2.0 and v2.40, we selected every other minor release (v2.0, v2.2, v2.4, ..., v2.38, v2.40). We utilized this process to generate an even spread of data points between the most recent release and the first release, providing an accurate summary of a project's history. The outcome from this project selection process is presented in table 6.3, along with the release numbers and respective release dates.

Table 6.3 Demographics of the projects under analysis.

| Project name | Domain | Releases | Release Dates |
|---|---|---|---|
| Apache Commons-lang | Java Libraries | 1.0 - 3.9 | Jul 2007 - Apr 2019 |
| Elasticsearch | Distributed Search Engine | 2.0.0 - 6.6.2 | Oct 2015 - Mar 2019 |
| Glide | Image caching library | 3.3.0 - 4.9.0 | Sept 2014 - Feb 2019 |
| Google Guava | Java Libraries | 9.0 - 27.1 | Apr 2011 - Mar 2019 |
| Hystrix | Fault tolerance library | 1.0.2 - 1.5.18 | Nov 2012 - Nov 2018 |
| Mockito | Unit Testing Framework | 1.8.0 - 2.28.1 | Jul 2009 - May 2019 |
| Netty | Asynchronous application | 4.0.0 - 4.1.34 | Jul 2013 - Mar 2019 |
| RxJava | Asynchronous Streaming | 2.0 - 2.2.7 | Oct 2016 - Feb 2019 |
| Selenium | Testing Framework | 3.0 - 3.141.59 | Oct 2016 - Nov 2018 |
| Spring-boot | Java packaging framework | 1.0 - 2.1.3 | Apr 2014 - Feb 2019 |

We chose to focus our analysis on seven pattern types; the Factory Method and Singleton patterns from the ‘Creational’ category [31], the Decorator and Object-Adapter patterns from the ‘Structural’ category [31], and the Observer, State, and Template Method patterns from the ‘Behavioral’ category [31]. Our initial intuition was that patterns in the behavioral category may be more prone to behavioral deviations, so we selected three pattern types from that category. Additionally, these seven pattern types provided us the largest sample size of detected pattern instances; many projects featured zero pattern instances of certain types, such as the Visitor or Prototype pattern. The count of pattern instance evolutions for each pattern type and across each project under analysis is shown in table 6.4. Note this is a count of pattern instance evolutions, not pattern instances; the difference being pattern instance evolutions track a single pattern instance across multiple versions, while pattern instances refer to a single pattern instance at a single software version.

Table 6.4 Count of pattern instance evolutions for each of the projects under analysis.

| Project Name | Decorator Evolutions | Factory Method Evolutions | (Object) Adapter Evolutions | Observer Evolutions | Singleton Evolutions | State Evolutions | Template Method Evolutions |
|---|---|---|---|---|---|---|---|
| commons-lang | 1 | 0 | 1 | 0 | 15 | 0 | 9 |
| elasticsearch | 13 | 68 | 126 | 0 | 214 | 186 | 81 |
| glide | 9 | 2 | 21 | 1 | 18 | 25 | 8 |
| guava | 4 | 20 | 4 | 0 | 35 | 28 | 89 |
| hystrix | 0 | 1 | 0 | 0 | 14 | 5 | 5 |
| mockito | 12 | 15 | 35 | 0 | 18 | 37 | 14 |
| netty | 14 | 52 | 42 | 0 | 97 | 138 | 56 |
| rxjava | 6 | 5 | 21 | 0 | 5 | 125 | 10 |
| selenium | 5 | 11 | 17 | 0 | 6 | 28 | 7 |
| springboot | 2 | 0 | 4 | 0 | 13 | 10 | 15 |
| Total | 66 | 174 | 271 | 1 | 435 | 582 | 294 |

## 6.4 Results

### 6.4.0 Preliminary Work[20]

To address the research questions, we began with a preliminary experiment in which we identified two specific behavioral deviations that can commonly occur while implementing design patterns. We know these behaviors are deviations because they detract from the intent of the design pattern and thus, have an undesired effect on TD. To illustrate these behaviors, consider the RBML sequence diagram of the IPS for the Observer pattern shown in figure 6.4 [30]. This example features a system that tracks the temperature and pressure of a kiln. The RBML sequence consists of the two roles in the Observer pattern; the Subject and the Observer. The two roles and their expected behaviors are shown on the top of the diagram. For behavioral conformance, it is expected that the Subject role calls the Update() operation on all the Observer roles when the Notify() operation is called. Then, the Observer calls the Subject using a GetState() operations. The bottom of the figure shows a pattern instance. Notice that the implementing classes in the sequence diagram conform to their role expectations. We use figure 6.4 as a baseline to which we inject our two behavioral deviations. Specifically, the two deviations we identifed are Excessive Action(s) and Improper Order of Sequences.

[20] Based on:
Reimanis D., Izurieta C., "Towards Assessing the Technical Debt of Undesired Software Behaviors in Design Patterns," IEEE ACM MTD 2016 8th International Workshop on Managing Technical Debt. In association with the 32nd International Conference on Software Maintenance and Evolution, ICSME, Raleigh, North Carolina, October 4, 2016.

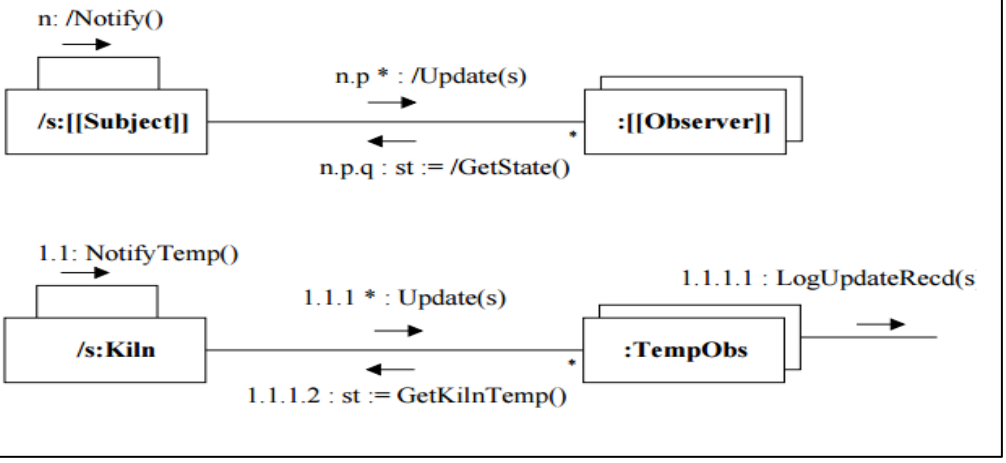


Figure 6.4 Diagram of the RBML sequence diagram for the Observer pattern [30], image from [64]. The Subject and Observer roles and their sequence of behaviors are shown on the top, and a conforming pattern instance is shown on bottom. The pattern instance conforms to the RBML sequence diagram.

6.4.0.1 Excessive Action(s). The first behavior deviation we consider involves one or more 'excessive' action(s) that occur during the standard runtime operation of the pattern. The excessive action(s) perform operations that are un-essential to the functional runtime behavior of the pattern. That is, if the excessive action(s) were to be removed, the pattern would behave in entirely the same manner (as expected). We characterize this type of behavior as behavioral grime because the excessive action(s) cause the pattern instance to not conform to the pattern's IPS. The excessive action(s) are not necessarily structural grime, but may be the result of implementing new functional requirements. Regardless, the intent of the pattern, according to its IPS is violated. The addition of excessive action(s) affects software maintainability and TD, because while the pattern will achieve the same external behavior, modifying the pattern in the future will require domain knowledge of the action(s) and their intent.

Our illustrative implementation of this behavioral deviation uses a loop construct that counts to 100 and sums values along the way. The loop's internal and external behaviors are not referenced by any other components in the remainder of the software application. Practically speaking, this may happen when a developer forgets to delete debugging code or decides upon a different strategy for the implementation of an algorithm half-way through development and forgets to delete the original code.

Although all design patterns allow for the introduction of new tasks (i.e., excessive actions) as interspersed elements of existing behaviors, the introduction itself needs to be explicitly described by elements of the RBML IPS diagram. If not, then the introduction of the behavior is unintended, even if it provides needed functionality for the software. The IPS of a design pattern must then be responsible for capturing the strictness of adherence with which instances are created. Only then, new functionality can be planned for without affecting TD. To illustrate how this behavior violates the RBML IPS diagram in our implementation, refer to figure 6.5 [30]. The Observer RBML IPS is shown on the top. Our implementation of the Observer pattern is shown on the bottom. This figure is similar to figure 6.4, with the exception that the excessive behavior has been injected. The excessive behavior is shown in red, and labeled with the operation 'Excessive()'. Notice that in the RBML IPS, the Observer role only performs one operation, which is to call GetState() from the Subject role. Conversely, the TempObs class performs two operations, Excessive() and then GetState(). Because of this, TempObs has behavioral grime.

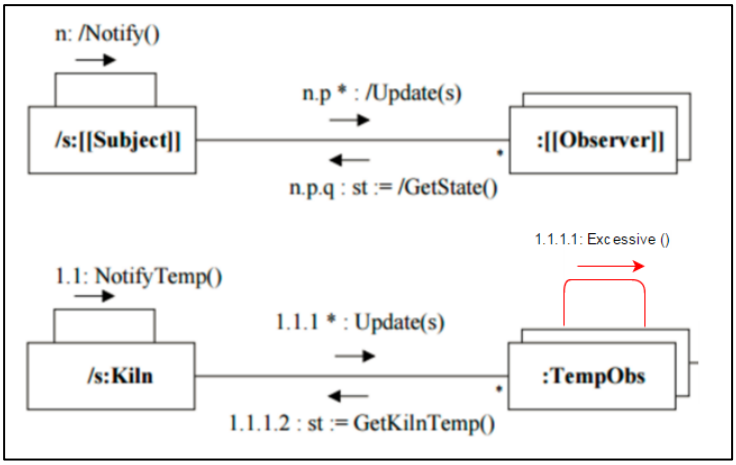


Figure 6.5 Excessive behavior grime in an Observer pattern instance (ObserverExcess) [30], image from [64]. The top sequence of this image illustrates the RBML IPS of the Observer pattern. The bottom sequence illustrates our application of the Observer, with the injected behavior 'Excessive()'. The injected behavior constitutes behavioral grime because it does not conform to the pattern's IPS.

6.4.0.2 Improper Order of Sequences. The second behavioral deviation we consider involves cases where the order of operations that a pattern should be following, according to the SPS and IPS, is improperly sequenced. This type of behavioral deviation constitutes behavioral grime because it causes the pattern instance to not conform to the IPS of the pattern. Additionally, this type of behavioral deviation affects the maintainability and TD of the pattern for much the same reasons as the excessive action(s) does; any modification of the pattern instance in the future will be hindered by the need to first understand the order of sequences in the application.

In our implementation of this behavioral deviation, we injected a class that represents a valid extension of each pattern. However, the injected class was only instantiated from the incorrect class role in each pattern. In other words, in our Observer

pattern instance we injected an observer that only received updates from other observers. This is incorrect behavior for this pattern because the Observer pattern instance should enforce that the subject is responsible for updating the observer. Practically speaking, this would happen if a developer who was unfamiliar with the Observer pattern, either a novice developer or a new hire, made changes to the existing pattern.

Figure 6.6 [30] illustrates the improper order of sequences errant behavior in our application. The top sequence of the figure features the RBML IPS of the Observer pattern. The bottom sequence illustrates the behavior of the Observer pattern in our application. The TempObs2 class was added to the pattern, which has the potential to be a proper extension to the pattern instance. However, its state is being updated from the TempObs class, which belongs to the Observer role. This is a violation of the RBML of the Observer pattern because the Subject is responsible for updating the Observers. TempObs2 is being updated immediately after the update to TempObs, even before TempObs has called the GetState() operation. This type of behavior constitutes behavioral grime.

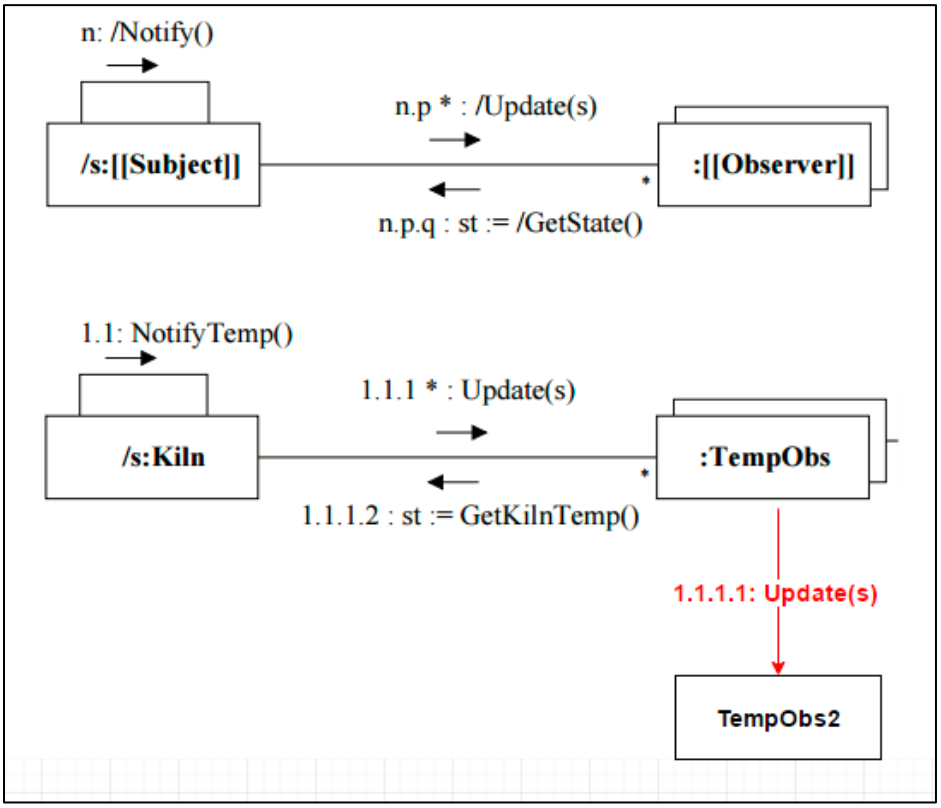


Figure 6.6 Improper order of sequences behavior grime in an Observer pattern instance (ObserverImprop) [30], image from [64]. The top sequence of this image illustrates the RBML IPS of the Observer pattern. The bottom sequence illustrates our application of the Observer, with the injected class TempObs2 and injected operation Update(s) (shown in red). The injected behavior constitutes behavioral grime because it does not conform to the pattern's IPS.

### 6.4.1 RQ1

With preliminary results identified, we began our assessment of research question 1, which is concerned with identifying how the behavior of a design pattern instance can deviate from the expected behavior of that pattern type. To further assess this question, we performed an *in-vitro* experiment [40] in which we implemented the proposed behavioral deviations from section 6.4.0. Specifically, we began with an implementation of the Observer pattern such that the implementation perfectly aligned to its SPS and IPS.

Such an instance might be impractical in the real-world, yet would mark a starting point for our experiments. To this Observer pattern instance, we injected code that constitutes modular structural grime, as presented by Schanz and Izurieta [67]. Modular structural grime is concerned with the relationships that pattern members may have with either other pattern members, or non-pattern members. Therefore, modular structural grime provides a constraint on all possible pattern behaviors. In other words, a given behavior, whether between pattern members or non-pattern members, cannot exist unless the two members share a relationship. To each injected modular grime instance, we applied the behavioral deviations as presented by Reimanis and Izurieta [64]. Specifically, these deviations are 'Improper Order of Sequences', in which expected behaviors occur in an incorrect order, and 'Excessive Actions' in which excessive actions hamper the run-time expectations of a pattern. For this work, we chose to focus on a subset of Excessive Actions, which we refer to as 'Repetitive Actions', or cases where the same behavior is performed within the same scope, or function call, of a pattern instance at run-time. After applying said behavioral deviations to the modular grime taxonomy, we generated a taxonomy of behavioral grime, which is shown in figure 6.7.

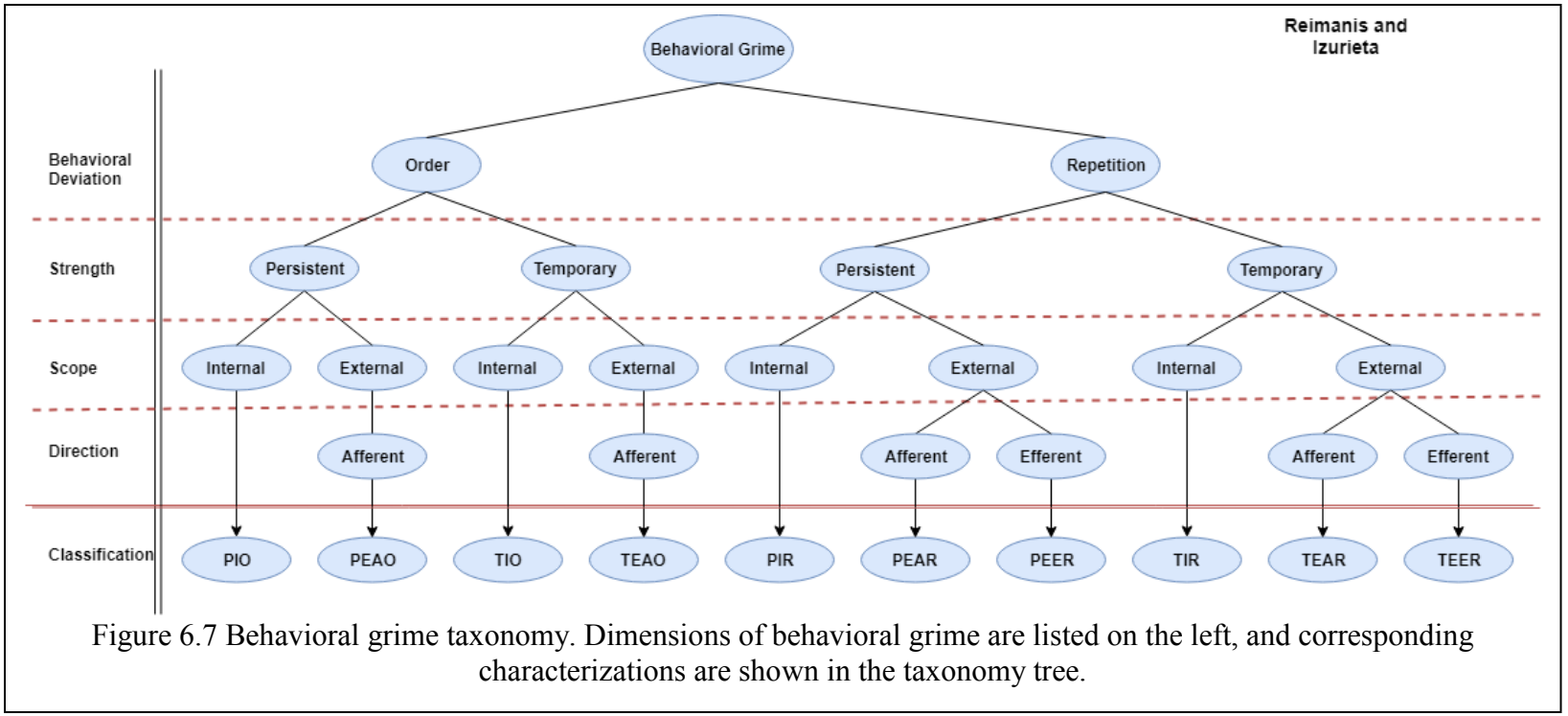


Figure 6.7 Behavioral grime taxonomy. Dimensions of behavioral grime are listed on the left, and corresponding characterizations are shown in the taxonomy tree.

The dimensions for this taxonomy are mirrored from the modular grime taxonomy [67], which are explained as follows. Strength refers to the strength of a relationship between two UML members; Persistent Strength refers to a UML association while Temporary Strength refers to a UML use-dependency. Scope refers to the context of the relationship between two UML members; Internal Scope refers to a relationship between two pattern members, and External Scope refers to a relationship between one pattern member and one non-pattern member. Direction refers to the direction of the relationships. Afferent Direction refers to an incoming relationship while Efferent Direction referring to an outgoing relationship. In the taxonomy, the Classification row refers to the acronym that captures that type of behavioral grime; for example, the PIO classification is an acronym for 'Persistent-Internal-Order' grime. This behavioral grime taxonomy closely mirrors the modular grime taxonomy presented in [67], with two exceptions. First, we have incorporated the 'Behavioral Deviations' dimension, which corresponds to the type of behavioral grime (Order or Repetition). Second, the taxonomy is not symmetrical across Order and Repetition sub-trees; specifically, the sub-tree pertaining to External Efferent Order (-EEO) type grime is nonexistent. This is because this sub-tree represents an outgoing relationship from a pattern member to a non-pattern member cannot be in an incorrect order; such relationships are not captured by the design pattern, and thus cannot be in an incorrect order.

While the taxonomy of behavioral grime was initially created from synthetic *in-vitro* examples, we validated this taxonomy by identifying instances of each form of

behavioral grime in real-world systems. These results are presented in research questions 2 and 3.

### 6.4.2 RQ2

This section reports the results from RQ2, which postures, “Is there evidence to suggest that behavioral grime is present in pattern instances of a single pattern type?.” To answer this question, consider table 6.5, which summarizes the grime counts found from our analysis. Each cell in the table refers to a non-unique count of behavioral grime across all pattern instances under analysis, of the corresponding pattern type. The phrasing non-unique grime refers to counting the same grime artifact more than once, if it appears in more than one pattern version. For this specific research question, we consider the columns of the table, because the columns report counts of behavioral grime for a single pattern type. For all patterns except the Observer pattern, we see relatively large counts of behavioral grime, with the State pattern reporting the largest raw count of non-unique instances of grime. However, these numbers appear inflated because we encountered a different number of pattern instances for each pattern type. To counter this inflation, we have included a ‘Normalized Total’ row, which refers to the Raw Total divided by the count of pattern instances for each pattern type.

Table 6.5 Count of behavioral grime across each pattern instance.

| Behavioral Grime Type | Decorator | Factory Method | (Object) Adapter | Observer | Singleton | State | Template Method | Total |
|---|---|---|---|---|---|---|---|---|
| PEAO | 0 | 0 | 0 | 0 | 0 | 372 | 0 | 372 |
| PIO | 0 | 0 | 0 | 0 | 0 | 13 | 0 | 13 |
| TEAO | 0 | 0 | 0 | 0 | 0 | 4014 | 0 | 4014 |
| TIO | 0 | 0 | 0 | 0 | 0 | 30 | 0 | 30 |
| PEAR | 1155 | 117 | 8309 | 0 | 0 | 14580 | 192 | 24353 |
| PEER | 9375 | 7722 | 12572 | 0 | 3340 | 25741 | 6458 | 65208 |
| PIR | 3723 | 102 | 1182 | 0 | 0 | 6206 | 164 | 11377 |
| TEAR | 8998 | 1166 | 15007 | 0 | 0 | 66026 | 258 | 91455 |
| TEER | 41655 | 41407 | 82704 | 0 | 34632 | 201773 | 76324 | 478495 |
| TIR | 3691 | 739 | 2683 | 0 | 24 | 9148 | 617 | 16902 |
| Raw Total | 68597 | 51253 | 122457 | 0 | 37996 | 327903 | 84013 | |
| Normalized Total | 64.47 | 19.66 | 34.50 | 0 | 10.70 | 39.83 | 19.65 | |

To answer this exploratory research question, we look at the raw and normalized counts of grime across the various pattern types. We encountered behavioral grime from every pattern we studied except the Observer pattern, and the most prevalent form of behavioral grime for each individual pattern type was TEER grime, or Temporary External Efferent Repetition grime. This is not a surprise, as grime of this type manifests itself as non-pattern members that are used by a pattern, but only as a use-dependency (not an association). This may occur when a new functionality requires extension of a design pattern instance, but pressures from management or clients force a quick-and-dirty change. Generally speaking, behavioral grime concerned with Order was the rarest form of grime, and was only identified in State pattern instances. The counts were low; for example, we identified only 13 instances of PIO (Persistent Internal Order) grime in our study. Upon further investigation, we discovered that all PIO grime came from the same pattern instance, suggesting that this form of behavioral grime might be rare. Furthermore, we found no evidence of behavioral grime in the Observer pattern instances under our analysis. However, because we identified only a single Observer pattern instance from the ten projects under analysis, a meaningful exploration of the Observer pattern is not possible. Regardless, in terms of answering our second research question (RQ2), we are able to answer in the affirmative for six out of seven of our pattern types (all except the Observer pattern), that behavioral grime is indeed present in pattern instances of a single pattern type.

6.4.3 RQ3

This section reports the results from RQ3, which generalizes RQ2 to consider, “Is there evidence to suggest that behavioral grime is present in pattern instances across different pattern type?.” To answer this research question, we begin by referencing table 6.5, considering the rows of the table which represent the counts of behavioral grime types across pattern instances. We see that many behavioral grime types are present in pattern instances of more than one pattern type. Specifically, PEER, TEER, and TIR grime appear in all pattern types analyzed, except the Observer pattern. PEER (Persistent External Efferent Repetition) and TEER (Temporary External Efferent Repetition) grime are not necessarily a surprise, based on intuition. Grime of these forms represents a relationship from a pattern member to a non-pattern member, likely indicating an extension of the pattern instance to accommodate new functionalities within the software project. TIR (Temporary Internal Repetition) grime refers to temporary use dependencies that appear between two or more pattern members as the pattern ages, indicating new relationships between that likely facilitate new functionalities elsewhere in the code-base. The appearance of PEER, TEER, and TIR grime across the Decorator, Factory-Method, (Object) Adapter, Singleton, State, and Template-Method patterns confirms that these patterns are susceptible these forms of grime.

Furthermore, PEAR, TEAR, and PIR grime was found in five of the seven pattern types analyzed, excepting the Observer and Singleton pattern. PEAR (Persistent External Afferent Repetition) and TEAR (Temporary External Afferent Repetition) grime refers to non-pattern members that establish a relationship to pattern members, over the pattern’s

lifetime. The lifetime of these non-pattern members is not specified under this analysis; they could be new classes added later in the project's lifetime, or they could be classes that have existed since the first version of the software, and the relationship between it and the pattern might have been established in a later version. PIR (Persistent Internal Repetition) grime refers to a stronger relationship, exclusively between pattern members. The elusiveness of these three forms of grime (PEAR, TEAR, PIR) in the Singleton pattern is not a surprise. The majority of Singleton pattern instances featured only one class, and did not deviate from the expected structure or behavior. Because of the uniqueness of a Singleton pattern, specifically that only one instance of that class is allowed, intuitively it seems rare that additional classes would be created that depend on the Singleton because these additional classes would not be able to retrieve a new instance of the Singleton. There is an important distinction here, between PIR and TIR grime. Recall TIR grime refers to the addition of UML use-dependencies between two pattern members, and PIR grime refers to the addition of UML associations between two pattern members. Our results indicate that TIR grime was found in Singleton instances, but PIR grime was not. If PIR grime was found in a Singleton, it might indicate that an extraneous class variable references the Singleton class itself, which violates the purpose of the Singleton pattern because the Singleton object should only be stored and retrievable from one single class variable. The Singleton IPS allows class variables to reference the Singleton class itself, but the existence of PIR grime suggests this is happening more than the allowable amount. Such an instance of a Singleton would be referred to as a rotted pattern, because the addition of elements would have violated its

purpose. Therefore, our results indicate that only five of the seven pattern types analyzed are susceptible to PEAR, TEAR, and PIR grime. These pattern types are the Decorator, Factory-Method, (Object) Adapter, State, and Template-Method patterns.

Order grime was the rarest form of grime encountered from our analysis, and was only present in small amounts in State pattern instances. We see three likely viable options for this phenomenon. First, it could hold that the State pattern is the only design pattern under our analysis that is susceptible to Order grime. However, when answering RQ1, in section 6.4.1, we successfully injected Order grime into an Observer pattern instance to show that Order grime can exist in Observer patterns. Because this occurred in an *in-vitro* setting and not an *in-vivo* one [40], we have chosen to not consider it in this discussion. A second viable explanation for this phenomenon holds that the State pattern was the largest sampled pattern type in our analysis, and thus the scope of our results when considering the State pattern is over-amplified. In statistical terms, these results could have appeared due to a sampling bias which increases the likelihood that grime might exist in a State pattern instance. A third explanation for this result is related to the second explanation, but considers that we did not sample enough of the other pattern members to uncover the 'true population group' that such a form of grime exists in. It seems most likely that this third option holds true; that Order grime is indeed rare but that other pattern types are exposed to it, yet replicating these experiments on more pattern instances and across more software projects might yield pattern instances that contain Order grime.

6.4.4 RQ4

This question considers the extent to which a pattern instance can have both structural and behavioral grime. To answer this question, we consider the presence of structural and behavioral grime for a single pattern instance, across each pattern type. The grid in figure 6.8 displays the labels for all four possibilities of structural and behavioral grime presence and absence, in a single pattern instance. For example, the label **A** would be applied to a pattern instance if the instance accumulated at least one case of both structural and behavioral grime, over its evolution. In order to answer this question, we labeled every pattern instance in our analysis according to this labeling scheme and summed the counts of each label across each pattern type. The results from this aggregation are presented in table 6.6.

| | Behavioral grime present | Behavioral grime absent |
|---|---|---|
| Structural grime present | **A** | **B** |
| Structural grime absent | **C** | **D** |

Figure 6.8 Grime quadrant of possible grime types. For a given pattern, rows correspond to at least once instance of behavioral grime existing in the pattern, and columns correspond to at least one case of structural grime existing in the pattern.

Table 6.6 Count of labels, according to figure 6.8, for each pattern instance evolution, separated by pattern type.

| **Pattern Type** | **A** counts | **B** counts | **C** counts | **D** counts |
|---|---|---|---|---|
| Decorator | 53 | 13 | 0 | 0 |
| Factory Method | 91 | 76 | 0 | 7 |
| (Object) Adapter | 232 | 34 | 0 | 5 |
| Observer | 0 | 1 | 0 | 0 |
| Singleton | 122 | 312 | 0 | 1 |
| State | 411 | 139 | 0 | 32 |
| Template Method | 131 | 147 | 6 | 10 |
| Total | 1040 | 722 | 6 | 55 |

Table 6.6 shows the counts of each grime possibility label across each pattern type under our analysis. The most immediate and striking result from this table is that the column featuring the label **C** is nearly empty, only showing six cases where a pattern instance evolution contained behavioral grime, but no structural grime. Additionally, all six cases occurred within the Template Method design pattern. Upon initial glance, this result seems counter-intuitive; the structure of a pattern dictates behavior, so it seems impossible for a pattern instance to be susceptible to behavioral grime when there is no structural grime. However, the important distinction is that grime is only considered as such if it manifests as unexpected elements within a pattern instance. In pattern instances where behavioral grime is present yet structural grime is absent, the structural elements are correct and expected while the behavioral elements are unexpected. As an example where such an occurrence would appear, consider a variable that is being initialized more times than expected; the fact the variable has a dependency or association to another class matches with structural expectations, but structural checks would fail to find issue when the same variable is being initialized more than once. The structural perspective only illustrates such a case as one single structural element; thus a deeper level of granularity

into the system is required, which is what behavioral grime analysis provides. While we only found six instances of label **C** across our pattern instance evolutions, label **A** shows the case where both structural and behavioral grime were found at least once in a pattern instance evolution. Label **A** is the most prevalent label, suggesting that patterns tend to be susceptible to both structural and behavioral grime. Label **B** considers cases where structural grime is present, yet behavioral grime is absent. Similar to label **C**, this label may seem non intuitive– how can a pattern instance have structural grime elements with no behavioral grime elements? The explanation is similar to the explanation for label **C**, yet flipped. A pattern can have unexpected structural elements that constitute grime, but if each of those elements behave how they are expected to behave; i.e., properly, there would be no behavioral grime. Such a case might occur when applying a design pattern for a library, that will be used heavily by other components in the project, or other projects. Compared to label **A**, we found a slightly lower number of label **B** patterns, yet still more than labels **C** and **D.** This result suggests that the structural aspects of a pattern are violated more frequently than the behavioral aspects, but the most common case is that both structure and behavior are violated together.

### 6.4.5 RQ5

This research question is concerned with identifying the relationship between structural and behavioral grime. To answer this question, we began by generating a pairwise scatter-plot for each type of structural and behavioral grime, which is shown in figure 6.9. Structural grime is shown on the x-axis, and behavioral grime is shown on the

y-axis. Points in the scatter-plot represent the count of modular grime and type of behavioral grime for a single pattern instance.


Figure 6.9 Pairwise scatter-plots illustrating the relationships between structural grime, shown on the x-axis, and behavioral grime, shown on the y-axis.

To assess the strength of each relationship between structural and behavioral grime, we calculated pairwise correlation coefficients and respective *p*-values for each of structural grime and behavioral grime. The nature of our data is a count, which falls under the ratio numeric scale, and a visual assessment of the scatter-plots suggests a

linear relationship in several plots. We choose to use Pearson's method to evaluate the precise nature of the relationships, because Pearson's provides a parametric estimate of correlation coefficients, compared to the nonparametric alternatives of Spearman's ρ or Kendall's τ. The application of Pearson's requires addressing two primary assumptions; the normality assumption and the independence assumption. We may say we have not violated the normality assumption to a great extent because of the large sample size of our data [87]. However, we cannot say we have satisfied the independence assumption. Specifically, each data point comes from a single pattern instance in a single software version, and pattern instances may appear in more than one software version, meaning grime in a future version might be, and likely is, dependent on grime in previous versions. We alleviate this concern because of the number of pattern instance evolutions we have detected, which is captured in table 6.4, but we cannot say we have satisfied the independence assumption. Regardless, we assume this threat to validity, and provide the correlation coefficients for each pairwise relationship between structural and behavioral grime in table 6.7, with strong relationships ($r > 0.60$ or $r < 0.60$) shown in bold. We also calculated the *p*-values for each pairwise correlation coefficient, which is shown after every correlation coefficient values in table 6.7. Because this work is concerned with identifying the strength of the relationship between pairwise metrics, we chose to assume a very weak relationship exists in the first place. In other words, all of our null hypotheses assume no correlation. Therefore, each *p*-value corresponds to the probability that the correlation coefficient we received did not occur because of chance, under the assumption that the true correlation coefficient is zero, which implies a very weak

relationship. *P*-values across each pairwise comparison are shown as the second value in each cell in table 6.7.

Table 6.7 Correlation coefficients and respective p-values for each pairwise type of grime, using Pearson's method. Columns feature structural grime, while rows feature behavioral grime. Correlation coefficients are shown first and p-values second, separated by a forward slash. Strong relationships ($r > 0.60$ or $r < -0.60$) are shown in bold.

| | PEA grime | PEE grime | PI grime | TEA grime | TEE grime | TI grime |
|---|---|---|---|---|---|---|
| PEAO grime | 0.2511 / <1e-16 | 0.1583 / <1e-16 | **0.6633** / <1e-16 | **0.5962** / <1e-16 | 0.1488 / <1e-16 | **0.6812** / <1e-16 |
| PIO grime | -0.0055 / 0.37 | 0.0003 / 0.96 | -0.0021 / 0.73 | -0.0061 / 0.33 | 0.0027 / 0.66 | -0.0022 / 0.72 |
| TEAO grime | 0.0971 / <1e-16 | 0.4705 / <1e-16 | 0.1321 / <1e-16 | 0.2181 / <1e-16 | 0.4000 / <1e-16 | 0.3688 / <1e-16 |
| TIO grime | -0.0008 / 0.89 | 0.0433 / 2.65e-12 | -0.0060 / 0.34 | -0.0063 / 0.31 | 0.0294 / 2.04e-6 | -0.0045 / 0.47 |
| PEAR grime | 0.2475 / <1e-16 | 0.1311 / <1e-16 | 0.4653 / <1e-16 | 0.5028 / <1e-16 | 0.1178 / <1e-16 | 0.5090 / <1e-16 |
| PEE grime | 0.1694 / <1e-16 | **0.6984** / <1e-16 | 0.3415 / <1e-16 | 0.3453 / <1e-16 | **0.5991** / <1e-16 | 0.4174 / <1e-16 |
| PIR grime | 0.1408 / <1e-16 | 0.2630 / <1e-16 | 0.4215 / <1e-16 | 0.4251 / <1e-16 | 0.2571 / <1e-16 | 0.4453 / <1e-16 |
| TEAR grime | 0.0675 / <1e-16 | 0.0537 / <1e-16 | 0.1398 / <1e-16 | 0.3237 / <1e-16 | 0.0527 / <1e-16 | 0.1527 / <1e-16 |
| TEER grime | 0.2395 / <1e-16 | **0.6912** / <1e-16 | 0.2607 / <1e-16 | 0.2613 / <1e-16 | **0.8219** / <1e-16 | 0.4101 / <1e-16 |
| TIR grime | 0.1306 / <1e-16 | 0.3837 / <1e-16 | 0.3201 / <1e-16 | 0.3325 / <1e-16 | 0.3841 / <1e-16 | 0.3941 / <1e-16 |

Because of the inherent relationship between structure and behavior, specifically that structure enforces behavior, our initial expectations held that grime types across their mirrored dimensions would share a strong relationship. Specifically, these would be forms of grime that share every dimension except for their structural or behavioral distinction. For example, TEE (Temporary External Efferent) grime and TEER (Temporary External Efferent Repetition) grime share every dimension except for their

structural and behavioral distinction. We found that only two such pairs of mirrored grime shared strong relationship; PEE (Persistent External Efferent) and PEER (Persistent External Efferent Repetition), and TEE and TEER grime. These two pairs support our expectations, but no other mirrored pairs do. This is an interesting finding because it suggests that across mirrored pairs, structural and behavioral grime do not appear at the same rate. Two scenarios explain this result. The first scenario considers one instance of structural grime that has multiple behavioral grime instances associated with it, such as a variable that constitutes structural grime, that is being improperly initialized multiple times. In such a case, each improper initialization would constitute as behavioral grime. The second scenario considers pattern members that contain structural grime, yet their behavior aligns with behavioral expectations, so they would not constitute behavioral grime. Upon initial inspection, this scenario appears impossible because if a pattern element has structural issues, then surely any behaviors associated with it are issues as well. However, this is not the case because the definition of pattern grime allows for this scenario. As an example, consider a pattern instance that contains two different variables that fulfill the same role, as a relationship between pattern members. Additionally, these two variables have identical behaviors that match behavioral expectations of the pattern. If the specifications of this pattern hold that only one variable should be fulfilling that role, then one variable would constitute structural grime. However, because the two variables behave identically, we cannot say which variable should be the one that constitutes structural grime. Therefore, such a pattern would contain structural grime with no behavioral grime, illustrating our second scenario.

Such a scenario might be rare in the real world, yet it is important to consider that it exists and may explain this situation.

Considering grime types that are not mirrored across their dimensions, we found strong relationships between the following pairs: TEER/PEE, PEER/TEE, PEAO/PI, PEAO/TEA, and PEAO/TI. Discussing Repetition grime first (TEER/PEE and PEER/TEE), strong relationships between these two pairs are not unexpected. The dimensions these pairs of grime share are 'External Efferent', which refer to non-pattern members that a pattern member uses. Grime of these types would appear in a pattern instance to accommodate new features in the software project that the pattern instance uses. From a design perspective, this means the pattern instance is being extended, but not in the correct and intended manner. Such improper extensions would happen if under pressure to release new features quickly, or perhaps if a developer was unfamiliar with the design pattern. In terms of the strong Order grime relationships we encountered, a visual analysis of the scatter plots in figure 6.9 suggests that these strong correlation coefficient values might be unfounded. Order grime was the rarest form of grime we encountered, yet when it was found the counts for it were relatively high. This, coupled with the numerous counts of structural grime we encountered across pattern instances with any type of Order grime, implies that a pattern instance with Order grime likely has high structural grime, which explains the strong relationship. This result may be contentious, as the correlation coefficients are strong and their respective *p*-values are low, yet a visual analysis of the plots shows otherwise. Our claim is that Order grime and structural grime are related in that the presence of Order grime is associated with high

values of structural grime, but ultimately patterns that contain more instances of Order grime are required before a more accurate correlation coefficient describing the relationship can be attained.

Of particular note in these relationship plots are the low *p*-values. Recall our null hypothesis that there exists a very weak relationship between structural and behavioral grime types, specifically one in which the correlation coefficient is zero. All *p*-values with respect to Repetition grime are very low, suggesting that we reject the null hypothesis that the correlation coefficients are equal to zero. This result advocates that a relationship does exist between structural and behavioral grime types. However, we cannot claim that the correlation coefficients we found are accurate estimates of the true relationship between any pair of structural and behavioral grime instance. Though, we can point out that due to the large sample size of grime instances, coupled with visual analyses of the scatter plots, the correlation coefficients may provide a good estimate of the actual relationship. Of course, more experiments need to be performed to assert this result.

### 6.4.6 RQ6

This research question is concerned with identifying if the size of a design pattern instance is related to the amount of behavioral grime in that pattern instance. To capture design pattern size, we chose to use an adaptation of Li and Henry's *Size2* metric, which we refer to as M7 and is explained in table 6.4. Similarly to RQ5 in section 6.4.5, we began by generating pairwise scatter-plots showing size and each behavioral grime type to visually assess trends. This scatter-plots are shown in figure 6.10.

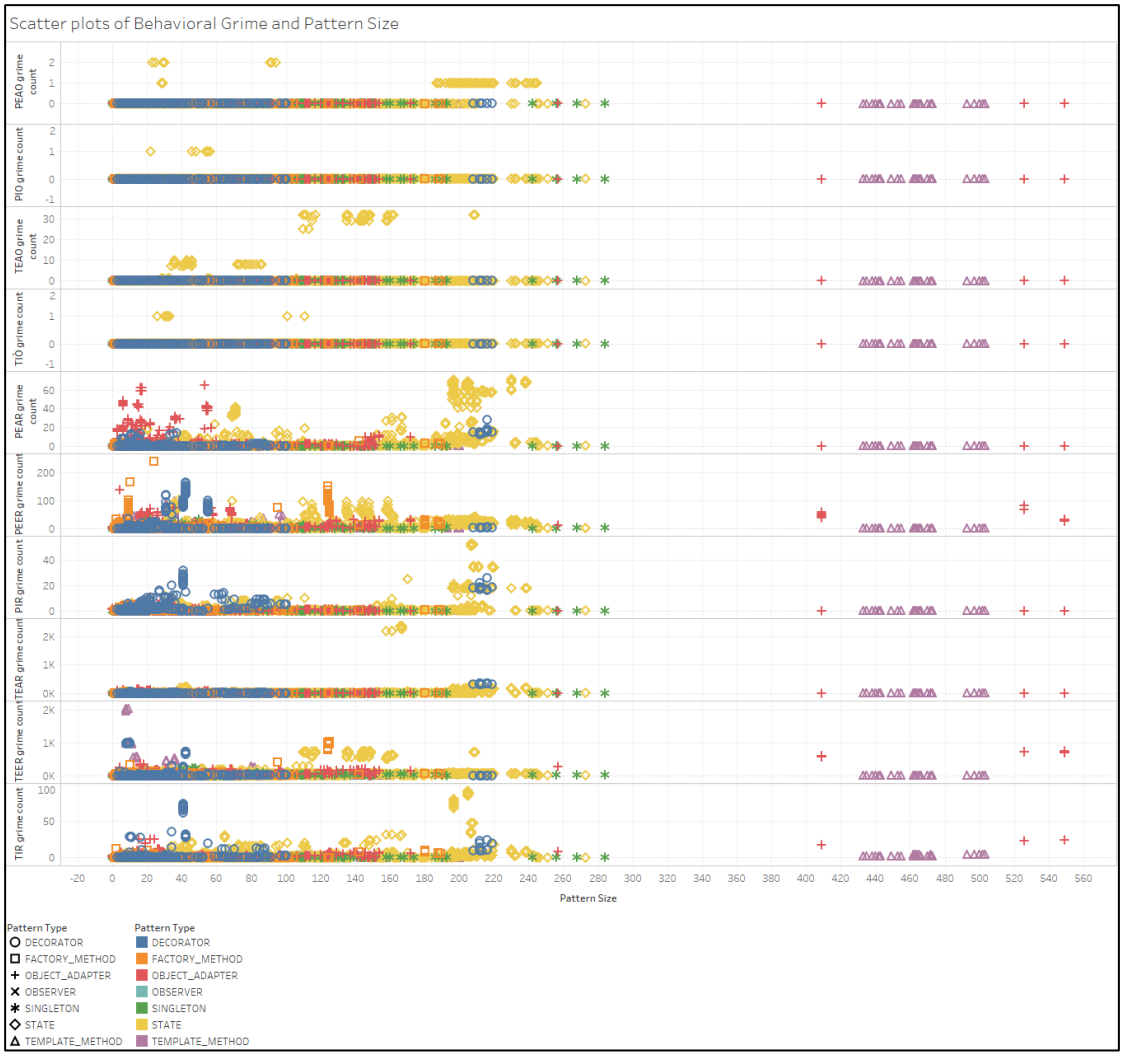


Figure 6.10 Scatter plots of behavioral grime and pattern size. Behavioral grime is on the y-axis, Pattern Size is on the x-axis, and points in the plot represent individual pattern instances, which are shaped and colored according to their design pattern type.

An initial visual inspection of the scatter plots in figure 6.10 reveals that no clear relationship appears from the data. Some patterns with small size (0-40 members) have large counts of behavioral grime, while some pattern with medium size (190-230 members) also have large counts. The internal Repetition grime cases (PIR and TIR) appear to have the most monotonically linear relationship, but many points in the plot feature zero grime. To further assert the data, we calculated the correlation coefficients

and respective *p*-values for each pair of behavioral grime and pattern size, under similar conditions as our structural vs behavioral analysis, in section 6.4.5. That is, because the nature of our data is a count which falls under the ratio numeric scale, we chose to use Pearson's method for calculating the correlation coefficients. Pearson's requires assessment of the independence assumption, which we cannot say we have satisfied because of several confounding factors, such as the fact that our data consists of evolutionary data, in which one value likely depends on a value from a previous evolution state. We can say we have alleviated the independence assumption slightly because of our large sample size, and that any one evolutionary chain will not carry as much weight because of the sheer number of data in the analysis. The null hypotheses we use to test our *p*-values are the cases where correlation coefficient is equal to zero, which implies a very weak relationship. Therefore, each *p*-value corresponds to the probability that the correlation coefficient we received is not due to chance, under the assumption that the true correlation coefficient is zero. We show the correlation coefficients and respective *p*-values in table 6.8.

Table 6.8 Correlation coefficients and respective p-values for each type of behavioral grime and pattern size, using Pearson's method. Columns show the correlation coefficients and p-values, and rows shows behavioral grime.

| **Behavioral grime type** | **Correlation Coefficient** | **p-value** |
|---|---|---|
| PEAO grime | 0.4508 | <1e-16 |
| PIO grime | 0.0114 | 0.06 |
| TEAO grime | 0.1751 | <1e-16 |
| TIO grime | 0.0090 | 0.14 |
| PEAR grime | 0.3456 | <1e-16 |
| PEE grime | 0.2797 | <1e-16 |
| PIR grime | 0.3231 | <1e-16 |
| TEAR grime | 0.1897 | <1e-16 |
| TEER grime | 0.2918 | <1e-16 |
| TIR grime | 0.2874 | <1e-16 |

As expected based on our visual analysis of the scatter plots in figure 6.10, the correlation coefficients as presented in table 6.8 are relatively low for all forms of Repetition grime (0.18 - 0.34), and most forms of Order grime (0.00 - 0.45). This suggests a weak relationship between these forms of pattern grime and pattern size. Furthermore, the *p*-values associated with these correlation coefficients are very low, suggesting we reject the hypotheses that the relationship between size and each type of behavioral grime is zero. In other words, the correlation coefficient values we received are correct in the context of this study, under the assumption that the true relationships between behavioral grime and pattern size is zero. However, they are still quite low and would not be useful in any predictive sense. This finding is interesting though; it implies that behavioral grime appears in pattern independently of the pattern's size. The definitions and specifications of design patterns that we use are extendable in the sense that a pattern could have any number of members, yet still conform perfectly (i.e., no grime or rot). And indeed, we see this from the data. Several pattern instances that have

over 500 members have zero behavioral grime. Pattern instances such as these would be considered good implementations of the expected design pattern because they do not deviate from their intent. Therefore, extending these pattern instances to allow for future functionalities would be easier than one deviates significantly. Our results indicate that pattern instances with the opposite measurements are present too. That is, pattern instances with small size yet a high count of behavioral grime. Some patterns under this designation feature dozens of behavioral grime instances while only having a single-digit size. In code, this occurs when one or a few members are used in an unexpected manner a multitude of times; consider a pattern instance's variable that is used by numerous non-pattern members, in an unexpected manner. Pattern instances such as these illustrate cases where the amount of behavioral grime makes the pattern instance difficult to extend and maintain, which sacrifices many of the good qualities usage of the pattern offers in the first place.

### 6.4.7 RQ7

Research question 7 is concerned with identifying the rate at which patterns accumulate behavioral grime. This question is fairly broad, but left so intentionally. We understand that numerous confounding factors exist in this space that affect the rate at which a pattern instance accumulates behavioral grime, such as project coding standards, developer habits, changing technology dependencies, project domain, etc., but several of these confounding factors are impossible to retrieve and capture in a model. Therefore, we outline the process and results from our analysis, but we leave the format of the statistical models general so that terms can be added to it, to capture these confounding

factors if a particular domain has knowledge of them. The process we chose to answer this question involved first identifying the factors that have a significant effect on behavioral grime. To do this, we performed several ANOVA calculations, one for each form of behavioral grime, with three factors we had available based on our project selection process, and one derived factor. The factors we had available from the selection process are Project, Pattern Type, and Software Version, which are all categorical variables. The derived factor is Pattern Age, explained in the metrics table, table 6.2 in section 6.2, which is a categorical variable representing the number of previous software versions we have seen a pattern instance in. This makes each of our ANOVAs a four-way ANOVA, but to generalize this model one would include the factors they had captured from their specific circumstance. Because of our intuition that interacting factors likely exist in our data, i.e. that behavioral grime is contingent on the combination of Project and Pattern Type, and the fact we have a large sample size, we chose to consider several combinations of interactions in our ANOVA. Specifically, the interaction terms we considered were Project and Pattern Type, Project and Pattern Age, Pattern Type and Pattern Age, and the three-way interaction between Project, Pattern Type, and Pattern Age. We chose not to look at interaction terms involving Software Version because (1) such interactions do not have a meaningful explanation in the real world, and (2) it increases model complexity to a point where computational power starts to become questioned. In terms of assumptions, we assessed the independence assumption, the normality assumption, the equal variance assumption, and the homogeneity assumption.

We found that there were no major violations of any assumption for each ANOVA. The final model we used is presented below:

$$Y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + \delta_l + (\alpha\beta)_{ij} + (\alpha\delta)_{il} + (\beta\delta)_{jl} + \alpha\beta\delta_{ijl} + \varepsilon_{ijkl}$$

where:

$Y_{ijkl}$ refers to the measurement of behavioral grime with levels i, j, k, l

$\mu$ refers to the grand mean.

$\alpha_i$ refers to the ith level of the Project variable.

$\beta_j$ refers to the jth level of the Pattern Type variable.

$\gamma_k$ refers to the kth level of the Software Version variable.

$\delta_l$ refers to the lth level of the Pattern Age variable.

$\varepsilon_{ijkl}$ refers to the random error present in the data.

Table 6.9 shows the results from our ANOVA models, listing the F-values in the cells of the table. We chose to do this because the F-values provide more insight into the data than simple *p*-values, because many of our *p*-values are incalculably small. However, we have emboldened F-values that have corresponding statistically significant *p*-values < 0.05.

Table 6.9 Results from ANOVA models. Behavioral grime types are shown on the rows, and ANOVA model terms (including interaction terms) are shown on the columns. Cells show the F-value corresponding to the variance the column term takes on the given behavioral grime type. Emboldened F-values represent statistically significant p-values $p < 0.05$.

| Behavioral Grime Model | Project | Pattern Type | Software Version | Pattern Age | Project: Pattern Type | Project: Pattern Age | Pattern Type: Pattern Age | Project: Pattern Type: Pattern Age |
|---|---|---|---|---|---|---|---|---|
| PEAO grime | **97.653** | **159.102** | 1.442 | **46.852** | **36.524** | **12.944** | **7.331** | **5.217** |
| PIO grime | **2.785** | **7.257** | 0.687 | 1.484 | **2.379** | 0.326 | 0.864 | 0.276 |
| TEAO grime | **46.944** | **58.613** | 0.225 | 1.948 | **18.234** | 0.974 | 1.239 | 0.371 |
| TIO grime | **78.910** | **199.884** | 0.055 | **10.626** | **35.054** | **5.761** | **2.677** | **1.930** |
| PEAR grime | **43.219** | **150.872** | 0.403 | **8.672** | **27.077** | **2.194** | **2.443** | 0.967 |
| PEER grime | **135.943** | **186.576** | **1.569** | 0.622 | **54.096** | **5.062** | **2.364** | **5.124** |
| PIR grime | **71.393** | **458.892** | 0.175 | **27.677** | **30.901** | **2.984** | **4.4345** | **2.322** |
| TEAR grime | **6.084** | **29.763** | 0.484 | 0.389 | **4.082** | 0.765 | 0.161 | 0.529 |
| TEER grime | **106.499** | **89.940** | 0.178 | 2.166 | **16.141** | **4.206** | 0.260 | 1.030 |
| TIR grime | **78.910** | **199.884** | 0.055 | **10.626** | **35.054** | **5.761** | **2.677** | **1.930** |

The results from our ANOVAs in table 6.9 provide several insightful glimpses into our exploration of the terms that dictate the presence of behavioral grime. First, three model terms accounted for a statistically significant amount of variance for every behavioral grime type; Project, Pattern Type, and the interaction between Project ID and Pattern Type. Particularly, Pattern Type accounted for the most variance in the model in all cases except for TEER grime, followed by Project, and finally the interaction between the two. This means that the type of design pattern nearly always accounts for more variance in the data with respect to behavioral grime than the software project. Though, the software project still accounts for a large and significant amount of the variance in the data as well. The interaction between the two, which explained in less statistically technical terms translates to 'the combination of the design pattern type and the software project that the design pattern instance exists in', also accounts for a significant amount of variance in the data, but less so than either one of its two building terms. The Software Version term accounted for a statistically significant amount of variance for the PEE (Persistent External Efferent) grime type, but it was much lower of a contribution than the other terms. Pattern Age accounted for a significant amount of variance in half of the models, specifically the PEAO, TIO, PEAR, PIR, and TIR grime models. This means that Pattern Age may play an important role when considering the presence of PEAO, TIO, PEAR, PIR, or TIR grime, but the ANOVA results only yield explanations of the variance in the data, and do not provide any causative or predictive power. Finally, the interaction terms including Pattern Age frequently accounted for a statistically significant

amount of variance in the data, but these yielded low F-values respective to the individual terms that make up the interaction terms.

The results from the ANOVAs table are insightful, but they themselves do not provide an equation that explains the rate at which behavioral grime appears in patterns. Rather, the next step in our process to answer this research question involves utilizing the results from our ANOVAs table, table 6.9, to generate regressions that explain the data. While seeming applicable, time series analysis does not apply to our data because classical time series analysis considers one sampling of data across a series of time, and our data contains many replicates. Additionally, our variables related to time, specifically Software Version and Pattern Age, are categorical variables and do not represent equally continuous spaces of time between each measurement; rather, they represent discrete points with the potential of having little direct relation to the previous measurement, because it is impossible to glean what has occurred between two successive measurements. Instead of time series analysis, we perform regression analysis, the first step of which is to generate plots to visually assess the nature of relationships between behavioral grime counts and the statistically significant terms in our models. These plots are presented in figure 6.11, with two exceptions. First, we have not shown Order grime in these plots because of the few instances of Order grime. Most plots with Order grime showed little counts of behavioral grime. Second, we have chosen to exclude pattern instances that have behavioral grime counts of zero in these plots alone. Because of the large number of pattern instances with zero behavioral grime counts, it was very difficult to distinguish between trendlines, representing the mean of the data. These cases are

excluded from the plots only, and only to aide in visual inspection. The final model we fit does include these cases where the behavioral grime count is zero.

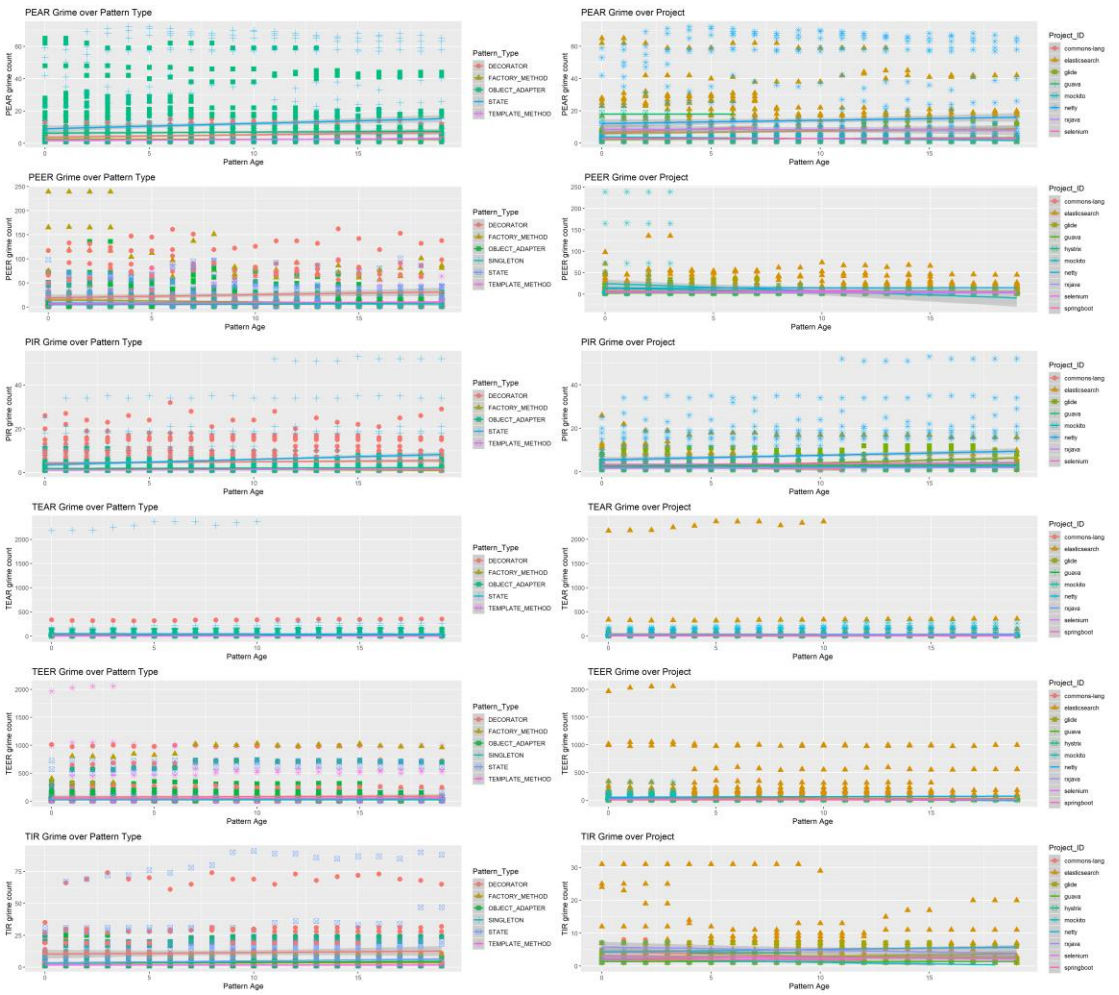


Figure 6.11 Counts of Repetition behavioral grime over Pattern Age, across the Pattern Type factor on the left column and the Project factor on the right column. Every point within the plots refers to a singular pattern instance. To generate visually clear plots in which the linear regression lines can be clearly seen, we have chosen to exclude pattern instances that have behavioral grime counts of zero. Our linear model fitting, explained in section 6.4.7, does include these cases, but for visual aesthetics they are excluded here.

A visual inspection of the plots shows a lot of variance in the data, but largely linear relationships between grime count and both of Pattern Type and Project. Because of these reasons, we choose to fit a linear regression model from the data, considering the

terms Pattern Type, Project, and Pattern Age. Note that we understand Pattern Age is a categorical variable, yet we treat it as if it were a continuous variable to keep the resulting model simple. Though with this decision, we do not make interpolation or extrapolation claims with Pattern Age, and we address this decision in the Threats to Validity section, section 6.6. For similar reasons, we chose not to include the interaction terms because of the increased complexity every interaction term would add to the model; with 7 Pattern Types and 10 Projects we would have 69 additional model coefficients. Note that one combination of Pattern Type and Project would be captured by the intercept term within the model, so we would only generate 69 coefficients. The results from our linear regression analysis is shown in tables 6.10 through 6.17, with each table referring to a single type of behavioral grime, each row representing the variable term from the model, and each column referring to the statistical estimate of corresponding model term. We have elected to exclude tables for PIO and TIO grime because they featured very low counts of grime, and the estimates we calculated from them are inaccurate and therefore provide limited value. Furthermore, the first row showcasing 'Intercept' refers to the case capturing the Decorator Pattern Type in the commons-lang Project.

Table 6.10 shows the results from our fitted model for PEAO grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: "The rate at which PEAO occurs in the elasticsearch project in Factory-Method pattern instances is equal to 0.002 + (-0.010) + (-0.009) + (Pattern Age * 0.0007). Generally speaking, we found very few instances of PEAO grime

so the estimates are near zero. Many of them are negative, suggesting that PEAO grime is less in contexts corresponding to the row name the coefficient comes from. Though the Intercept, which represents the commons-lang Project and the Decorator Pattern Type, and the (Object) Adapter, Observer, and State Pattern Types, and the netty project provided positive estimates, suggesting that in these contexts PEAO grime is higher. Pattern Age is slightly positive, demonstrating that PEAO grime increases as a pattern ages. Ultimately, these low estimates are likely due to the low count of PEAO grime instances we identified in our study; future studies are needed to assert the strength of these findings.

Table 6.10 PEAO grime model linear regression estimates. p-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05)

| Variable | Coefficient | Standard Error | t-value | *p*-value |
|---|---|---|---|---|
| Intercept | 0.002 | 0.008 | 0.31 | 0.75989 |
| elasticsearch | -0.010 | 0.007 | -1.29 | 0.19778 |
| glide | -0.019 | 0.008 | -2.28 | **0.02257** |
| guava | -0.010 | 0.007 | -1.27 | 0.20358 |
| hystrix | -0.010 | 0.010 | -1.03 | 0.30436 |
| mockito | -0.019 | 0.008 | -2.28 | **0.02253** |
| netty | 0.028 | 0.007 | 3.69 | **0.00022** |
| rxJava | -0.040 | 0.008 | -5.01 | **5.5e-07** |
| selenium | -0.025 | 0.008 | -3.01 | **0.00265** |
| springboot | -0.010 | 0.009 | -1.08 | 0.28162 |
| Factory Method | -0.009 | 0.004 | -2.22 | **0.02655** |
| (Object) Adapter | 0.001 | 0.004 | 0.26 | 0.79716 |
| Observer | 0.010 | 0.028 | 0.38 | 0.70057 |
| Singleton | -0.007 | 0.004 | -1.91 | **0.05649** |
| State | 0.045 | 0.003 | 11.64 | <**2e-16** |
| Template Method | -0.004 | 0.004 | -1.16 | 0.24568 |
| Pattern Age | 0.0007 | 0.0001 | 5.19 | **2.1e-07** |

Table 6.11 shows the results from our fitted model for TEAO grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: “The rate at which TEAO occurs in the elasticsearch project in Factory Method pattern instances is equal to 0.033 + (-0.137) + (-0.107) + (Pattern Age * 0.005). Similarly to PEAO grime, we found very few instances of TEAO grime, so many of the estimates are near zero. Additionally, many of them are negative, suggesting that TEAO grime is less in contexts corresponding to the row name the coefficient comes from. However, and identical to PEAO grime, the (Object) Adapter, Observer, and State Pattern Types, and the netty project provided positive estimates, suggesting that in these contexts TEAO grime is higher. Note that the Intercept, which represents the commons-lang Project and the Decorator Pattern Type, was also positive. Pattern Age is slightly positive, demonstrating that TEAO grime increases as a pattern ages. Similarly to PEAO grime, these low estimates are likely due to the low count of TEAO grime instance we identified in our study; future studies are needed to assert the strength of these findings.

Table 6.11 TEAO grime model linear regression estimates. P-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05).

| Variable | Coefficient | Standard Error | t-value | p-value |
|---|---|---|---|---|
| Intercept | 0.033 | 0.145 | 0.23 | 0.8156 |
| elasticsearch | -0.137 | 0.133 | -1.03 | 0.3016 |
| glide | -0.202 | 0.146 | -1.39 | 0.1652 |
| guava | -0.099 | 0.137 | -0.73 | 0.4684 |
| hystrix | -0.110 | 0.175 | -0.63 | 0.5281 |
| mockito | -0.201 | 0.144 | -1.39 | 0.1639 |
| netty | 0.371 | 0.134 | 2.77 | **0.0056** |
| rxJava | -0.420 | 0.138 | -3.04 | **0.0024** |
| selenium | -0.263 | 0.145 | -1.81 | **0.0700** |
| springboot | -0.103 | 0.159 | -0.65 | 0.5166 |
| Factory Method | -0.107 | 0.075 | -1.43 | 0.1536 |
| (Object) Adapter | 0.019 | 0.071 | 0.27 | 0.7839 |
| Observer | 0.122 | 0.487 | 0.25 | 0.8015 |
| Singleton | -0.075 | 0.068 | -1.10 | 0.2718 |
| State | 0.486 | 0.067 | 7.23 | **4.8e-13** |
| Template Method | -0.052 | 0.071 | -0.73 | 0.4625 |
| Pattern Age | 0.005 | 0.002 | 2.37 | **0.0179** |

Table 6.12 shows the results from our fitted model for PEAR grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: “The rate at which PEAR occurs in the elasticsearch project in Factory Method pattern instances is equal to 1.368 + (-0.298) + (-1.439) + (Pattern Age * 0.020). We found many more instances of PEAR grime than PEAO or TEAO grime, and therefore our estimates are generally larger than the estimates for either PEAO or TEAO grime. Though, many of them are negative, suggesting that PEAR grime is less in contexts corresponding to the row name the coefficient comes from. Though, the positive Intercept representing the commons-lang Project and the Decorator Pattern

Type offsets some of these negative values. Similar to PEAO and TEAO grime, the (Object) Adapter, and State Pattern Types, and the netty project provided positive estimates, suggesting that in these contexts PEAR grime is higher. Pattern Age is slightly positive, illustrating that PEAR grime slowly increases as a pattern ages.

Table 6.12 PEAR grime model linear regression estimates. p-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05).

| Variable | Coefficient | Standard Error | t-value | p-value |
|---|---|---|---|---|
| Intercept | 1.368 | 0.393 | 3.47 | **0.0005** |
| elasticsearch | -0.298 | 0.360 | -0.82 | 0.407 |
| glide | -1.144 | 0.394 | -2.89 | **0.003** |
| guava | -0.556 | 0.370 | -1.50 | 0.132 |
| hystrix | -0.518 | 0.473 | -1.09 | 0.273 |
| mockito | -0.978 | 0.390 | -2.50 | **0.012** |
| netty | 0.812 | 0.362 | 2.24 | **0.025** |
| rxJava | -1.364 | 0.373 | -3.65 | **0.0002** |
| selenium | -1.078 | 0.393 | -2.73 | **0.006** |
| springboot | -0.832 | 0.431 | -1.93 | **0.053** |
| Factory Method | -1.439 | 0.202 | -7.09 | **1.33e-12** |
| (Object) Adapter | 1.218 | 0.193 | 6.29 | **3.21e-10** |
| Observer | -0.400 | 1.319 | -0.30 | 0.761 |
| Singleton | -1.434 | 0.185 | -7.71 | **1.29e-14** |
| State | 0.627 | 0.181 | 3.45 | **0.0005** |
| Template Method | -1.235 | 0.193 | -6.38 | **1.76e-10** |
| Pattern Age | 0.020 | 0.006 | 3.35 | **0.0008** |

Table 6.13 shows the results from our fitted model for PEER grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: “The rate at which PEER occurs in the elasticsearch project in Factory Method pattern instances is equal to 9.952 + (-0.601) + (-6.713) + (Pattern Age * 0.031). Our coefficients are generally larger than the estimates for Order

grime, and are larger than PEAR grime estimates. Though, many of them are negative, suggesting that PEER grime is less in the contexts corresponding to the row name the coefficient comes from. Similarly to PEAR grime, the large and positive Intercept offsets many of these negative coefficients. Aside from the Intercept which represents the commons- lang Project and the Decorator Pattern Type, only two contexts provided positive coefficient estimates, which were the hystix and netty projects. This suggests that PEER grime is higher in these contexts. Interestingly, Pattern Age is slightly negative, asserting that PEER grime slowly decreases as a pattern ages.

Table 6.13 PEER grime model linear regression estimates. p-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05).

| Variable | Coefficient | Standard Error | t-value | p-value |
|---|---|---|---|---|
| Intercept | 9.952 | 0.679 | 14.65 | **2e-16** |
| elasticsearch | -0.601 | 0.621 | -0.97 | 0.33306 |
| glide | -1.445 | 0.681 | -2.12 | **0.03400** |
| guava | -2.484 | 0.639 | -3.89 | **0.00010** |
| hystrix | 0.147 | 0.817 | 0.18 | 0.85728 |
| mockito | -1.865 | 0.674 | -2.77 | **0.00564** |
| netty | 2.522 | 0.625 | 4.03 | **5.6e-05** |
| rxJava | -3.911 | 0.645 | -6.06 | **1.4e-09** |
| selenium | -2.692 | 0.679 | -3.96 | **7.4e-05** |
| springboot | -2.542 | 0.744 | -3.41 | **0.00064** |
| Factory Method | -6.713 | 0.350 | -19.18 | **2e-16** |
| (Object) Adapter | -5.386 | 0.334 | -16.11 | **2e-16** |
| Observer | -8.243 | 2.276 | -3.62 | **0.00029** |
| Singleton | -9.133 | 0.321 | -28.46 | **2e-16** |
| State | -5.672 | 0.313 | -18.09 | **2e-16** |
| Template Method | -7.384 | 0.334 | -22.11 | **2e-16** |
| Pattern Age | -0.031 | 0.010 | -2.91 | **0.00364** |

Table 6.14 shows the results from our fitted model for PIR grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for

the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: “The rate at which PIR occurs in the elasticsearch project in Factory Method pattern instances is equal to 3.442 + (-0.070) + (-3.579) + (Pattern Age * 0.009). Our coefficients relating to Project are similar in size to the estimates for Order grime, but are much smaller than PEAR or PEER grime estimates. However, the coefficients relating to Pattern Type are larger. This suggests that Project Type holds more weight when providing PIR grime rates. Many coefficient estimates are negative, suggesting that PIR grime is less in the contexts corresponding to the row name the coefficient comes from. Similarly to PEAR and PEER grime, the large and positive Intercept offsets many of these negative coefficients. Aside from the Intercept which represents the commons-lang Project and the Decorator Pattern Type, only two contexts provided positive coefficient estimates, which were the netty and selenium projects. This suggests that in these contexts PIR grime is higher. Pattern Age is slightly positive, demonstrating that PIR grime slowly increases as a pattern ages.

Table 6.14 PIR grime model linear regression estimates. p-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05).

| Variable | Coefficient | Standard Error | t-value | p-value |
|---|---|---|---|---|
| Intercept | 3.442 | 0.160 | 21.42 | **2e-16** |
| elasticsearch | -0.070 | 0.147 | -0.48 | 0.6303 |
| glide | -0.068 | 0.161 | -0.43 | 0.6692 |
| guava | -0.144 | 0.151 | -0.95 | 0.3405 |
| hystrix | -0.128 | 0.193 | -0.66 | 0.5064 |
| mockito | -0.313 | 0.159 | -1.97 | **0.0493** |
| netty | 0.458 | 0.148 | 3.10 | **0.0019** |
| rxJava | -0.702 | 0.152 | -4.60 | **4.2e-06** |
| selenium | 0.261 | 0.160 | 1.63 | 0.1032 |
| springboot | -0.319 | 0.176 | -1.81 | **0.0697** |
| Factory Method | -3.579 | 0.082 | -43.22 | **2e-16** |
| (Object) Adapter | -3.129 | 0.079 | -39.58 | **2e-16** |
| Observer | -3.458 | 0.538 | -6.42 | **1.4e-10** |
| Singleton | -3.575 | 0.075 | -47.09 | **2e-16** |
| State | -2.684 | 0.074 | -36.19 | **2e-16** |
| Template Method | -3.470 | 0.079 | -43.91 | **2e-16** |
| Pattern Age | 0.009 | 0.002 | 3.95 | **7.7e-05** |

Table 6.15 shows the results from our fitted model for TEAR grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: "The rate at which TEAR occurs in the elasticsearch project in Factory Method pattern instances is equal to 9.705 + (2.40) + (-9.966) + (Pattern Age * 0.001). Our coefficients are much larger than the coefficients capturing Order grime, and are similar in value to the coefficients for Persistent Repetition grime. Many coefficient estimates are negative, suggesting that TEAR grime is less in the contexts corresponding to the row name the coefficient comes from. Similarly to the forms of Persistent Repetition grime, the large and positive Intercept corresponding to the

commons-lang Project and the Decorator Pattern Type offsets many of these negative coefficients. Aside from the Intercept, only two contexts provided positive coefficient estimates, which were the elasticsearch and netty projects. This suggests that in these contexts TEAR grime is higher. Pattern Age is very slightly positive, demonstrating that TEAR grime slowly increases very slowly as a pattern ages.

Table 6.15 TEAR grime model linear regression estimates. p-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05).

| Variable | Coefficient | Standard Error | t-value | p-value |
|---|---|---|---|---|
| Intercept | 9.705 | 3.579 | 2.71 | **0.0067** |
| elasticsearch | 2.400 | 3.275 | 0.73 | 0.4636 |
| glide | -4.452 | 3.591 | -1.24 | 0.2151 |
| guava | -1.532 | 3.369 | -0.45 | 0.6492 |
| hystrix | -1.238 | 4.308 | -0.29 | 0.7737 |
| mockito | -4.176 | 3.551 | -1.18 | 0.2396 |
| netty | 2.106 | 3.297 | 0.64 | 0.5228 |
| rxJava | -5.539 | 3.400 | -1.63 | 0.1033 |
| selenium | -5.044 | 3.579 | -1.41 | 0.1587 |
| springboot | -2.017 | 3.923 | -0.51 | 0.6070 |
| Factory Method | -9.966 | 1.844 | -5.40 | **6.6e-08** |
| (Object) Adapter | -5.208 | 1.761 | -2.96 | **0.0031** |
| Observer | -5.265 | 11.996 | -0.44 | 0.6607 |
| Singleton | -10.879 | 1.691 | -6.43 | **1.3e-10** |
| State | -0.611 | 1.652 | -0.37 | 0.7112 |
| Template Method | -9.590 | 1.760 | -5.45 | **5.1e-08** |
| Pattern Age | 0.001 | 0.056 | 0.03 | 0.9798 |

Table 6.16 shows the results from our fitted model for TEER grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: "The rate at which TEER occurs in the elasticsearch project in Factory Method pattern instances is equal to 41.631 + (2.739) + (-31.750) +

(Pattern Age * 0.017). Our coefficients are the largest coefficients for all forms of grime we identified, which is not a surprise considering TEER grime was the most widely encountered. Many coefficient estimates are negative, suggesting that TEER grime is less in the contexts corresponding to the row name the coefficient comes from. Similarly to the forms of Persistent Repetition grime, the large and positive Intercept corresponding to the commons-lang Project and the Decorator Pattern Type offsets many of these negative coefficients. Aside from the Intercept, only two contexts provided positive coefficient estimates, which were the elasticsearch and netty projects. This suggests that in these contexts TEER grime is higher in these projects. Pattern Age is very slightly positive, demonstrating that TEER grime slowly increases very slowly as a pattern ages.

Table 6.16 TEER grime model linear regression estimates. p-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05).

| Variable | Coefficient | Standard Error | t-value | p-value |
|---|---|---|---|---|
| Intercept | 41.631 | 5.433 | 7.66 | **1.9e-14** |
| elasticsearch | 4.739 | 4.972 | 0.95 | 0.34049 |
| glide | -18.853 | 5.452 | -3.46 | **0.00055** |
| guava | -15.067 | 5.114 | -2.95 | **0.00322** |
| hystrix | -5.593 | 6.540 | -0.86 | 0.39238 |
| mockito | -13.407 | 5.391 | -2.49 | **0.01290** |
| netty | 23.159 | 5.005 | 4.63 | **3.7e-06** |
| rxJava | -21.977 | 5.162 | -4.26 | **2.1e-05** |
| selenium | -12.757 | 5.433 | -2.35 | **0.01889** |
| springboot | -12.185 | 5.956 | -2.05 | **0.04078** |
| Factory Method | -31.750 | 2.800 | -11.34 | **2e-16** |
| (Object) Adapter | -18.379 | 2.673 | -6.87 | **6.4e-12** |
| Observer | -22.922 | 18.211 | -1.26 | 0.20816 |
| Singleton | -41.871 | 2.567 | -16.31 | **2e-16** |
| State | -16.160 | 2.508 | -6.44 | **1.2e-10** |
| Template Method | -22.893 | 2.672 | -8.57 | **2e-16** |
| Pattern Age | 0.017 | 0.085 | 0.20 | 0.84198 |

Table 6.17 shows the results from our fitted model for TIR grime. As an example of how to glean a rate from this table, consider the Coefficient values from the table for the rows Intercept, elasticsearch, Factory Method, and Pattern Age. These estimates provide the statement: “The rate at which TIR occurs in the elasticsearch project in Factory Method pattern instances is equal to 3.845 + (-0.457) + (-3.489) + (Pattern Age * 0.007)”. The scalar values of our coefficients are small for Repetition grime, and on par with Order grime coefficients. Many coefficient estimates are negative, suggesting that TIR grime is less in the contexts corresponding to the row name the coefficient comes from. Similarly to the other forms of Repetition grime, the relatively large and positive Intercept corresponding to the commons-lang Project and the Decorator Pattern Type offsets many of these negative coefficients. Aside from the Intercept, only one context provided positive coefficient estimates, which was the netty project. This suggests that TIR grime is higher in the netty project. Pattern Age is slightly positive, demonstrating that TIR grime slowly increases slowly as a pattern ages.

Table 6.17 TIR grime model linear regression estimates. p-values, which test the null hypothesis that the estimate is equal to zero, are shown emboldened if they are statistically significant (< 0.05).

| Variable | Coefficient | Standard Error | t-value | p-value |
|---|---|---|---|---|
| Intercept | 3.845 | 0.263 | 14.61 | **2e-16** |
| elasticsearch | -0.457 | 0.240 | -1.90 | **0.05767** |
| glide | -0.939 | 0.264 | -3.55 | **0.00038** |
| guava | -0.567 | 0.247 | -2.29 | **0.02195** |
| hystrix | -0.400 | 0.316 | -1.26 | 0.20675 |
| mockito | -1.116 | 0.261 | -4.27 | **1.9e-05** |
| netty | 0.711 | 0.242 | 2.93 | **0.00337** |
| rxJava | -0.918 | 0.250 | -3.67 | **0.00024** |
| selenium | -0.972 | 0.263 | -3.69 | **0.00022** |
| springboot | -0.507 | 0.288 | -1.76 | **0.07865** |
| Factory Method | -3.489 | 0.135 | -25.71 | **2e-16** |
| (Object) Adapter | -2.718 | 0.129 | -20.98 | **2e-16** |
| Observer | -2.971 | 0.882 | -3.37 | **0.00076** |
| Singleton | -3.718 | 0.124 | -29.89 | **2e-16** |
| State | -2.468 | 0.121 | -20.30 | **2e-16** |
| Template Method | -3.482 | 0.129 | -26.90 | **2e-16** |
| Pattern Age | 0.007 | 0.004 | 1.85 | **0.06504** |

The results from tables 6.10 through 6.17 illustrates one important finding that is not immediately revealed through the ANOVAs tables in table 6.9. Specifically, as expected from our ANOVAs table analysis shown in table 6.9, the Project and Pattern Type variables dictated the count of behavioral grime much more strongly than Pattern Age. In fact, the coefficient values we calculated for Pattern Age were consistently very small, with the largest being from PEAR grime in table 6.12 with a value of 0.020. Practically speaking, this means that ignoring the Project and Pattern Type, a pattern instance would have to age through roughly 50 versions before it saw a single instance of PEAR grime. This is in our highest Pattern Age coefficient too; lower Pattern Age coefficients imply cases where a pattern instance would have to age longer than this

predicted value to generate a single instance of behavioral grime. This finding illustrates that the Project and Pattern Type have much more bearing on how grime is added or removed from a pattern instance. An interesting side-note is that in the PEER grime table, table 6.13, the estimated coefficient for Pattern Age is negative. This suggests that PEER pattern grime generally is removed as a pattern ages. From a quality perspective, PEER grime can be considered an especially unwanted type of grime because it refers to cases where a pattern class contains an association to a non-pattern class. As associations are stronger relationships than use-dependencies, it would be preferable to remove Persistent forms of grime for the equivalent Temporary form of grime, if the situation allows. The data reflects this thought, for PEER grime types.

### 6.4.8 RQ8

Our eighth research question considers if state of the art software quality analysis tools are capable of identifying behavioral grime. This is the first step in our exploration of how design pattern grime affects pattern and system quality. Our rationale is as follows: if state of the art software quality analysis tools are capable of detecting and measuring behavioral grime, the results from such tools would heavily complement the findings from this study. If these tools are not capable of such, it is required to extend upon those tools to capture an understanding of how behavioral grime affects system quality. To assess this research problem, we performed a systematic search to identify state of the art software analysis tools that perform behavioral analysis. The systematic approach that involved searching popular search engines for terms including the keywords ‘software quality analysis tools’. For each tool we identified, we searched

features of the tool on their website, specifically looking for terms that indicate behavioral analysis, such as ‘Behavioral Analysis’. We reason that if a tool performs some form of behavioral analysis similar to what we perform in this study, they would be quick to generalize the capabilities of their tool, and therefore would label their form of analysis ‘Behavioral Analysis’. The findings from this study is presented in table 6.18.

Table 6.18 Summary of results from RQ8, which considers if state of the art software quality analysis tools are capable of identifying and/or measuring behavioral grime.

| Tool name | Tool description | Behavioral Analysis? |
|---|---|---|
| FindBugs | Static analysis tool to look for bugs | No |
| PMD | An extensible cross-language static analyzer | No |
| SonarQube | Your teammate for Software Quality and Security | No |
| QATCH[6] | An Adaptive framework for software product quality model assessment | No |
| Understand | Visualize your code | No |
| Parasoft | Automated Software Testing Tools for Creating High Quality Software | No |
| Coverity | Find and fix security and quality issues as you code, fast | No |

The results from table 6.18 indicate that none of the software quality analysis tools we identified from our search are capable of performing behavioral analysis. Nearly all tools advertised ‘static code analysis’ as their primary feature, of which behavioral analysis is the corollary to. To verify the results from this study, we performed an *in-vitro* experiment [40] wherein we tested the tool SonarQube [32] on a pattern instance

evolution. This pattern instance evolution is the same one used to answer RQ1 in section 6.4.1, in which we started with a perfect implementation of an Observer pattern, with perfect conformance to specifications and no grime. Recall that to this pattern instance we simulated design pattern evolution by injecting one singular form of grime, which constituted a version. We removed each injected grime instance in the next version to ensure that the presence of pattern grime was the only variable that was changing. We evaluated the quality and technical debt of this pattern instance evolution using the tool SonarQube, and found that the changing quality of each pattern instance across versions was insignificant, and any deviation was due to the addition or removal of lines of code, which serve as a normalization value in SonarQube results. This means that SonarQube, which does not claim to perform behavioral analysis, is incapable of performing behavioral analysis. However, this result illustrates a gap in the state of the art, specifically that these tools are incapable of performing behavioral analysis. This means the next steps of this research will seek to extend existing quality models to provide behavioral analysis capabilities, which can ultimately be incorporated into these state-of-the-art quality tools to provide more useful results to practitioners and stakeholders alike.

### 6.4.9 RQ9

Research question 9 considers how the ISO 25010 software quality specification [36] can be implemented such that a resulting operational model includes design pattern grime in its calculations. Recall from section 6.2.1 that the ISO 25010 software product quality model consists of eight primary quality characteristics at its highest level of abstraction, and each of those eight quality characteristics has a set of quality properties

that represent the grounding of various properties that are important from an implementation perspective. The problem of extending and implementing the ISO 25010 software quality model involves identifying code-level violations and mapping them to the various quality properties, which in term map to the quality characteristics. We have elected to perform this mapping process in the same manner as the tool QATCH [70], which provides a holistic approach to the mapping process and greater quality calculations, in a simple manner that ensures the resulting model is not too complex that it distracts from the value it offers. Models that fit into this category of being too complex, such QUAMOCO [76], rarely see practical use because of the complicated nature of the mapping process, which requires in-depth and manual calibration involving connecting each code-level measurement to each software quality property or characteristic, as well as manually specifying the weight that each measurement has on each quality entity. The tool QATCH avoids this issue by requiring stakeholders to specify fuzzy levels of importance for each quality property, characteristic, and code-level measurement in a table format, and automatically performing the mapping process based on its holistic assumption. See figure 6.12 for a visual example of the QATCH model, which is taken from [70]. As an example illustrating how the mapping process occurs, a user of QATCH specifies that a code-level measurement A is twice as impactful as code-level B measurement on the quality property C, and therefore will generate weights between A and C, and B and C, that illustrate this relationship. The mathematical process that is responsible for generating the weights involves a Fuzzy Logarithmic Least Squares method wherein each entity (measurement, property, or characteristic) at each

layer in the quality hierarchy is ranked according to the other entities at that layer, and mapping weights are assigned based on the lower quartile, median, and upper quartile values that the entity was calculated, such that the sum of all weights going into the next-higher level of the hierarchy sums to 1. The utilization of this model to capture system quality assures that an extension has no effect on the default model's code-level measurement calculations, and only affects the edge weights that make up the calculation for next-highest level of the overall hierarchy. In other words, the effect of each code-level measurement from the default QATCH model on each quality property and characteristic remains the same, yet we allow for the incorporation of design pattern grime measurements such that we can understand the effects of design pattern grime on each quality property and characteristic.

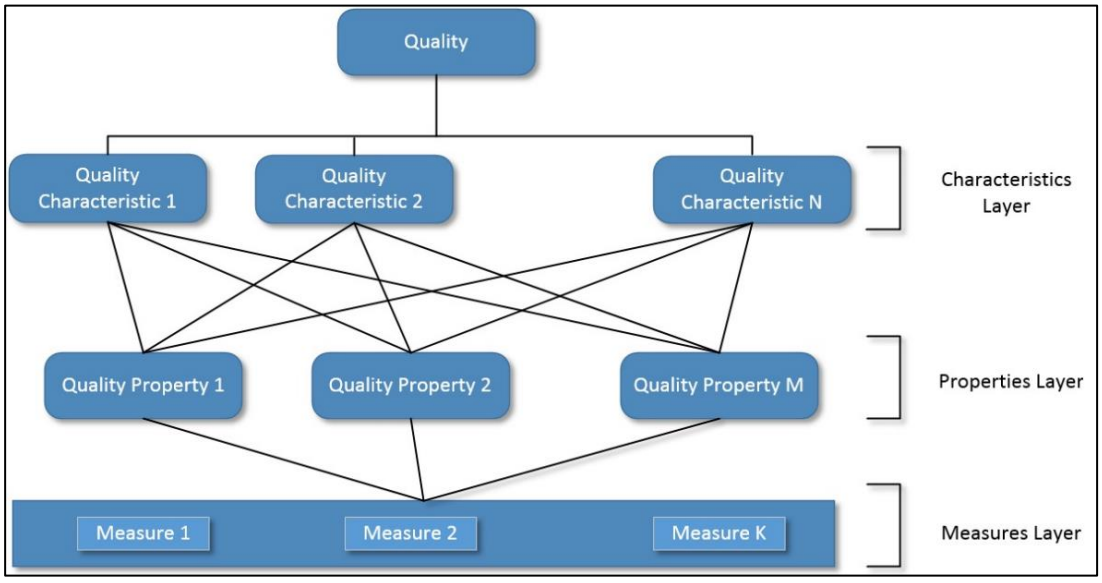


Figure 6.12 Visual illustration of the QATCH model, image taken from [70].

To perform this model extension, we first need to select metrics that summarize the code-level measurements for design pattern grime we have computed. This step is

important for two reasons. First, it ensures the resulting model is not too complex that it will preserve the simplicity property, which was a major reason for choosing QATCH in the first place. Specifically, we have 16 code-level measurements for design pattern grime, one for each form of grime including both structural and behavioral, and the inclusion of all 16 code-level measurements would substantially increase the number of quality rankings an end-user would need to perform. This increase is on the order of thousands of more rankings, because all metrics need to be pairwise ranked against all other metrics, and for each quality characteristic under consideration, which is eight according to the ISO 25010 specification [36]. Second, the default QATCH model hierarchy consists of metrics that summarize code-level measurements, so for consistency sake we also build metrics that summarize design pattern grime. The metrics we selected to summarize design pattern grime are presented in table 6.19.

Table 6.19 Summarization metrics we have selected for the QATCH model extension.

| Metric name | Description | Range | Relationship with Quality |
|---|---|---|---|
| Pattern Structural Integrity | M1 from table 3.1 | [0,1] | direct |
| Pattern Behavioral Integrity | M2 from table 3.1 | [0,1] | direct |
| Pattern Instability | M6 from table 3.1 | [0,1] | inverse |
| Pattern Structural Aberrations | Count of occurrences of all structural grime in a pattern instance divided by the pattern size. | [0, ∞] | inverse |
| Pattern Behavioral Aberrations | Count of occurrences of all behavioral grime in a pattern instance divided by the pattern size. | [0, ∞] | inverse |

Table 6.19 also shows the expected relationship each metric takes with quality via the column ‘Relationship with Quality’. In other words, this column signifies how quality changes as the metric’s measurement changes. A *direct* relationship implies that both increase or decrease together, while an *inverse* relationship implies the opposite happens; as the measurement increases, quality decreases, or vice versa. We selected values for the relationship with quality borrowing from analogous Object-Oriented system metrics, such as Pattern Instability being the design pattern equivalent of Instability from [50], as well as domain knowledge of the metrics, i.e., Pattern Structural and Behavioral Aberrations embody negative and unexpected additions to a design pattern, therefore are inversely related to quality. In terms of integration into the QATCH model, each metric from table 18 fits into the ’Properties’ layer, representing their own node, presented from figure

6.12. Because of the simplicity of QATCH, nothing else from the base model needs to be changed; therefore we ensure this extension has no effect on the default model's code-level measurement calculations, and corresponding quality properties. However, because of the holistic nature of QATCH, the values of the quality characteristics and greater quality measurement will be changed, but that change indicates the extension has been completed correctly.

### 6.4.10 RQ10

Research question 10 is concerned with identifying the relationship between behavioral grime and system quality and TD. In essence, this question involves calculating system quality using the QATCH model [70] for each version of each project under analysis, and applying a correlation analysis to quality and design pattern behavioral grime. A key point here is that because of QATCH's holistic nature, which provides a better estimation of system quality than piece-wise analysis, we do not capture the quality or grime counts of individual pattern instances. Rather, we use aggregation measurements to summarize quality and grime counts of individual pattern instances across projects and versions. In other words, each project at each version will have one measurement for both quality and pattern behavioral grime. Before the QATCH model can be applied to these data, two important steps are necessary; ranking quality entities and model calibration.

6.4.10.1 Ranking Quality Entities. The process of ranking quality entities involves specifying the importance for each Property and Characteristic entity presented in figure

6.12, compared to all other entities a that layer, to make up the later above it. That is, for each layer in the QATCH model we need to specify importance of each entity **in the layer below it**. For the Property layer, we consider all Measurements of the QATCH default model, including (Bad Functionality, Comprehensibility, Redundancy, Structuredness, Assignment, Resource Handling, Cohesion, Coupling, Complexity, Messaging, and Encapsulation) and our extension Measurements presented from research question 9 in section 6.4.9 (Pattern Structural Integrity, Pattern Behavioral Integrity, Pattern Instability, Pattern Structural Aberrations, Pattern Behavioral Aberrations). The Measurements from the QATCH default model come from the PMD ruleset list[21] and the CKJM (Chidamber and Kemerer Java Metrics[16]) extended metric package[22]. For the Characteristics layer, we properly exemplify the ISO-25010 specification by considering all eight quality properties (Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability) from figure 6.1.

We complete the ranking process in a bottom-up approach, starting with the Measurements so that we can complete the Property layer. To complete the rankings, we utilized the default QATCH rankings as well as our domain knowledge of design pattern grime. The Property rankings are presented in appendix A, figures 6.16a and 6.16b, with Characteristics shown on the left side of each line, and each Property's ranked importance is shown on top of each line. Tick lines indicate when multiple Properties share the same importance. Once the Properties layer has been completed, we completed

---

[21] https://pmd.github.io/

[22] https://github.com/mjureczko/CKJM-extended

the Characteristics layer. When completing this layer, we assumed what we consider to be a ‘Standard Operations’ perspective, or one that models the average work week of a practitioner in an agile setting. In this perspective, a practitioner maintains a status quo, with perhaps a few pressing matters, yet not in an emergency mode. Of course, this is not true for all situations, yet the benefits of using QATCH allows for easy re-configuration of rankings to fit any need. Our rankings for the Characteristics layer is presented in appendix A, figure 6.17.

6.4.10.2 Model Calibration. QATCH requires model calibration before it can be appropriately applied to measure quality in a project. The calibration process involves calculating the metrics that make up the Properties layer across a large number of projects, and selecting the lowest, median, and highest value of each metric after removing outliers via inter-quartile range selection [70]. This process of ensures that the model contains maximum level of variability from the benchmark data. To calibrate our model, we used all of the projects and all their versions we present in this study treating each as if it were an independent project. We do this because we don’t concern ourselves with the scalar values of system quality, but rather we care about the relationships between quality and behavioral grime. Regardless, in [70], it is shown that little to no statistical difference occurs in model calibration when the benchmark repository features greater than 1.3 million lines of code, which we surpass with our projects. Once our model was calibrated, we evaluated every project and every version of this analysis (in table 6.3) using our implementation and calibration of QATCH.

After model calibration, we calculated the Total Quality Index (TQI) for each project across each version. The precise value of the TQI comes from the calibration process; if the model is calibrated on different benchmark repositories then each of these projects will yield different TQI values. Though, as [70] suggests, as the size of the benchmark repository increases, little statistically significant change is seen in TQI scores. The TQI value can be interpreted as follows: 'The quality score of a project when considering all of the benchmark repository as a reference point.' The TQI score must be in range [0,1], inclusive, and it is considered that scores between [.8 and 1.0] represent 5-star quality, [.6 to .8] represent 4 star quality, etc. The TQIs for each project are presented in figure 6.13. The Software Versions are presented on the x-axis, while TQI measurement is presented on the y-axis. Lines on the graph capture the TQI score of a software project over the versions under analysis. Recall that Software Version is a categorical variable, so while tempting, we cannot perform time-series analysis on these data. A visual inspection of figure 6.13 reveals that the quality of these projects does not majorly fluctuate over the versions we analyzed, beyond roughly a 5% change in TQI score. Though none of the projects feature monotonicity, which implies that either or both of: (1) when new features are added their quality is higher than existing code, or (2) developers are actively taking steps to refactor their code to improve quality.

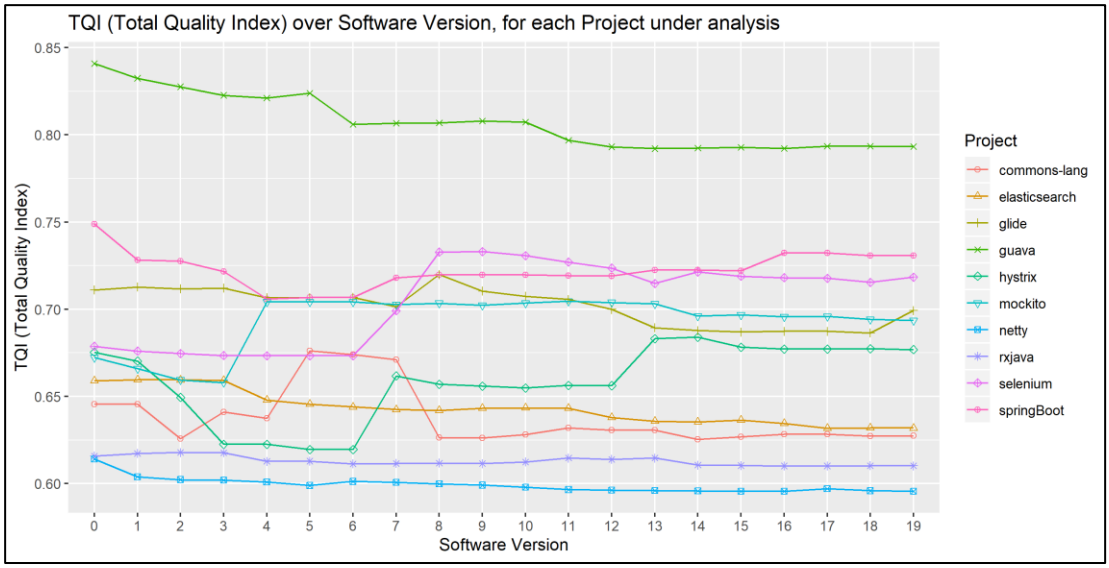


Figure 6.13 Quality scores (TQI) for each project under analysis, over each version under analysis. Software Version is shown on the x-axis, while TQI measurement is shown on the y-axis. Lines on the graph capture the TQI score of a software project over the versions under analysis

After quality calculations were complete, we generated scatter plots between pattern grime and quality to identify the precise nature of the relationship. Specifically, we use our metric Pattern Behavioral Aberrations, from table 6.19, as an aggregation of pattern grime per software project and across versions because of the holistic nature of our quality measurement. This scatter plot is presented in figure 6.14, with TQI scores shown on the y-axis, and scores of Pattern Behavioral Aberrations are shown on the x-axis. Pattern Behavioral Aberration scores capture how few instances of behavioral grime exist in a project, compared across projects in the benchmark repository. The score of Pattern Behavioral Aberrations is non-intuitive; high scores on the x-axis indicate **lower** individual measurements of grime, i.e. better pattern conformance, and lower scores indicate **higher** individual measurements of grime. Points in the graph represent an individual project's TQI score and Pattern Behavioral Aberration score. Generally, we

see a linear trend between Pattern Behavioral Aberrations and TQI, suggesting our relationship will be linear in nature. Because the nature of our data is a rank-based percentage, the rank being based on the benchmark repository we calibrated our QATCH model on, it falls under the ordinal numeric scale. This means we cannot use Pearson's method for calculating the correlation coefficients capturing the relationship between TQI and Pattern Behavioral Aberrations as we did for research question 5, from section 6.4.5. Instead, we use Spearman's rank-order correlation calculation. Spearman's provides a nonparametric alternative to Pearson's. The only assumption required for Spearman's is that the data has ordinal nature, which ours does. The estimate of Spearman's $\rho$ for these data is 0.594, suggesting a strong positive relationship between TQI and Pattern Behavioral Aberration score. When asserting the null hypothesis that the relationship is equal to zero, which implies no relationship, we received a $p$-value of $< 2.2^{-16}$, suggesting that the estimate of 0.594 is an accurate non-zero estimate of the true relationship between TQI and the Pattern Behavioral Aberration score.

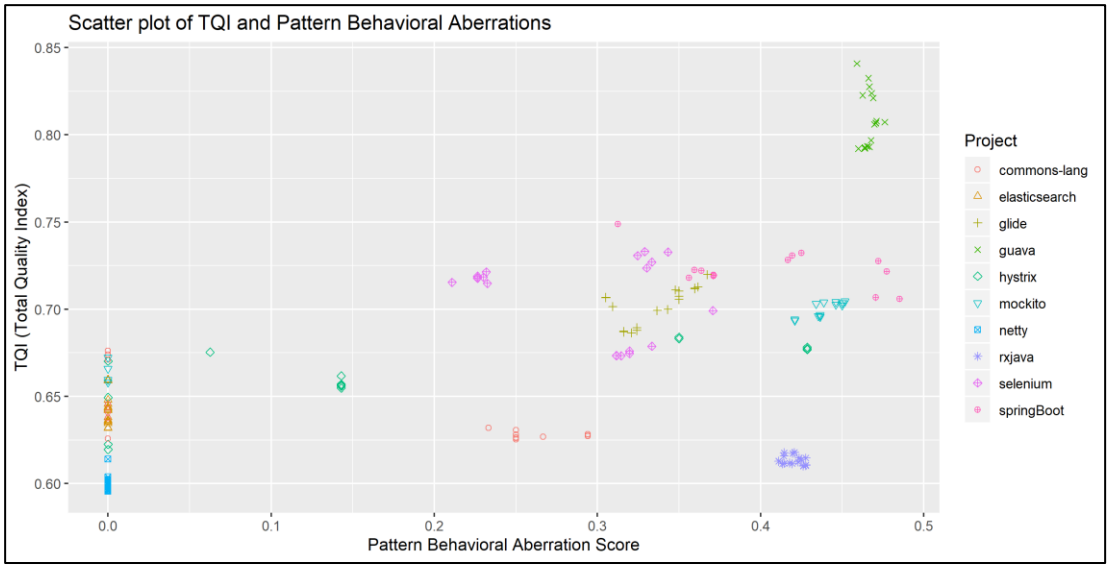


Figure 6.14 Scatter plot of TQI and Pattern Behavioral Aberrations. Scores of TQI are shown on the y-axis, and scores of Pattern Behavioral Aberrations are shown on the x-axis. Pattern Behavioral Aberration scores capture how few instances of behavioral grime exist in a project, compared across the other projects we used to calibrate the model. High scores indicate **lower** individual measurements of grime, i.e. better pattern conformance, and lower scores indicate higher individual measurements of grime. Points in the graph represent an individual project's TQI score and Pattern Behavioral Aberration score.

A strong relationship between TQI and Pattern Behavioral Aberration score indicates that pattern behavioral grime and system quality are strongly and inversely related. Recall that the Pattern Behavioral Aberration score captures how few instances of pattern behavioral grime occur in a software project when compared to all other projects in our benchmark repository. That is, a higher score indicates fewer instances of behavioral grime. Because of this, when we synthesize the results and translate Pattern Behavioral Aberration scores to design pattern behavioral grime measurements, we need to consider inverted relationships. Therefore, because we identified a strong relationship between TQI and Pattern Behavioral Aberration score, we can assert that a strong inverse relationship exists between quality and pattern behavioral grime.

This research question also aims to consider the relationship between pattern behavioral grime and Technical Debt. Recall that in research question 8, from section 6.4.8, we identified that no state-of-the art tools are capable of identifying or measuring behavioral grime, and that as a case study the state-of-the-art TD measurement tool SonarQube did not detect behavioral grime. Additionally, we know that results from a recent Dagstuhl seminar [4] suggest the definition of TD is narrowed to consider only internal code-quality issues, specifically pertaining to Maintainability. From these two points, we can conclude that no tools currently exist that calculate the TD score of behavioral grime, yet that Maintainability serves as the boundaries on a search space that contains all TD items. From this point, we choose to use Maintainability as a surrogate measurement for TD. While we do not have the means to calculate the actual effects of behavioral grime on TD, which requires domain-specific calibration techniques, we do know that any TD items must be contained within the Maintainability quality characteristic. Therefore, in order to assess the relationship between pattern behavioral grime and TD, we consider the relationship between pattern behavioral grime and Maintainability score.

To identify the relationship between pattern behavioral grime and Maintainability, we apply a similar analysis as we did between pattern behavioral grime and quality. This entails generating a scatter plot and calculating the Spearman correlation coefficient. The scatter plot between Pattern Behavioral Aberrations and Maintainability is shown in figure 6.15. Similar to the TQI scatter plot in figure 6.14, scores of Maintainability are shown on the y-axis, and scores of Pattern Behavioral Aberrations are shown on the x-

axis. Pattern Behavioral Aberration scores capture how few instances of behavioral grime exist in a project, compared across projects in the benchmark repository. High scores indicate **lower** individual measurements of grime, i.e. better pattern conformance, and lower scores indicate higher individual measurements of grime. Points in the graph represent an individual project's Maintainability score and Pattern Behavioral Aberration score. We see a similar linear trend in this scatter plot as we did for the TQI scatter plot, and therefore decided to calculate Spearman's $\rho$ for the same reasons. Our data yields a Spearman's $\rho$ value of 0.6652, implying a strong positive relationship between Pattern Behavioral Aberration scores and Maintainability. The $p$-value for this correlation coefficient is $< 2.2^{-16}$, implying the estimate of 0.6652 is an accurate non-zero estimate of the true relationship between Maintainability and the Pattern Behavioral Aberration score. For the same reasons as the TQI analysis, we can state that a strong negative relationship exists between pattern behavioral grime and Maintainability. Furthermore, we generalize this statement to state that a strong negative relationship exists between pattern behavioral grime and Technical Debt.

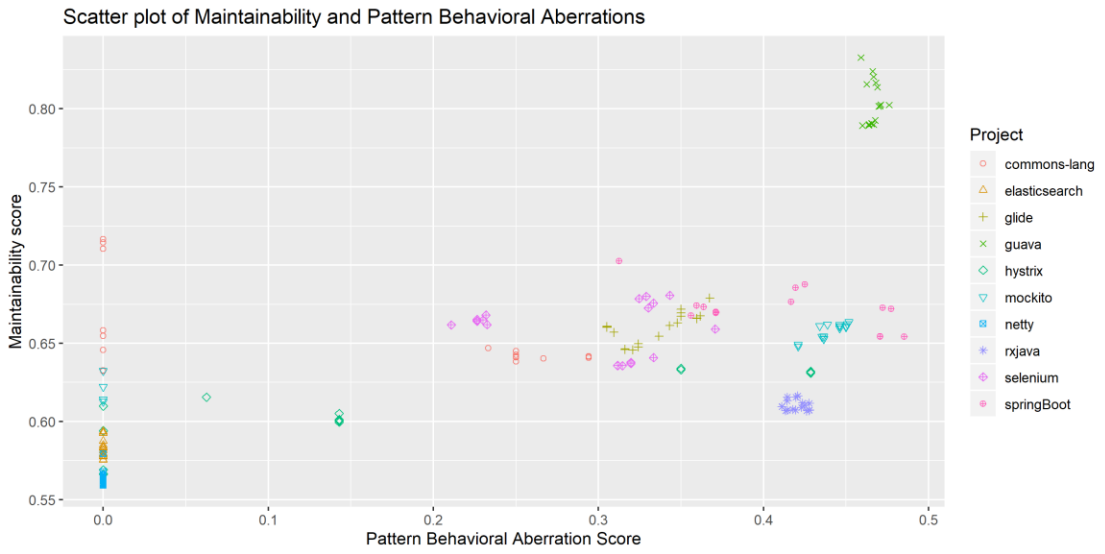


Figure 6.15 Scatter plot of Maintainability score and Pattern Behavioral Aberrations. Scores of Maintainability are shown on the y-axis, and scores of Pattern Behavioral Aberrations are shown on the x-axis. Pattern Behavioral Aberration scores capture how few instances of behavioral grime exist in a project, compared across the other projects we used to calibrate the model. High scores indicate **lower** individual measurements of grime, i.e. better pattern conformance, and lower scores indicate higher individual measurements of grime. Points in the graph represent an individual project's Maintainability score and Pattern Behavioral Aberration score.

## 6.5 Discussion

The high-level goals of our study were two-fold. First, we sought to investigate the usefulness of incorporating behavioral analysis in the context of design patterns, complementing existing structural models. Second, we sought to evaluate the relationship behavior and quality so that high- level goal of our study was to investigate the relationship between pattern behavioral grime and system quality and technical debt.

6.5.1 Structure vs Behavior

With respect to the relationship between structure and behavior, our results from research questions 1-7 indicate that studying behavior offers a new dimension of insight into design pattern evolution. Prior to this study, only structural analysis methods have been utilized to study design pattern evolution, which do provide better analysis than none at all, yet do not provide a complete view of the patterns or system. This statement is backed by our findings from table 6.5, which show that pattern instances are most likely to contain both structural and behavioral grime at some point in their lifetime. Interestingly, a large number of pattern instances contained structural grime but no behavioral grime. In the absence of behavioral grime analysis, this distinction would not be possible. The two categories of most prevalent grime, labeled **A** and **B** according to figure 6.8, would be combined as one designation if behavioral grime had not been explored. Label **C**, corresponding to pattern instances that contain behavioral grime but no structural grime, were considered impossible in the real world because of our initial understandings that structural deviations enforces behavioral deviations. In other words, a behavioral deviation would be impossible without a structural deviation in the first place. Our findings illustrate a select few pattern instances that violate this level of understanding, but it still represents a violation of our perceived understanding of design pattern evolution. Our initial statement that structural deviations enforce behavioral deviations is misguided; instead, a better statement is that structural deviations *guide* behavioral deviations. In other words, the structural aspects of a pattern, and more specifically the instances they are violated in, play a strong role on the presence of

behavioral deviations, but they do not restrict the presence of behavioral deviations. While the case where behavioral deviations appear in the absence of structural ones appears rare in software projects, it is still a possibility. This finding is significant, if not just for the context and research area of this study, but for (1) studies that seek to explore behavior in non-pattern settings, such as security concerns or computational performance, as well as (2) explorations of behavior in tools that perform quality and technical debt analysis. It is important to remember that just because an issue does not exist within a project's structure, issues might still exist with the project's behavior.

### 6.5.2 Behavior and Quality

Research questions 8-10 sought to answer the high-level goal concerned with the relationship between behavior and quality. We first identified that no state-of-the-art tools currently exist that identify behavioral grime. This finding, while not being a surprise, illustrates a gap in the field. And the identification of this gap, coupled with our results from our first high-level research goal which shows that behavior offers a new perspective into software quality assurance techniques, reveal that operational models of behavioral analysis offer a niche solution to a novel problem. However, it is the view of these authors that the implementation details of such operational models will always be under contention, simply because different stakeholders hold different views of what should be considered important in software quality. For example, a company following a rigorous lifecycle to release human-critical software will be concerned with different quality items than an indie video game development company; such differences will always exist to some degree. This problem is why we have elected to choose a

configurable quality model, specifically QATCH [70], to extend. The configuration in this model comes from the varying degrees of importance for each of the software quality Properties and Characteristics, and we show our rankings for the degrees of importance in appendix A, figures 6.16a, 6.16b, and 6.17. We expect that these rankings will change under different domains, because of the different quality concerns of stakeholders. Our extension of the QATCH model accommodates this, by providing design pattern evolution quality Properties that a user of QATCH can configure to their degree of importance. Specifically, we focus on the behavioral aspects of this extension because that is our primary concern in this research. Yet, this extension shows the process that any user of QATCH can follow to properly extend a model they are concerned with. This research and process proves the concept that was proposed in [70].

In terms of behavioral effects of system quality, we found strong inverse relationships between quality and behavioral grime, meaning that under our calibration of the quality model QATCH, the addition of behavioral grime elements in a project was strongly correlated with a decrease in system quality. We need to point out a few assumptions we made to reach this statement. First, we assumed a ‘strict’ implementation of design patterns. That is, we allowed each design pattern instance to have one incoming non-pattern member, and one outgoing non-pattern member. This decision was based on two premises: (1) patterns need some non-pattern relationships to actually be functional, and (2) preservation of maintainability in the design pattern instance is more important compared to system functionality. The second premise we make because we reason that non-design pattern based solutions can be used to solve as many problems as pattern-

based solutions, but the choice to use a design pattern implies one expends more effort designing and developing the pattern instance, but that that effort pays off in the long run with faster extensions of the code in the future. This ‘strictness’ expectation can be configured though; our tools and methods presented herein have the capability built-in, so that users that want a more relaxed implementation of design pattern instances can consider such.

A second assumption we made in this research is that the addition of pattern grime has a homogeneic and monotonically-negative impact on system quality. Meaning that each instance of grime that is added, per grime type, has the same negative effect on system quality. It could very well be that in certain applications, adding grime to a design pattern instance is simply the optimal solution to the problem, and our models and tools do not capture this possibility. However, capturing this phenomenon is incredibly difficult, and would likely require domain-specific implementations of models and tools. We aim to provide general models and tools with our research, with the benefit of configurability, yet we understand this large assumption in our study.

## 6.6 Threats to Validity

There are several design and implementation considerations in this study that threaten the validity of the results. External validity is concerned with the generalization of results. In this study, we limited ourselves to 20 minor-release versions of ten Java projects, chosen based on popularity from the online repository GitHub. While we attempted to systematically select projects so that our results would be generalizable, we

can only claim that our results hold true for the projects under analysis, and for the language (Java). More case studies are necessary before more general claims can be made concerning behavioral impacts, and specifically design pattern behavioral grime. However, our extension of the QATCH quality model is generalizable, and the manner in which it was developed makes it easy to do so [70].

Internal validity refers to the ability to reach causal conclusions based on the study design. Internal validity is minimal in this study because we make no causal claims, just correlations and linear model-fitting. In terms of the correlation results, specifically research question 5 from section 6.4.5, and research question 10 from section 6.4.10, we do not make claims beyond identifying the rate at which structural grime and behavioral grime, and behavioral grime and quality, increase together. In terms of research question 7, from section 6.4.7, which identifies the rate that patterns develop behavioral grime, we do provide estimates for a linear model equation. However, these estimates and their greater equation cannot be used to interpolate or extrapolate the rate at which behavioral grime appears in pattern instances. Generally, and though many times done incorrectly, linear models should not be used to extrapolate data, which we do not do here. In terms of interpolation, we cannot interpolate on these data either. This is because our choice of time-dependent variables are Software Version and Pattern Age, which are both categorical variables. We do not know how much physical time actually passed in between subsequent Software Versions, or resulting Pattern Ages. While Pattern Age is a derived metric based on Software Version, we are still under the same constraints; no data interpolation can be performed. However, our results showed that Project and

Pattern Type had the largest statistically significant impact on design pattern grime, suggesting their estimates supply better approximations for pattern behavioral grime.

Construct validity refers to the choice of independent and dependent variables, with respect to the conclusions of the study. Construct validity is threatened in our study from two major sources; pattern coupling and the measurements of software quality. In terms of the pattern coupling, it is important to note that we do not consider pattern coupling in this study. Pattern coupling entails that two or more separate pattern instances, likely from two or more separate pattern types, share one or more members. Our definitions of design pattern grime, and our conclusions, do not take into account the possibility that grime for one pattern instance might be a necessary member of a separate pattern instance. In terms of the measurements of software quality, construct validity is violated because of the selection of quality Properties used in the default QATCH model. The default QATCH model uses 11 Properties, originating from the PMD ruleset list[23] or the CKJM-extended metric package[24]. These Properties provide beneficial perspectives into quality, but they do not encompass all Properties that need to be considered when accounting for quality. We do extend the QATCH model to consider behavioral perspectives, but ultimately more Properties may need to be developed before a complete grasp of quality can be achieved.

[23] https://pmd.github.io/

[24] https://github.com/mjureczko/CKJM-extended

## 6.7 Conclusion

Our research goals focused on the exploration and initial understandings of behavioral deviations, as they pertain to design pattern evolution and software quality assurance. To this end, we have constructed a taxonomy that classifies behavioral grime types. Furthermore, we designed and implemented a case study wherein we measured counts of structural and behavioral grime, as well as software quality and TD, across pattern instance evolutions pertaining to seven design pattern types, originating from 20 versions of ten open source software projects. We evaluated the relationships between structural and behavioral grime and found statistically significant cases of strong correlations between specific types of structural and behavioral grime. We computed regressions that capture the rate at which design pattern behavioral grime appears in pattern instances, specifically finding that pattern type and project provided the most dominating terms in the models. We extended a state-of-the-art operationalized quality model, QATCH [70] to incorporate model terms that capture design pattern evolution properties, including behavioral grime, and we identified that a strong inverse relationship exists between design pattern behavioral grime and system quality. Furthermore, we identified a strong inverse relationship between design pattern behavioral grime and Maintainability, suggesting that behavioral grime is strongly related to TD.

# CHAPTER SEVEN

# CONCLUSIONS

## 7.0 Foreword

This chapter presents the conclusions of this doctoral dissertation. We begin in section 7.1 by re-iterating the problem statement, followed by a summarization of the work presented in the greater body of this document in section 7.2. Section 7.3 lists the contributions of this work, and section 7.4 considers the future of this work. Section 7.5 concludes.

## 7.1 Problem Statement

Software quality assurance techniques provide software developers and managers with the methods and tools necessary to monitor their software product to encourage fast, on-time, and bug-free releases for their clients. Ideal circumstances hold that the methods and tools of software quality assurance provide significant value and highly-specialized results to product stakeholders, while being fully incorporated into a firm's process and with actionable and easy-to-interpret outcomes. However, modern approaches fall short on these goals, and while many QA techniques exist that provide results to stakeholders, many times these results do not provide their stated value or are simply ignored. We claim this is due to two primary influences. First, current software QA approaches do not fully reveal all aspects of a software product because of their focus on static, or structural analysis. By itself, static analysis is not detrimental, yet it simply does not provide

sufficient insight into a product's inner-workings to allow for a thorough analysis. Second, many QA techniques provide general packaged solutions, which fail to capture domain-specific concerns. Different firms have different expectations of quality, both from an end-user perspective and from an internal software quality perspective. Packaged solutions do not provide maximum value because they either do not allow for the ability to configure the solution to cater to firm needs, or the customizations they provide are difficult to implement because of the arbitrary process in which such a solution is calibrated. Specifically, our formal problem statement is as follows:

> *Ideal circumstances hold that software quality assurance efforts provide significant, highly-specialized, and immediate value to software product stakeholders. However, many modern approaches fall short on these desires, due to lack of models that fully capture the entities of a system, as well as models that fail to capture domain specific concerns.*

To remedy these issues, we have committed to the exploration of behavioral analysis techniques, which consider the mechanisms that occur as a product is executing its code at runtime. Specifically, we focus on design pattern evolution because of the known quality properties of design patterns, yet our methods capture all instances where expected product behavior is known. The exploration of behavioral analysis techniques complements existing structural analysis techniques, expanding upon the capabilities of state-of-the-art QA techniques. Furthermore, the manner in which we developed and evaluated these newfound capabilities, via extending an existing quality model that is

highly-customizable yet easy-to-use and interpret, encourages a straightforward and non-arbitrary customization that fits all domains.

## 7.2 Summary of Work

Chapter 1 formulated the problem statement of this dissertation, and set the stage for the work performed in the greater body of this document. Chapters 2 and 3 served as empirical evidence that the problem statement is indeed an issue in the field. Specifically, Chapter 2 revealed that modern QA methods support developer intuition, and Chapter 3 revealed that out-of-the-box implementations of state-of-the-art tools provided differing results on what is considered good software quality. These two results laid the groundwork for the larger body of work presented in this dissertation.

Chapter 5 illustrates the results from a presentation [63] to the greater empirical software engineering community at the International Doctoral Symposium on Empirical Software Engineering (IDoESE'15), of a proposed plan of action to address a clear gap in the research. This plan entailed exploring behavioral deviations in the context of design pattern evolution, so that QA techniques can be advanced further, to ultimately supply practitioners and managers with more advanced and useful techniques to monitor and act on software QA. The feedback we received was that our four goals were very ambitious, and it was suggested we remove the fourth goal pertaining to prediction of behavioral deviations, which we elected to do. Yet it was agreed upon that such a plan would provide significant value to the field.

Chapter 6, which is based on a publication [83] at the International Conference on Software Reuse (ICSR'19) conference, and a work-in-progress submission to the IEEE Transactions on Software Engineering, presented the results from the remaining two research goals. These goals are paraphrased as: (1) "*investigation of design pattern instances for the purpose of identifying and characterizing behavioral grime*", and (2) "*quantify the impact of behavioral grime on quality and TD*". To address the first goal, we constructed a taxonomy of design pattern behavioral grime that considers all known forms of behavioral grime, and is used as a complement to existing structural taxonomies. We then evaluated the relationship between behavioral grime and structural grime, to illustrate how the two forms of analysis can complement one another. We found that strong relationships exists between five pairs of structural and behavioral grime, specifically TEER/PEE, PEER/TEE, PEAO/PI, PEAO/TEA, and PEAO/TI. To address the second goal, we extended an existing state-of-the-art operational quality model to incorporate model-based behavioral issues, and we used the extended model to evaluate the relationship between behavioral grime and quality and TD. We found that the presence of behavioral grime has a strong negative correlation with system quality, and a strong negative correlation with Maintainability, which serves as a surrogate measurement to TD.

## 7.3 Contributions

The contributions of this body of work are the following:

1. Identification of model-based behavioral deviations in code.

2. Classification of model-based behavioral deviations in code into a taxonomy.
3. Comparison to existing structural models to reveal how behavioral analysis can complement structural analysis.
4. Extension of an existing state-of-the-art quality measurement model to incorporate model-based behavioral deviations.
5. Evaluation of the relationship between model-based behavioral deviations and system quality and TD.

## 7.4 Future Work

Because this work marks the beginning of the exploration of a new phenomenon, there are many routes for future work to extend this work. First, expanding upon the behavioral grime taxonomy would be a valuable prospect. This would require a combined effort of *in-vitro* and *in-vivo* work, where the *in-vitro* work represents the possible or theoretical forms of behavioral grime, and the *in-vivo* work validates that such a form of behavioral grime exists in the real-world. We were not able to conceive of additional forms of behavioral grime when completing this work, but that does not mean additional forms of behavioral grime do not exist in software systems. If additional forms of behavioral grime are discovered, subjecting them to the same process as presented herein would be helpful for generating a deeper understanding of them. Specifically, this would entail characterizing their forms in an extended taxonomy, and evaluating them with respect to quality and TD.

A second form of future work entails expanding on the projects under analysis. The behavioral grime work presented in this dissertation features ten Java project, and 20 versions of each project. However, this sample is not representative of the entire population of software project, so our ability to generalize is limited. Performing replication studies on more projects would build on the results from this study, which would increase understanding of this field.

A third outlet of future work involves exploring more design pattern types for behavioral grime. The work in Chapter 6 considers only seven pattern types, which were selected because they were the most populous pattern types reported from the design pattern detection tool we used [75]. We found numerous forms of behavioral grime across six of the seven pattern types, with the exception of the Observer pattern, though we only identified one pattern instance evolution of the Observer pattern. Expanding on the pattern types will help identify the extent of behavioral grime.

## 7.5 Conclusion

This body of work encompasses the work performed to fulfill the requirements of the Doctor of Philosophy degree. Herein, we identified a gap in the research field pertaining to software quality assurance. Chapters 2 and 3 served as empirical evidence such a gap exists. Chapter 5 proposed a plan to explore the gap, which was vetted by the empirical software engineering research community as being a valuable contribution to the field. Chapter 6 completed the proposed contribution, resulting in the identification, classification, and evaluation of model-based behavioral deviations in software.

# REFERENCES CITED

1. Ammann, P., & Offutt, J. (2008). Introduction to software testing. Cambridge University Press.

2. Ampatzoglou, A., Michou, O., and Stamelos, I. Building and mining a repository of design pattern instances: Practical and research benefits, Entertainment Computing, Volume 4, Issue 2, April 2013, Pages 131-142, ISSN 1875-9521, DOI= http://dx.doi.org/10.1016/j.entcom.2012.10.002.

3. Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J.D. and Penix, J., 2008. Using static analysis to find bugs. *IEEE software*, *25*(5), pp.22-29.

4. Avgeriou, P., Kruchten, P., Ozkaya, I. and Seaman, C., 2016. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports* (Vol. 6, No. 4). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

5. Bakota, T., Hegedűs, P., Körtvélyesi, P., Ferenc, R. and Gyimóthy, T., 2011, September. A probabilistic software quality model. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)* (pp. 243-252). IEEE.

6. Baldwin, C. and Clark, K. 2000. *Design Rules: The power of Modularity*. Vol. 1. MIT Press., Cambridge, MA.

7. Bansiya, J. and Davis, C. G. 2002. A hierarchical model for object-oriented design quality assessment. In *IEEE Transactions on Software Engineering* 28, 1 (Aug. 2002), 4- 17. DOI=http://dx.doi.org/10.1109/32.979986.

8. Basili, V. R., Selby, R. W., and Hutchens, D. H. 1986. Experimentation in Software Engineering. In *IEEE Transactions on Software Engineering* 12,7 (July 1986), 733-743. DOI=http://dx.doi.org/10.1109/TSE.1986.6312975.

9. Basili, V.R., 1992. *Software modeling and measurement: the Goal/Question/Metric paradigm*.

10. Basili, V., Caldiera, G., and Rombach, H. D. 1994. The goal question metric approach. Encyclopedia of Software Engineering. 2, 528-532. DOI=http://dx.doi.org/ 10.1002/0471028959.sof142.

11. Bieman, J.M., and Wang, H. 2006. Design pattern coupling, change proneness, and change coupling: A pilot study. Technical Report. Colorado State University.

12. Brooks, A., Roper, M., Wood, M., Daly, J., and Miller, J. 2008. Replication's Role in Software Engineering. In *Guide to Advanced Emprirical Software Engineering*, Shull, F., Singer, J., and Sjøberg, D. I. K. Springer London, Springer, 365-379.

DOI=http://dx.doi.org/10.1007/978-1-84800-044-5_14.

13. Brown, W. H., Malveau, R. C., McCornnick III, H. W., and Mowbray, T. J. 1998. Antipatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley & Sons, NY.

14. Budgen, D., Turner, M., Brereton, P. and Kitchenham, B.A., 2008, September. Using Mapping Studies in Software Engineering. In *PPIG* (Vol. 8, pp. 195-204).

15. Campbell, D.T. and Cook, T.D., 1979. *Quasi-experimentation: Design & analysis issues for field settings*. Chicago: Rand McNally College Publishing Company.

16. Chidamber, S.R. and Kemerer, C.F., 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, *20*(6), pp.476-493.

17. Chin, S., Huddleston, E., Bodwell, W. and Gat, I., 2010. The economics of technical debt. *Cutter IT Journal*, *23*(10), p.11.

18. Collard, M.L., 2005, May. Addressing source code using srcml. In *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC'05)*.

19. Cunningham, W. 1992. The Wycash portfolio management system. In *OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications* (Dec. 1992). OOPSLA '92. SIGPLAN ACM, New York, NY 29-30. DOI= http://dx.doi.org/10.1145/157709.157715.

20. Curtis, B., Sappidi, J. and Szynkarski, A., 2012. Estimating the principal of an application's technical debt. *IEEE software*, *29*(6), pp.34-42.

21. Curtis, B., Sappidi, J. and Szynkarski, A., 2012, June. Estimating the size, cost, and types of technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt* (pp. 49-53). IEEE Press.

22. Dale, M.R., and Izurieta, C. 2014. 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). ACM, New York, NY, USA, Article 37, 4 pages. DOI=http://doi.acm.org/10.1145/2652524.2652560.

23. De Veaux, R. D., Stats: data and models, 3rd ed ed., Boston: Pearson Education,

2012.

24. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., and Mockus, A. Does code decay? Assessing the evidence from change management data, Software Engineering, IEEE Transactions on, vol.27, no.1, pp.1-12, Jan 2001.

25. Feitosa, D., Avgeriou, P., Ampatzoglou, A. and Nakagawa, E.Y., 2017, November. The evolution of design pattern grime: An industrial case study. In *International Conference on Product-Focused Software Process Improvement* (pp. 165-181). Springer, Cham.

26. Feitosa, D., Ampatzoglou, A., Avgeriou, P. and Nakagawa, E.Y., 2018. Correlating pattern grime and quality attributes. *IEEE Access*, *6*, pp.23065-23078.

27. Ferenc, R., Hegedűs, P., and Gyimóthy, T.,, "Software Product Quality Models," in Evolving Software Systems, T. Mens, A. Serebrenik and A. Cleve, Eds., Berlin, Heidelberg, Springer Berlin Heidelberg, 2014.

28. Fowler, M., Beck, K., Brant, J., and Opdyke, W. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman, Inc., Reading, MA.

29. France R. E. S., Kim, D., and Ghosh, S., Metarole-Based Modeling Lanugage (RBML) Specification V1.0, 2002.

30. France, R.B., Kim, Dae-Kyoo, Ghosh, S., and Song, E. 2004. A UML-based pattern specification technique, Software Engineering, IEEE Transactions on, vol.30, no.3, pp.193, 206. DOI=http://dx.doi.org/10.1109/TSE.2004.1271174

31. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

32. Gaudin, O., 2009. Evaluate your technical debt with Sonar. *Sonar, Jun*.

33. Griffith, I., and Izurieta, C. 2013. Design Pattern Decay: An Extended Taxonomy and Empirical Study of Grime and its Impact on Design Pattern Evolution. In Proceedings of the 11th ACM/IEEE International Doctoral Symposium on Empirical Software Engineering and Measurements, USA.

34. Griffith, I., and Izurieta, C. 2014. Design pattern decay: the case for class grime. In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14). ACM, New York, NY, USA,

Article 39, 4 pages. DOI=http://doi.acm.org/10.1145/2652524.2652570

35. ISO, I., 1991. Information technology-software product evaluation-quality characteristics and guide lines for their use. *Iso/iec is*, *9126*.

36. ISO/IEC 25010: Systems and software engineering. Systems and Software Quality Requirements and Evaluation (SQuaRE). System and software quality models, 2011.

37. Izurieta, C., and Bieman, J. 2007. How software designs decay: A pilot study of pattern evolution. In Proceedings of the First Symposium on Empirical Software Engineering and Measurement (Madrid, Spain, 2007). ESEM 2007. 449-451. DOI= http://dx.doi.org/10.1109/ESEM.2007.55.

38. Izurieta, C. 2009. Decay and Grime Buildup in Evolving Object Oriented Design Patterns. Ph.D. Dissertation. Colorado State University, Fort Collins, CO, USA. Advisor(s) James Bieman. AAI3385139.

39. Izurieta, C. and Bieman, J. 2013. A multiple case study of design pattern decay, grime, and rot in evolving software systems. In *Software Quality Journal*, 21, 2 (June 2013),
289-323, DOI=http://dx.doi.org/10.1007/s11219-012-9175-x.

40. Juristo, N. and Moreno, A. M. 2010. *Basics of Software Engineering Experimentation* (1st ed.). Springer Publishing Company, Incorporated.

41. Kendall, M. G. 1938. A new measure of rank correlation. In *Biometrika*, 30 (1938), 81-93.

42. Kim, D. 2004. A Meta-Modeling Approach to Specifying Patterns, Ph.D. Dissertation. Colorado State University, Fort Collins, CO, USA. Advisor(s) Robert France.

43. Kim, D. The Role-Based Metamodeling Language for Specifying Design Patterns. In Toufik Taibi, editor, Design Pattern Formalization Techniques. Idea Group Inc., 2006.

44. Knoop, J., Rüthing, O. and Steffen, B., 1994. *Partial dead code elimination* (Vol. 29, No. 6, pp. 147-158). ACM.

45. Lajoie, R. and Keller, R.K., 1995. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert. In *Object-Oriented Technology for Database and Software Systems* (pp. 295-312).

46. Letouzey, J.L. and Ilkiewicz, M., 2012. Managing technical debt with the SQALE method. *IEEE software*, *29*(6), pp.44-51.

47. Letouzey, J.L., 2012, June. The SQALE method for evaluating technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)* (pp. 31-36). IEEE.

48. Li, W. and Henry, S., 1993. Object-oriented metrics that predict maintainability. *Journal of systems and software*, *23*(2), pp.111-122.

49. Marinescu, R., 2012. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, *56*(5), pp.9-1.

50. Martin, R.C., 2002. *Agile software development: principles, patterns, and practices*. Prentice Hall.

51. McCabe, T.J., 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4), pp.308-320.

52. McConnell, S., 2008. Managing technical debt. *Construx Software Builders, Inc*, pp.1-14.

53. Mordal-Manet, K., Balmas, F., Denier, S., Ducasse, S., Wertz, H., Laval, J., Bellingard, F. and Vaillergues, P., 2009, September. The squale model—A practice-based industrial quality model. In *2009 IEEE International Conference on Software Maintenance* (pp. 531-534). IEEE.

54. Nugroho, A., Visser, J. and Kuipers, T., 2011, May. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt* (pp. 1-8). ACM.

55. O'Keeffe, M. and Cinnéide, M.Ó., 2006, March. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)* (pp. 10-pp). IEEE.

56. Ohlsson, N. and Alberg, H. 1996. Predicting fault-prone software modules in telephone switches. In *IEEE Transactions on Software Engineering*, 22, 12 (Dec. 1996), 886-894, DOI= http://dx.doi.org/10.1109/32.553637.

57. Olague, H.M., Etzkorn, L.H., Gholston, S. and Quattlebaum, S., 2007. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on software Engineering*, *33*(6), pp.402-419.

58. Ostrand, T. J. and Weyuker, E. J. 2007. How to measure success of fault prediction models. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting* (SOQUA '07). ACM, New York, NY, USA, 25-30. DOI= http://doi.acm.org/10.1145/1295074.1295080.

59. Ott, R. and Longnecker, M.1993. *An introduction to statistical methods and data analysis*. Vol. 4. Duxbury Press, Belmont, CA.

60. Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (December 1972), 1053-1058.

61. Petersen, K., Feldt, R., Mujtaba, S. and Mattsson, M., 2008, June. Systematic mapping studies in software engineering. In *Ease* (Vol. 8, pp. 68-77).

62. Plasil, F., & Visnovsky, S. (2002). Behavior protocols for software components. IEEE transactions on Software Engineering, 28(11), 1056- 1076.

63. Reimanis D., Izurieta C., "A Research Plan to Characterize, Evaluate, and Predict the Impacts of Behavioral Decay in Design Patterns," IEEE ACM IDoESE, 13th International Doctoral Symposium on Empirical Software Engineering, Beijing, China, October 19 2015.

64. Reimanis, D. and Izurieta, C., 2016, October. Towards assessing the technical debt of undesired software behaviors in design patterns. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)* (pp. 24-27). IEEE.

65. Rumbaugh, J., Jacobson, I. and Booch, G., 2004. *Unified modeling language reference manual, the*. Pearson Higher Education.

66. Sangal, N., Jordan, E., Sinha, V., and Jackson, D. 2005. Using dependency models to manage complex software architecture. In Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05). ACM, New York, NY, USA, 167-176. DOI=http://doi.acm.org/10.1145/1094811.1094824.

67. Schanz, T., and Izurieta, C. 2010. Object oriented design pattern decay: a taxonomy. In Proceedings of the 2010 ACMIEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA, Article 7, 8 pages. DOI=http://doi.acm.org/10.1145/1852786.1852796.

68. Schwanke, R., Xiao, L., and Cai, Y. 2013. Measuring architecture quality by structure plus history analysis. In *2013 35th International Conference on Software*

*Engineering (ICSE)* (San Francisco, CA, May18 - 26 2013). ICSE '13. IEEE, San Francisco, CA, 891-900. DOI= http://dx.doi.org/10.1109/ICSE.2013.6606638.

69. Shull, F. J., Carver, J. C., Vegas, S., and Juristo, N. 2008. The role of replications in Empirical Software Engineering. In *Empirical Software Engineering* 13, 2 (April 2008), 211- 218. DOI= http://dx.doi.org/ 10.1007/s10664-008-9060-1.

70. Siavvas, M.G., Chatzidimitriou, K.C. and Symeonidis, A.L., 2017. QATCH-An adaptive framework for software product quality assessment. *Expert Systems with Applications*, *86*, pp.350-366.

71. Strasser, S., Frederickson, C., Fenger, K., and Izurieta, C. 2011. An automated software tool for validating design patterns. In Proceedings of the of ISCA 24th International Conference on Computer Applications in Industry and Engineering (HI, USA, November 16-18). CAINE'11.

72. Sunyé, G., Pollet, D., Le Traon, Y., & Jézéquel, J.M. "Refactoring UML models," In: Proc. Int. Conference Unified Modeling Language (pp. 134-138), Lecture Notes in Computer Science 2185, Springer, Heidelberg. 2001.

73. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. and Noble, J., 2010, December. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference* (pp. 336-345). IEEE.

74. Tom, E., Aurum, A., and Vidgen, R. 2013. An exploration of technical debt. J. Syst. and Softw. 86, 6 (Jun. 2013), 1498- 1516. DOI=http://dx.doi.org/10.1016/j.jss.2012.12.052.

75. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis, S.T., 2006. Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, *32*(11), pp.896-909.

76. Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., Seidl, A., Goeb, A. and Streit, J., 2012, June. The quamoco product quality modelling and assessment approach. In *Proceedings of the 34th international conference on software engineering* (pp. 1133-1142). IEEE Press.

77. Warmer, J. B., & Kleppe, A. G. (1998). The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series).

78. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A.

2012 *Experimentation in software Engineering*. Springer Berlin Heidelberg. DOI= http://dx.doi.org/10.1007/978-3-642-29044-2.

79. Wong, S., Cai, Y., Kim, M., and Dalton, M., 2011. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, New York, NY, USA, 411-420. DOI= http://doi.acm.org/10.1145/1985793.1985850.

80. Zazworka, N., Shaw, M.A., Shull, F. and Seaman, C., 2011, May. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt* (pp. 17-23). ACM.

81. Zazworka, N., Seaman, C. and Shull, F., 2011, May. Prioritizing design debt investment opportunities. In *Proceedings of the 2nd Workshop on Managing Technical Debt* (pp. 39-42). ACM.

82. Zazworka, N., Vetro, A., Izurieta, C., Wong, S., Cai, Y., Seamon, C., and Shull, F. 2013. Comparing four approaches for technical debt identification. In *Software Quality Journal* (April 2013), 1-24, Springer US.

83. Reimanis D., Izurieta,C, "Behavioral Evolution of Design Patterns: Understanding Software Reuse through the Evolution of Pattern Behavior," 18th International Conference on Software Systems and Reuse, ICSR 2019. In: Peng X., Ampatzoglou A., Bhowmik T. (eds) Reuse in the Big Data Era. Vol 11602, Springer Cham. https://doi.org/10.1007/978-3-030-22888-0_6 Cincinnati, OH, June 26-28 2019.

84. Wieringa, R.J., 2014. *Design science methodology for information systems and software engineering*. Springer.

85. Feitosa, D., 2019. *Applying Patterns in Embedded Systems Design for Managing Quality Attributes and Their Trade-offs*. PhD dissertation. University of Groningen, The Netherlands.

86. Izurieta C., Reimanis D., Griffith I., Schanz T. "Structural and Behavioral Taxonomies of Design Pattern Grime," 12th Seminar on Advanced Techniques & Tools for Software Evolution. SATToSE 2019, Bolzano, Italy, July 8-10, 2019.

87. Pearson, E.S., 1929. Some notes on sampling tests with two variables. Biometrika, pp.337-360.

# APPENDIX A

## RANKINGS OF QUALITY ENTITIES FOR QATCH MODEL CALIBRATION

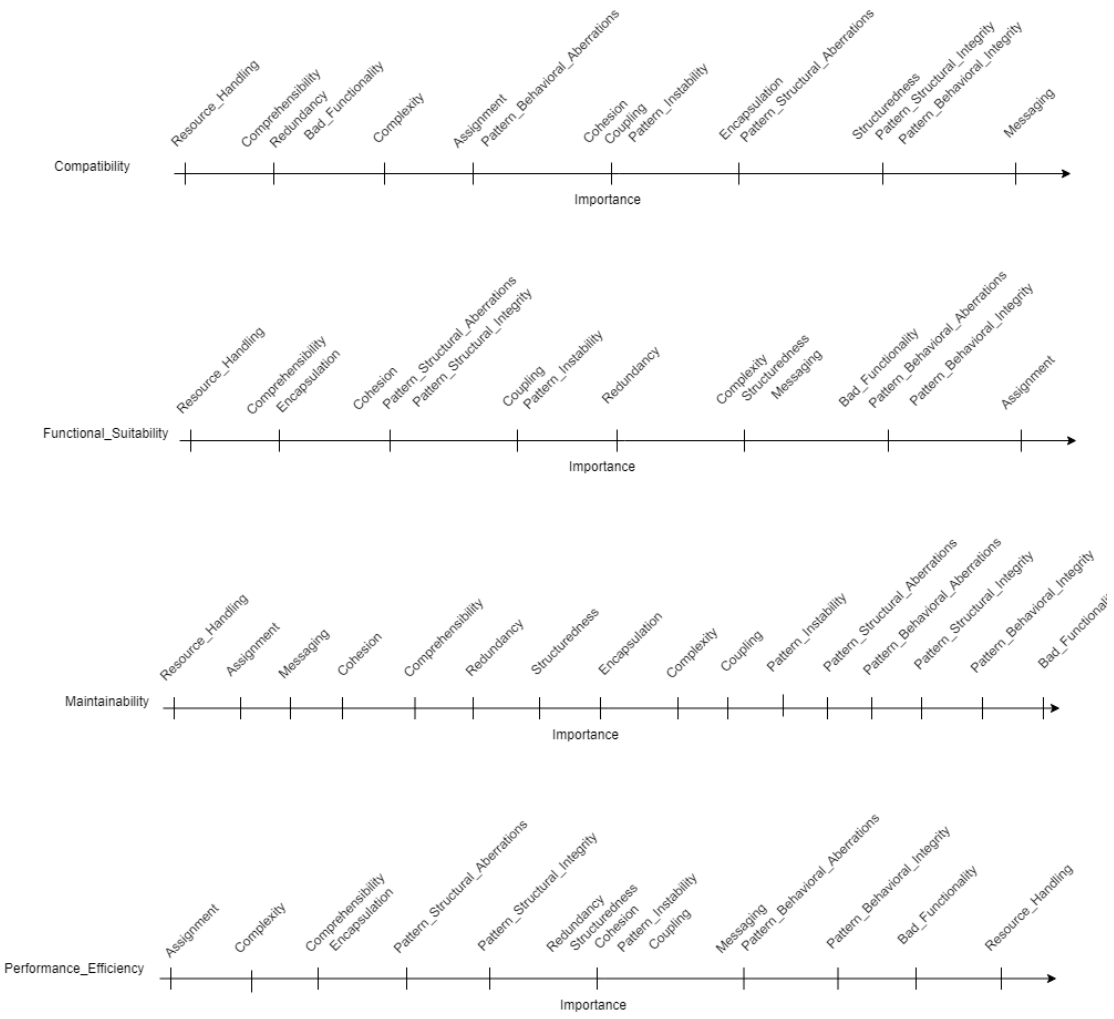


Figure 6.16a Our rankings for the Properties for our implementation of the QATCH [6] model. Quality Characteristics are shown on the left side of each line, and the Property's ranked importance is shown on top of each line. Tick lines indicate when multiple Measurements share the same importance.

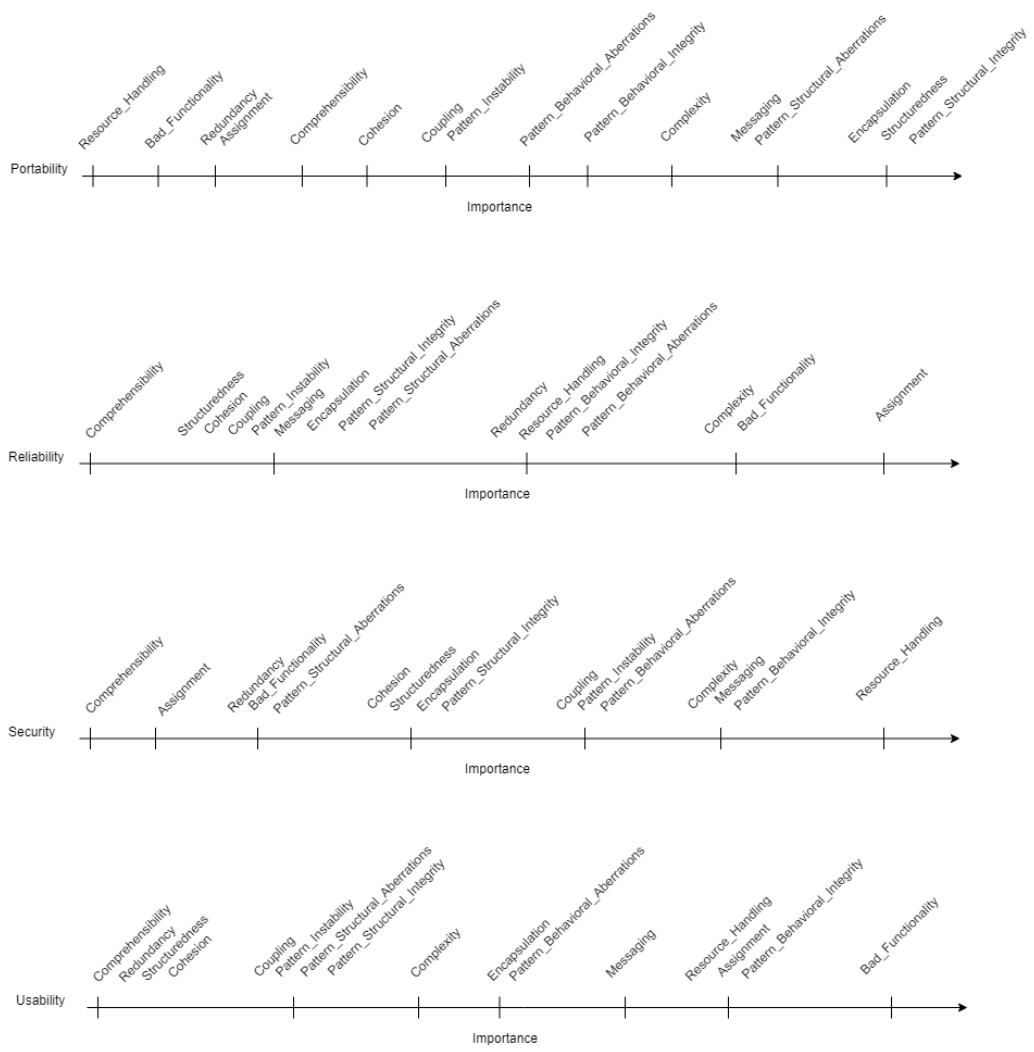


Figure 6.16b Our rankings for the Properties for our implementation of the QATCH [6] model. Quality Characteristics are shown on the left side of each line, and the Property's ranked importance is shown on top of each line. Tick lines indicate when multiple Measurements share the same importance.

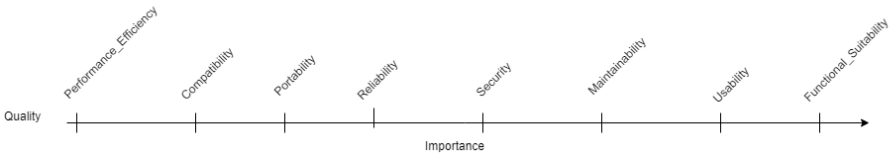


Figure 6.17 Our rankings for the Characteristics for our implementation of the QATCH [6] model. Quality is shown on the left side of each line, and the Characteristic's ranked importance is shown on top of each line.