DESIGN PATTERN DECAY – A STUDY OF DESIGN PATTERN GRIME AND ITS IMPACT ON QUALITY AND TECHNICAL DEBT

by

Isaac Daniel Griffith

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

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 2021

DEDICATION

I dedicate this dissertation to the memory of my uncle Lynn Griffith, a man whose outlook and dedication towards bettering himself consistently reminds me that we should all strive towards this goal.

ACKNOWLEDGEMENTS

I want to acknowledge my dissertation chair Dr. Izurieta who stood by me and this work for the very long time it took to complete. Similarly, I would like to acknowledge my dissertation committee members: Dr. Poole, Dr. Wittie, and Dr. Borkowski, who have also continued serving and providing guidance over this long journey. I would also like to acknowledge the support of my family, especially my wife Lora, who has never doubted that I would complete this work. Finally, I would like to thank my students at Idaho State University. Without their support, I would not have completed this work.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF TABLES – CONTINUED

Table Page

LIST OF FIGURES

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

LIST OF ALGORITHMS

ABSTRACT

Technical debt is a financial metaphor describing the trade-off between the short-term benefits gained and long-term consequences of design and implementation shortcuts taken over the evolution of a software product. These shortcuts typically manifest as design disharmonies such as code smells, anti-patterns, or design pattern grime.

Design pattern grime, which manifests as the accumulation of unnecessary or unrelated software artifacts within design pattern instance classes is of serious concern. Design patterns represent agreed upon methods to solve common problems and are based upon sound principles of good design; thus, these pattern instances' decay implies an evolution away from good design.

Currently, little is known about the causal nature of design pattern grime on technical debt and quality or how these three issues interrelate. What is the nature of the relationships between structural design pattern grime, software maintainability, and technical debt measurement?

To better understand design pattern grime, we have extended the structural grime taxonomy. We developed an approach to generate design pattern grime instances and inject them with design pattern grime. Using this approach, we conducted 7 experiments evaluating the effects of 26 forms of grime, at 6 severity levels within 16 design pattern types, on software maintainability and technical debt. The results showed that depending upon grime type, grime severity, and pattern type, grime does significantly affect both maintainability and technical debt.

We also conducted a verification study on pairs of pattern instances from open-source software systems to evaluate how well the injection process represents the real effects of grime and to verify the results of the experiments. The results of this study showed that there is a disconnect between the injection process and reality, indicating that refinements are still needed. However, the verification study worked as expected in indicating where issues may exist in the process.

CHAPTER ONE

INTRODUCTION

Quality is free, but only to those who are willing to pay heavily for it.

–T. DeMarco and T. Lister

In 1992, Ward Cunningham developed a financial metaphor to explain the need for continuous refactoring to stakeholders, coining the term Technical Debt [58]. Technical Debt has since gained traction as a major concern for software engineers, as detailed in a 2012 CAST Research report [233]. This report detailed the analysis of 745 applications comprising 365 million lines of code and found that, on average, there is $3.61 of Technical Debt per line of code. This result implies that Technical Debt is a significant factor in the long-term cost and sustainability of a software product. A fact later enshrined in the following definition of Technical Debt:

> "In software intensive systems, technical debt–is a design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability." [21]

The research community has explored many different aspects of Technical Debt (TD). However, the primary focus of this dissertation is Technical Debt Management (TDM). Technical Debt Management has the primary goals of preventing TD from being incurred or keeping TD at reasonable levels [164]. These goals are achieved through a set of nine activities, two of which are the focus of this dissertation: TD measurement and TD identification.

TD measurement concerns the development and evaluation of techniques to estimate the accumulated TD within a software system [164]. Li et al. [164] identified six categories for TD measurement, of which the most prominent approach is through the use of mathematical formulas or models [50,59,60,63,99,160,161,202,203]. Although, several studies have shown a relationship between the incurrence of TD and a compromise of one or more quality characteristics (or sub-characteristics), most notably maintainability [164], the relationships between TD, TD Items, and Quality remains undefined [103].

Thus, we consider the second aspect of TDM we are concerned with, TD identification. **TD identification** focuses on the detection of issues resulting from intentional and unintentional technical decisions [164]. Li et al. [164] identified four categories of TD identification, with the two most pertinent being:

1. Code Analysis: Source code analysis using various techniques including static analysis and metrics to identify coding rule violations and architectural or design issues.

2. Dependency Analysis: Analyze the dependencies between software artifacts.

TD Identification is of utmost importance, as the gap between TD, TD Items, and Quality appears to stem from existing quality models lacking representation of under-studied forms of TD. One such form is Design Pattern Grime, first identified by Izurieta [132] and later identified as a form of grime by Izurieta et al. [131] as a part of the TD landscape.

Although the forms of TD have expanded well beyond the TD landscape; Grime was selected as it affects a software artifact that has a known specification, *design patterns* [96]. Patterns are a widely known approach for describing generalized solutions to well-known problems that occur during software development, and they also provide a means to encode software design principles and practices, i.e., design knowledge. Furthermore, this knowledge formalized through pattern specification languages, such as the Role-Based Meta-modeling Language (RBML) [90], provides the basis for the definition and evaluation of pattern instance decay.

This decay occurs when a pattern instance deviates from its specification [128] by introducing artifacts or relationships which are not functionally or structurally necessary, we call this *design pattern grime* [132].

1.1 Research Design

In this section, we describe the research design comprising this dissertation. In Section 1.1.1 we introduce the problem statement central to this research. In Section 1.1.2 we describe the design science framework employed in this endeavor. In section 1.1.3 we identify the research questions considered in this dissertation. Finally, Section 1.1.4 describes the empirical methods employed in this research.

1.1.1 Problem Statement

As we are considering it, Grime is effectively artifacts negative evolution of a pattern instance's architecture and an evolution away from good design. Furthermore, Grime embodies both Design and Architectural TD (TD introduced in the design phase or affects the system design or architecture and all of its dependencies). These forms of TD have far-reaching consequences, as both direct the implementation of the software and can be very costly to reengineer [186, 187, 202, 278]. Additionally, studying a phenomenon in both forms of TD can provide insight into techniques to address these more significant problems.

In summary, the overarching goal of this research is to evaluate Grime and its effects on quality and Technical Debt indices by characterizing its nature, taxonomy, and effects using a measurement-driven approach. Though specific to design pattern grime, this approach serves as a general approach helpful in developing a deeper understanding of software engineering phenomena. Studies into the effects and properties of technical debt related phenomena such as design pattern grime have relied primarily on correlation analysis and observational study. Thus, little is known about the causal nature of these phenomena when considering Technical Debt and Software Maintainability. Furthermore, very few studies have explored the relationship to existing Technical Debt estimation approaches or Maintainability as defined in operationalized quality models. Towards this, and focusing on design pattern grime, we ask the following overarching research question:

What is the nature of the relationships between structural design pattern grime, software maintainability, and technical debt measurement?

1.1.2 Design Science Framework

In this research, we have adopted a design science framework [275]. Design Science, as introduced by March and Smith [178], describes an approach whereby stakeholders, serving human purposes, create or improve "things" within technological solutions.

Wieringa [275] identified two critical components for addressing the problems of interest using Design Science:

- **Design Problems:** "call for a change in the real world and require an analysis of actual or hypothetical stakeholder goals" [275]
- **Knowldege Questions:** "do not call for a change but ask for knowledge about the world as it is" [275]

These two concepts are related. In that, each of these question types is answered in separate but interacting ways. First, design problems are answered via the design cycle in which a proposed solution is designed and evaluated against the stakeholder goals, which generates new knowledge questions. Conversely, we apply empirical methods within an empirical cycle to investigate and answer the knowledge questions while generating new design questions.

This Design Science framework, when considering the interplay between the two cycles, is well suited for Software Engineering research. The problems we encounter often require the evaluation, modification, or creation of software artifacts while also pushing the boundaries of current theory and knowledge.

1.1.3 Problem Decomposition

This section presents the design problems and knowledge questions to be addressed in this dissertation and how each is related. To evaluate address the stated problem, we have used the Goal Question Metric (GQM) [28] approach to define the following research goals and their related research questions. For the purposes of this hierarchy, we consider both Knowledge and Design Questions to be Research Questions (RQ).

In order to address our stated problem of understanding the relationship between design pattern grime, quality, and technical debt, we first need to have a deeper understanding of Design Pattern Grime. Specifically, a deeper understanding of Class and Organizational Grime was needed. This need led to the Research Goal RG1. From this goal, we developed the two research questions:

RG1: Analyze design patterns to elaborate on the complete taxonomy of Class and Organizational Grime.

- **RQ1.1:** What are the types of Class Grime?

  Rationale: This is a fundamental question of this research, inquiring as to the nature of Class Grime.

- **RQ1.2:** What are the types of Organizational Grime?

  Rationale: This is a fundamental question of this research, inquiring as to the nature of Organizational Grime.

The development of this taxonomy helped us to understand the types of grime and how to design the studies that would allow us to address the stated problem. Specifically, we were concerned with the following Design Problems(DP):

DP1: How can we conduct a full-factorial experiment to evaluate the relationship between grime and quality and grime and technical debt when we are unsure how often design pattern grime occurs?

- **DP1.1:** How can we generate design pattern instances?
- **DP1.2:** How can we ensure design patterns have the correct type of grime?
- **DP1.3:** How do we measure the quality of a pattern instance?
- **DP1.4:** How do we measure the technical debt of a pattern instance?
- **DP1.5:** How do we automate the data collection and experimentation process?

Solving these design problems led to the development of the method described in Chapter 8 and its implementation. Solving DP1.1 led to the design pattern generation technique documented in Section 4.3. Solving DP1.2 led to the creation of the Software Injection approach described in Chapter 6. Solving DP1.3 led to our implementations of the Quamoco Quality Modeling [270] approach, and eventually, the SIG Maintainability Model [114] as described in 5. Solving DP1.4 led to the implementation of both the CAST [233] and Nugroho et al. [203] TD Estimation approaches as described in Chapter 5. Finally, solving DP1.5 led to the implementation of the Arc Framework as described in 3. The solutions to these problems led to the ability to address RQ2, using the Arc Framework and executing the simulation experiments.

The following two RGs address the heart of the stated problem via the simulation experiments. We refine these two goals into the two research questions. The experiments are described in detail in Chapter 10 wherein the research questions are further refined.

RG2: Analyze design pattern instances afflicted with design pattern grime for the purpose of evaluation with respect to the ISO/IEC 25010 Maintainability subcharactersitics [126], from the perspective of researchers, in the context of generated Java™ design pattern instances.

RQ2: How does each type of grime affect software product maintainability?

Rationale: Evaluate the assertion that as grime builds up in a pattern instance or software system, it will negatively affect the software or pattern instance's maintainability.

RG3: Analyze design pattern instances afflicted with grime for the purpose of evaluation with respect to the Technical Debt Principal and Interest, from the perspective of researchers, in the context of generated Java™ design pattern instances.

RQ3: How does each type of grime affect a software product's technical debt estimate?

Rationale: Evaluate the assertion that as grime builds up in a pattern instance or software system, it will increase the technical debt principal and interest.

Because the results of the simulation experiments were based on generated design patterns into which grime was injected, we cannot be sure that this is representative of actual grime in genuine software. Thus, a new set of Design Problems manifested.

DP2: How do we verify the results of the experiments with real software?

DP2.1: How do we incorporate design pattern detection into the Arc Framework?

DP2.2: How do we track instances of design patterns across versions of a software system?

DP2.3: How do we modify the Software Injector to allow for controlled injection to mimic what is has occurred in between versions?

The solutions to these design problems directly led to the capability to execute the verification studies described in Chapter 11. Specifically, the solution to DP2.1 as described in Chapter 4 led to the ability to incorporate and gather design pattern instance data from real software systems. Furthermore, the solution to DP2.2 as described in Algorithm 4.3 led to the ability to track patterns across versions. Finally, the ability to further control the injector was developed and, in combination with the prior solutions, allowed for the execution of the verification studies in Chapter 11. Thus, we could now handle addressing the following Research Goal:

RG4: Analyze design pattern instances for the purpose of comparing injected and observed instances of grime with respect to their ISO/IEC 25010 Maintainability subcharacteristics attributes and Technical Debt Principal and Interest from the perspective of researchers in the context of open source Java™ software projects.

RQ4: Do observed and injected grime have a similar effect on the Maintainbility subcharacteristics and Technical Debt Principal and Interest?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on Maintainability and Technical Debt Principal and Interest as the natural process of grime accumulation.

1.1.4 Empirical Research Methodology

In order to answer the knowledge questions posed, we used and extended the methods from empirical software engineering. Specifically, we developed an empirical method designed to understand software engineering phenomena. Towards the evaluation of this method, we employed taxonomy development to extend the structural grime hierarchies. Using these taxonomies, we conducted simulation experiments using software injection techniques and design pattern generation techniques we developed. Finally, we developed a technique to verify the experimental results, which term a verification study. For both the simulation experiments and the verification study, we needed to collect a large amount of data for which the specific tools and techniques did not already exist. Thus, the majority of this research endeavor was focused on developing the Arc Framework, which is an operationalization of the method itself. An overview of the methods employed to answer our research questions is further described in the following section.

1.2 Overview of the Dissertation

The body of this dissertation contains 11 chapters. Chapter 2 provides provides background on the current issues and concepts fundamental to the phenomena under study and the research methods used.

Chapters 3 – 7 describe the development of data collection and automation framework. Chapter 3 details the underlying data model, the integration of external tools, and the control of study workflows. This framework is derived from earlier publications at the 2011 International Workshop on Machine Learning Technologies in Software Engineering [104] and the 24th International Conference on Computer Applications in Industry and Engineering [105].

In Chapter 4 we describe the integration of an external design pattern detection tool and the implementation of design pattern generation used during the simulation experiments.

Chapter 5 describes the implementation of software quality models and technical debt estimation techniques. This work was based on earlier publications at the 8th International Symposium on Empirical Software Engineering and Measurement [101], the 6th International Workshop on Managing Technical Debt [103], and the 11th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement [129].

In Chapter 6 we describe the software injection technique utilized in the simulation experiments. This technique is based on work published at the 8th International Symposium on Empirical Software Engineering and Measurement [101]. Finally, in Chapter 7 we describe our approach to automated detection of design pattern grime, which is used in the verification studies.

Chapters 8 – 12 discuss the results of the application the empirical studies. In Chapter 8 we describe the empirical method we have developed to study software engineering phenomena and which guides the remaining chapters of the dissertation.

In Chapter 9 we define the taxonomies for Class and Organizational Grime. This work was published in the 8th International Symposium on Empirical Software Engineering and Measurement [101] and at the 12th Seminar on Advanced Techniques & Tools for Software Evolution [134].

In Chapter 10 we present the simulation experiments evaluating the effects of design pattern grime on maintainability and technical debt. These experiments were conducted on generated design patterns which were injected with varying levels of grime. Next, in Chapter 11 we present the verification studies wherein we compare the results of the natural evolution of design pattern instances to those which have been injected with similar grime. Finally, Chapter 12 concludes this dissertation with a summary of the essential findings and a path for future work.

CHAPTER TWO

BACKGROUND AND RELATED WORK

This chapter explores the main concepts and foundational work on which the proposed research is based. This includes design patterns and design pattern evolution, software decay, technical debt (TD), and software product quality. The chapter concludes with a section identifying the gaps in existing research.

2.1 Design Pattern Evolution

Design patterns were widely introduced to the software engineering community by Gamma et al. [96]. Design patterns are abstract solutions forged in experience to commonly recurring design problems. These patterns are a type of micro-architecture subject to evolutionary issues and design decay. However, few empirical studies of a relationship between design pattern evolution and decay exist in the literature. Rather, studies involving the evolution of design patterns tend to focus on how pattern change-proneness [19, 31, 98, 192].

In order to study design pattern instance decay a means to formally specify a pattern and validate instances is necessary. Various design pattern languages and specification techniques have been proposed [69, 72, 150, 151, 153, 195, 245, 247, 254] each with the same goal –a higher level of representational abstraction. Yet, although the specification aspects may well be understood, the verification of instances that conform to these specifications remains a hard problem.

The role-based meta-modeling language (RBML) is an approach to specify design patterns based on an underlying metamodel [90, 150, 151]. This meta-model extends the UML™ meta-model [1] which allows the instances to be visually described and constrained using the Object Constraint Language (OCL) [2]. The use of OCL allows the defined specifications to have a varying degree of generality. In order to make use of the specifications, a means to validate pattern instances against the specification is required.

Kim [148] initially proposed a method for evaluating the structural conformance of a pattern instance to the RBML specification. This proposal was followed by Kim and Shen's [149, 152] divide-and-conquer approach. Based on this approach, Strasser et al. [251] developed a tool to calculate a score for the conformance rating of a design pattern instance given its RBML specification. Recently, Lu and Kim [168, 169] have developed an approach to validate conformance of behavior and sequence diagrams of pattern instances. Kim and Whittle [147] utilized RMBL to help generate designs using design pattern specifications.

2.2 Software Aging and Decay

Software evolution describes those processes which affect changes that refine the requirements and functionality of a software system. *Software decay*, a specific form of software evolution, describes a system that has evolved to become "harder to change than it should be" [71]. Parnas [210] later identified a complementary phenomenon known as *software aging*. Software aging describes the effects on system value due to changes in the system's environment. Several studies have been conducted on software decay and aging, as well as on the rejuvenation of software as a means to circumvent the effects of these phenomena [71, 106, 123, 205, 261].

2.3 Technical Debt

Technical Debt is a concept introduced by Ward Cunningham [58] as a financial metaphor to describe the trade-off between quality engineering and satisfying short-term goals. The following subsections describe work in the following areas describing the nature of the metaphor, methods of managing Technical Debt, impact and consequence of Technical Debt, and techniques for measuring Technical Debt.

2.3.1 Metaphor, Definition, and Properties

The notions surrounding technical debt until recently have been informal and underspecified. In place of this Tom et al. [258] conducted a systematic literature review to consolidate the concepts surrounding Technical Debt into a single taxonomy. This taxonomy classifies Technical Debt from either of two perspectives: by the underlying intention behind the decision (or lack thereof) to take on the debt or the type of artifact in which the debt occurs.

The intentional perspective is divided into *Strategic Debt, Tactical Debt, Incremental Debt*, and *Inadvertent Debt.* Strategic Debt is debt taken on intentionally as part of a larger long-term strategy. Tactical Debt is debt taken on intentionally as a reactionary response and satisfies short-term needs. Incremental Debt is debt taken as several small steps but accrues very easily and rapidly. Finally, Inadvertent Debt is debt taken on unintentionally and possibly unknowingly by the software development team. The location or artifact perspective is divided into *Code Debt, Design and Architectural Debt, Environmental Debt, Knowledge Distribution and Documentation Debt*, and *Testing Debt.*

Beyond classifying and understanding how debt occurs, some researchers have furthered understanding the metaphor itself. Nugroho et al. [203] indicate that the technical debt metaphor has several contexts from which it can be viewed, and they specifically look at it from the context of maintainability. Along similar lines Klinger et al. [154] look at Technical Debt from the perspective of enterprise development and indicate that using financial tools, decision theory, stakeholder-based quantification, and developing an understanding of unintentional debt are potential avenues of interest. Finally, Theodoropoulas et al. [257] view Technical Debt from the stakeholder perspective and provide a new definition based on the gap between an organization's technology infrastructure and its impact on quality.

More recent work has looked into the extent and practicality of the technical debt metaphor itself. Specifically, Schmid [236] [235] notes that as we explore technical debt the metaphor begins to breakdown. He notes, the intimate connection between future development and Technical Debt leads to an inability to measure Technical Debt itself objectively. This is due to the nature of the Interest property associated with technical debt items. Since technical debt interest has a probability indicating whether it may affect the system, we should instead focus not on measuring all Technical Debt (*Potential Technical Debt*). However, rather we should concern ourselves with the debt items that will have an impact (*Effective Technical Debt*) on upcoming feature development or maintenance.

2.3.2 Technical Debt Management

Technical Debt Management consists of several activities [164]. The primary focus has been identifying, cataloging, and remediation of debt items. Current industry practices includes identifying and tracking debt as part of the working project backlog [42, 157, 191] or as part of a separate technical debt list [107, 108, 239]. Essentially, we can think of the emergence of design disharmonies within a software system akin to taking on debt, and the longer they are allowed to remain (without refactoring), the more negative influence they will have on the system [84]. This influence acts as interest on the debt by increasing the amount of effort required to evolve the software [289].

Guo and Seaman [107, 108, 239] proposed a technical debt management framework (TDMF). Central to this framework is the Technical Debt List (TDL) stores information about known technical debt items within a software system. Three activities support this framework: Technical Debt Identification, Technical Debt Estimation, and Decision Making. Recently, Guo et al. [109] conducted a case study to evaluate the costs of using the TDMF. This study showed that after an initial high startup cost, the cost of monitoring and remediation of debt reduces to a reasonable level. Holvitie and Leppänen [118] have further enhanced the TDMF with an approach called DebtFlag. The main purpose of the DebtFlag is to reduce information redundancy to provide a more efficient debt propagation evaluation. This aids in a more accurate estimation of debt impact, interest, and interest probability.

Schmid [235–237] has also focused on developing an approach for selecting which debts should be removed. Schmid's work is based on a formalization of technical debt concepts to extend the TDMF using a 2D matrix representation coupled with an approximation scheme to select those technical debt items to refactor in the next release. Similarly Stochel et al. [250] approach this problem using a subsumption model of Technical Debt based on a modified Value Based Software Engineering [34] cost and estimation approach in order to estimate the return on investment (ROI) for each item. A technical debt versus portfolio assessment matrix, using ROI in a similar approach to that of Seaman and Guo [239], is used to evaluate each item provided the best savings per release (similar to that of Schmid [235–237]).

Decision support approaches for debt acquisition have been less forthcoming than for debt repayment. Nevertheless, Falessi et al. [73] are exploring current open problems concerning this topic as well as the required decision support constructs needed to address the problem. Ramasabba and Kemerer [215] developed an optimization approach utilizing multiple projections of a single codebase to evaluate decisions regarding both debt acquisition and repayment. Griffith et al. [102] conducted a simulation study of TD management strategies. These simulations showed that combining automated detection with a maximum TD threshold and remediation sprints is a superior combination. Furthermore, the models explored in the simulation study are representative of the models identified as used in practice by Martini et al. [186, 187].

As the technical debt landscape has evolved the research community's focus has moved from identifying technical debt to the underlying issues surrounding these items.

Specifically Falessi and Voegele [74] have recently conducted a case study to evaluate industry perspectives on design rule priority and validation. Here design rules are considered to be any empirically validated design principles that enhance software quality. This study found that classes with a high count of rule violations also tend to be defect-prone. Additionally, Tufano et al. [264] conducted an extensive case study on 200 open source systems by mining revision history. They found that code smells, known to affect the maintainability of a software system, are typically introduced to create the affected artifact. Furthermore, they found that developers at all levels are prone to creating code smells and that these smells are introduced more often due to time constraints. Furthermore, Mamun et al. [4] have also identified the primary causes for technical debt accumulation as: time constraints, hardware/software integration issues, improper or incomplete refactorings, or use of legacy, external, or open-source libraries. Additionally, Digkas et al. [64, 65] have shown that overall TD density of a project can be reduced when adding new code, which is *cleaner* then the overall project, a result corroborated by Freire et al. [95].

Recently, the InsightTD [227] project has conducted several global surveys in order to understand TD Managment from the practitioner point of view. This work utilized a replicable industry survey which has been executed with practitioners from Brazil [224, 226, 227], Chile [213], Columbia, the United States, and Serbia [175, 216]. This work has helped to illuminate how practitioners worldwide experience and understand technical debt and confirmed prior results [4, 11, 52, 53] while avoiding the limitations in scope of similar approaches [12, 17, 51, 173, 204, 246, 248]. It has also provided insights into technical debt causes and effects both in general [224, 226, 227] and from multiple points of view [214, 223] and levels of experience [91, 174]. Additionally, these researchers have provided tools and techniques useful to integrate TD Management, identification, and evaluation into the development lifecycle when considering different process models [92, 222]. Furthermore, this work has explored the impediments to and actions for removing technical debt [93–95].

2.3.3 Impact and Consequences

The impact of Technical Debt on engineering effort, project cost, and project quality is of utmost concern. An initial empirical inquiry conducted by Zazworka et al. [289] shows that Technical Debt negatively impacts software quality. Furthermore, Zazworka et al. [288] investigated prioritizing debts using a cost/benefit analysis approach. This study shows that technical debt negatively affects the correctness and maintainability. Recently, Griffith et al. [103] conducted a case study across several versions of several open-source Java™ systems evaluating the relationship between quality model attributes and technical debt measures. Results show little evidence of a relationship between the CAST [59, 60], SonarQube™ [99], or Marinescu's [181] approaches to measuring technical debt principal and quality attributes in the QMOOD quality model [25].

A key to understanding technical debt and its effects is to be able to understand the gaps and overlaps that may exist in the landscape [131] of TD item types. Zazworka et al. [290] identify several types of design debt (e.g., code smells, modularity violations, and design pattern grime) and tools which detect them. They identified that all the tools indicate different problems with little to no overlap. Further exacerbating this issue is the work of Alves et al. [7] which has further defined the technical debt landscape. Furthermore, the works of Li et al. [164], Alves et al. [6], and Rios et al. [225] have expanded landscape has been divided into 17 subtypes of Technical Debt (including Security Debt [124, 184, 221]), each consisting of multiple indicators. The original landscape, as proposed by Izurieta et al. [131] is now encompassed within the subtype of *Design Debt.* Fontana, Ferme, and Spinelli [84] state that although code smells are important components of the technical debt landscape, certain identified debt items may not constitute debt. Instead, they indicate that domain knowledge must be used as a filter to identify these misnomers and ensure that an accurate indication of Technical Debt is provided.

2.3.4 Design Debt

Industry practitioners, being closer to the source code, typically focus on Code and *Design Debt* [11]. Design disharmonies are a form of software decay that have been categorized in order to understand their nature. These categories are separated by the level of abstraction in which they occur (e.g., statements, methods, classes). They may also be categorized by the software artifacts affected, e.g., source code, unit tests, or databases. We discuss design defects affecting software systems at the statement, method, class, pattern, module, and system level in the following subsections.

2.3.4.1 Code Smells

Fowler et al. [88] initially described 22 code smells which indicate (possibly vehemently) that refactoring should be performed. These descriptions also included possible corrective refactorings. Since then, several others have extended this library of code smells. Kerievsky [138] added 5 additional code smells and helped to further explain several of the original code smells, while providing several new corrective refactorings. Mäntylä [201] and Mäntylä et al. [177] describe a taxonomy re-classifying the original 22 code smells based on how each affects a system.

2.3.4.2 Design Pattern Disharmonies

Initially, Moha et al. [199] defined a taxonomy of potential design pattern disharmonies and conducted an empirical study to investigate their existence. This taxonomy includes the following four types of defects: *Missing* refers to a design missing a needed design pattern. *Deformed* patterns are those which are not correctly implemented according to Gamma et al.'s [96] definition but which are not themselves erroneous. *Excess* is the overuse of design patterns in a software design. *Distorted* design patterns are distorted instances of a design pattern. Their study was conducted across several versions of an open-source Java™ project. They detected 38 design patterns instances, of which three were found to be non-harmful deformed design patterns. Furthermore, their research presented and evaluated multiple detection techniques, including manual, semi-automatic, and automatic techniques based on a combination of detection strategies and constraint satisfaction techniques. Unfortunately, this taxonomy was not formally defined.

Izurieta and Bieman [127] presented another taxonomy of design pattern decay. Seminal work by Izurieta [132] found that pattern realizations tend to accumulate artifacts that obscure the intended use of patterns. Two distinct categories of design pattern decay were identified:

Design Pattern Grime – the accumulation of unnecessary or unrelated software artifacts within the classes of a design pattern instance.

Design Pattern Rot – violations of the structure or architecture of a design pattern.

Compared to the Moha et al. taxonomy, grime relates most closely to the concept of deformed patterns, while rot relates most closely to distorted design patterns. Empirical studies showed only the presence of grime, which has led to the further development of three types of grime: *modular*, *class*, and *organizational* grime, each defined as follows:

Modular grime build-up of relationships involving the classes of a design pattern instance, where the relationships are unnecessary to facilitate the operation of the pattern.

Class grime build-up of fields and methods in the classes of a pattern instance, where these artifacts are unnecessary to facilitate the operation of the pattern.

Organizational grime the unnecessary distribution of pattern instance classes across namespaces or packages.

Empirical studies further showed only significant results for modular grime [128].

The modular grime results led Schanz and Izurieta [234] to further expand the taxonomy of modular grime. A series of empirical studies across open source systems was conducted to validate the existence of these types of grime. Further empirical studies on grime have shown implications in the area of testing [128]. Based on this work, Izurieta et al. [131] indicated that the technical debt landscape should include design pattern decay along with other types of design defects, such as code smells, anti-patterns, modularity violations, and specific lower-level code issues that affect design patterns.

Prior work involving design pattern grime has been conducted by Dale and Izurieta [62] and Griffith and Izurieta [101]. Dale and Izurieta evaluated the effects of modular grime on Technical Debt. Their work show, through experimentation, that temporary modular grime types have the most significant effect on Technical Debt. Griffith and Izurieta further developed the class grime taxonomy and showed, through experimentation, that each type of class grime negatively affects the understandability of a pattern instance. These studies utilized an early form of software injection to facilitate the experimental process. Specifically, Dale and Izurieta used a method that injects modular grime into Java™ bytecode [62]. Griffith and Izurieta used a method that modifies a model of the source code [101].

Later, Feitosa et al. [76, 77] have explored several aspects of grime in respect to industrial software projects. These studies first explored the evolution of modular, class, and organizational grime within five industry software systems. Their results showed that, in general, grime tends to grow linearly and similarly across a project. However, its particular accumulation location depends upon both the type of pattern and responsible engineer. In a later case study, Feitosa et al. [76] showed that there are strong correlations between class and modular grime with a decrease in correctness, moderate correlations between class and modular grime with a decrease in performance, where these quality characteristics are evaluated using static analysis tools. Additionally, they found that this result depended on the project, pattern type, and level of grime. Furthermore, they found that grime tends to show higher correlations with pattern classes that have a larger number of rule violations.

More recently, Reimanis and Izurieta have extended the grime beyond the structural and into the behavioral domain [134,219,220]. In this work, Reimanis and Izurieta developed a behavioral grime taxonomy over the modular grime taxonomy based upon the behavioral deviations including *Excessive Actions* and *Improper Order of Sequences*. They conducted a case study across 20 versions of 5 open source Java™ software systems, and their results of these studies indicated that behavioral grime has significant effects on the reusability of design patterns.

Another line of research into design pattern disharmonies has been conducted by Bouhours et al. [36–38]. They have studied what they term *spoiled patterns* [37]. Spoiled patterns are the results of incomplete or failed instantiation of a design pattern, intentionally or unintentionally. This research is motivated to improve design pattern education, to motivate better indication of when patterns need to be refactored, and to improve forward-driven and evolutionary design techniques [36]. Bouhours et al.'s study involved the manual collection of spoiled patterns based on student implementations rather than those from open source or industry software [36–38].

2.3.5 Design Debt Identification

The notion of design debt identification has been around nearly as long as the notion of design defects themselves. Detection efforts can be broken down into three major approaches: metric-based approaches [179,193,200,265,272], machine learning and artificial intelligence methods [139,141,144,145,212,232], and a combination of structural information and metrics [196–198].

One of the most widely extended methodologies is the *detection strategies* approach proposed by Marinescu [179]. A *detection strategy* is a filtering method that utilizes a combination of metric thresholds and set theory to identify probable locations of design disharmonies in code. Ratiu et al. [217] extended the detection strategy approach by including history and evolution information. This approach increased code smell detection accuracy by observing metrics across multiple versions rather than a single version. Ratiu et al. [218] and Gîrba et al. [110,111] utilize Formal Concept Analysis in order to allow historical analysis and change analysis to be coupled to the original detection strategy framework.

Following Marinescu, Munro [200] also used product metrics to help define detection algorithms for design defects. Munro, however, took it a step further (towards formalization) by defining a template to describe each design defect. Where the template consists of: bad smell name, measurement/process for detection, and an interpretation (set of rules) defining the defect [200].

To better understand the nature of code smells and to improve detection techniques Pietrzak and Walter [211] investigated the possibility of inter-smell relations. This work paved the road to formalizing the idea of relationships that exist between design disharmonies. Walter and Pietrzak [272] conducted additional research utilizing multiple criteria vectors including programmer experience, metrics, coding rules, historical information, and other detected code smells in order to increase detection capabilities.

Along the lines of further understanding of design disharmonies, Moha et al [196–198] conducted a domain analysis to develop a domain-specific language for detection rule definitions, a process called DECOR. This model was designed to encompass the notions of metrics, inter-relationships, and structural features.

An extension of Moha et al.'s DECOR approach was the HIST tool of Palomba et al. [209]. Polomba et al.'s approach utilizes historical information to detect code smells and anti-patterns, which normally could not be detected. HIST was evaluated on the change histories of several large open source Java™ projects in order to provide proof-of-concept. The authors note that the main limitation in their approach is the requirement of having a sufficiently long version history.

Due to the typical nature of design defects definitions as informally specified issues in designs or code, several approaches have been proposed to automate developing the algorithms for detection. This notion of automation has progressed from the semi-automated to fully automated generation of detection algorithms. Mihancea and Marinescu [194] were the first to propose such an approach. They used a genetic algorithm to tune the parameters for each filter to improve the accuracy of detection strategies.

In order to deal with the issues of manual or semi-automatic design defect detection, several approaches have been developed. Kessentini et al. have developed and evaluated numerous approaches to these problems [139–141, 172, 176, 208]. In these studies, supervised learning approaches are used to generate detection rules. The results show that rules generated using simulated annealing, harmonic search, genetic algorithms, and genetic programming approaches all outperform the results of the original DECOR [196] rules on DECOR's training data.

Mahouachi et al. [172] extend the genetic programming approach by including both detection and correction (via refactoring) together in order to improve both steps simultaneously. Mansoor et al. [176] have also extended the genetic programming approach to utilize a multi-objective approach to increase both precision and recall of the generated rules. In addition, other machine learning approaches have been developed. Recently, Fontana et al. [16, 87] have detailed an approach to use machine learning techniques for code smell detection. This work facilitated the development of a benchmark dataset and the evaluation of multiple classification algorithms against existing tools. Their results show that high accuracy can be achieved using various classification techniques, given that training data exists.

Given the reliance of detection approaches on software metrics, it is surprising that there is little empirical research into the feasibility of these approaches. Schumacher et al. [238] conducted an empirical study of automated detection within an industry setting. Using the CodeVizard tool [287], which is based on a metric-driven approach to automated detection, they found that in comparison to human classification, the automated detection performed very well. They also identified that combining automated detection with human review decreases the required maintenance effort. Further research into the evaluation of automated detection was conducted by Fontana et al [83]. This latter study compared four code smell detection tools across six versions of a single Java™ open source project. This study shows that the tools evaluated had a tendency to disagree and that although these tools may prove helpful, they are far from adequate.

2.3.6 Design Disharmonies and Quality

A large body of prior research exists concerning the relationship between design disharmonies and software quality. A large portion of this research has focused on the effects of code smells and anti-patterns on software maintainability. Early work was conducted by Olbrich et al. [207] including two longitudinal case studies across the version history of two open-source software projects. They showed that the affected classes are more likely to change, and the studied code smells negatively impact maintainability. A more extensive study by Khomh et al. [143] conducted a similar longitudinal case study but considered the relationships between change-proneness and 29 different code smells. The results of this study similarly showed that affected classes are highly change-prone.

Olbrich et al. [206] conducted another longitudinal study across three open-source systems. They found that the instances of god classes and brain classes studied exhibit less change and defects when normalized for size in the systems studied, contradicting previous results. Later, Khomh et al. [146] conducted a study on 13 anti-patterns in several releases of 4 open-source software systems. The results show that those classes affected by anti-patterns are more change- and fault-prone than others, when accounting for size. A further study of 16 open source Java™ system change histories conducted by Romano et al. [228] showed that changes are more common in those classes affected by anti-patterns. Another study by Yamashita and Counsell [281] found that code smells are significantly affected by the size, making comparisons between systems of varying sizes impossible.

Unfortunately, change- and fault-proneness are not comprehensive indicators of maintainability. Because of this, Yamashita and Moonen [282] conducted an empirical study to connect code smells to maintainability. This study connected maintainability factors defined by experts to developer impressions identified during their industrial case study. Yamashita and Moonen conducted a second multiple case study to develop the connection between code smells and maintainability [284]. Both studies found that typical indicators such as change size or complexity are not enough to assess the ability of code smells to predict maintainability issues. A study by Sjoberg et al. [244] further refined these results while also indicating that maintenance effort was not significantly affected by studied code smells. Each of these studies indicated that interactions between code smells should be studied to understand better how maintainability is affected [280, 282, 284].

Following this research, Yamashita and Moonen [283] evaluated the effects of inter-smell relations (previously identified and studied by Pietrzak and Walter [211] and Fontana and Zanoni [85]). This study was conducted across four Java™ systems known to have code smells. They found that when artifacts are affected with multiple code smells these smells tend to interact and affect maintainability. Furthermore, Yamashita and Counsell [281] found that there is no difference between smell co-location and coupling when considering the effect on maintainability. Overall, they found that a code smell based approach to maintainability assessment is superior to a metrics-only approach.

A more quantitative approach to evaluating the effect of code smells on software quality was conducted by Fontana et al. [86]. This study was conducted across the set of Java™ open-source systems collected in the *Qualitas Corpus* [255]. Their results indicate that the most prevalent code smells are Duplicate Code, Data Class, God Class, Schizophrenic Class, and Long Method, not discounting false positives due to tool error. They also show a greater indication of deterioration in maintainability in those systems with a high number of code smells. Finally, the research by Fontana et al. indicates that there is a connection between the system domain and the effect that code smells have on maintainability, a finding that is confirmed by Hall et al. [112].

Bán and Ferenc's study [43] was conducted using 228 open-source Java™ systems and PROMISE data concerning bug information for 34 of the systems. This study investigated the relationship between maintainability and anti-patterns and the correlation between anti-patterns and identified bugs. The results of this study showed that there is a positive correlation between anti-pattern affected areas of code and bug incidents, and that there is a negative correlation between anti-patterns and maintainability (as measured using the Columbus quality model [24]).

Most research has indicated that both code smells and anti-patterns affect quality by negatively impacting maintainability. However, only a single study has utilized a known quality model to conduct this evaluation [43]. Furthermore, all of the studies to date have been case studies, and the results have been restricted to either qualitative analysis or in the quantitative approaches, only correlation analysis. Given this, it is pertinent that an approach is necessary that will facilitate experimentation to provide an estimation of effect size and causal analysis.

There is also an issue in that current approaches are limited to the evaluation of only those items that can be detected by existing tools, thus limiting the analysis to code smells and anti-patterns. This is indicative of the need for an approach that can formalize definitions of design disharmonies in a generalizable way. Finally, there is no evidence regarding the relationship between design disharmonies and any other quality characteristics defined in the ISO/IEC 25010 specification [126] such as: *Functional Suitability*, *Reliability*, *Performance Efficiency*, *Usability*, *Security*, *Compatibility*, and *Portability*.

2.3.7 Measurement

Lastly, there must be a means to measure Technical Debt and its associated properties in a way that is meaningful to developers and stakeholders alike. Seminal work by Brown et al. [42] identified the technical debt metrics of: principal, interest, and interest probability. Subsequently, Nugroho et al. [203] contributed a formal model to calculate measurements for both interest and principal from a maintainability perspective. Additional measures, closely related to the technical debt landscape [157, 290] have been proposed to index the effect that design flaws (e.g., code smells and modularity violations) have on Technical Debt. For example, Marinescu [180] proposes a method to index the effect on quality produced by different code smells and anti-patterns based on the type, influence, and severity of the design flaw instance, thus creating a score that can be aggregated over the size of the system. In another approach, Nord et al [202] develop a strong foundation for measuring the Architectural Technical Debt based on the notion of prudent, deliberate, and intentional debt.

Letouzey [161] developed the SQALE quality and technical debt analysis model, which provides both the ability to estimate technical debt principal and several visualizations to illuminate the impact of Technical Debt. Notably, in a recent study Kosti et al. showed demonstrated that one can effectively estimate TD Principal as measured by SQALE using a collection of seven well-known maintainability related metrics [155]. Additionally, Curtis et al. [60] proposed methods to estimate the principal and interest as well as the size, cost, and type of Technical Debt. Other tools and technical debt estimation approaches have been developed for an introduction and evaluation of these tools, we suggest Amanatidis et al.'s [8] study comparing and benchmarking of these tools or Avgeriou et al.'s [22] recent work comparing existing TD measurement tools.

Given these various approaches for the quantification of Technical Debt and the wide range of differences in values, Izurieta et al [130] proposed a means to measure the error associated with the calculation of Technical Debt for these methods. They argue that a means to measure the systematic error introduced by these tools should be included with their values, similar to other scientific tools, and that a means to compare these tools and their error be developed. Although, a bit ahead of its time, serious questions have come to light for the most popular tools such as SonarQube and their estimates of TD Principal [231]. Additionally, there is a growing concern among researchers of the lack of consensus in TD evaluation between available tools, which has led to a lack of trust in industry [22, 159, 231, 253].

Previous research focus has been on the measurement of technical debt principal, but more recent research has turned towards measuring interest [10, 20, 46, 47, 185, 188, 243]. This shift in direction is due to research indicating that large principal values do not convey an understanding of the effect that Technical Debt will have on a project. However, Ampatzoglou et al. [13] have recently shown that artifacts showing similar levels of principal show similar levels of interest. Additionally, they have shown evidence of a relationship between TD Principal and Interest. Chatzigeorgiou et al [47] are developing an optimization technique which attempts to optimize the timeline for repayment based on historical data and metrics.

2.4 Software Quality

Quality models provide references against which software components can be measured. Theoretical models such as ISO specifications [125] [126] go only as far as offering guidelines along many dimensions of quality which must be operationalized to provide a working solution that the engineering community can use. A common criticism of theoretical models is that they are too ambiguous to be directly measurable. A comprehensive description of quality models is beyond the scope of this dissertation. However, Wagner [269] and Ferenc et al. [79] provide a significant account and history of quality models. We selected three operationalizations of ISO theoretical models: SQALE [161], Quamoco [270] and the SIG Maintainability Model [114].

In the last decade, the research community has also observed how Technical Debt has become a popular approach, supported by many tools, to track the progress of source code development by pointing out disharmonies (i.e., code smells) that need refactoring [63, 74, 180]. Their remediation can either be undertaken immediately or scheduled for a later date at the expense of incurring debt (i.e., principal and interest) [7, 42, 157, 203]. Technical Debt should not be confused with software quality, as the former is a metric that only characterizes the maintainability of a system. The new definition of Technical Debt (16K definition) explicitly states that "*technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.*" [21]. Although the focus of the definition is on only one aspect of the many dimensions that make up ISO based quality models, it is important to mention that the SQALE method to managing technical debt associates remediation costs that affect the technical debt index of a system by using a remediation function that takes into account all dimensions of quality, not just maintainability [160]. The SonarQube™ operationalization of SQALE deviates from its calculation of Technical Debt by only focusing on the technical debt ratio associated with maintainability. For a more comprehensive comparison between technical debt calculations and quality assessment approaches, see Griffith et al. [103].

2.4.1 SQALE

The SQALE (Software Quality Assessment based on Lifecycle Expectations) quality model is a generic approach to modeling software quality and can be applied to any language. It is based on the ISO/IEC 9126-1:2001 standard [125] (further referred to as ISO 9K). The approach is based on eight code characteristics organized chronologically in pyramidal form. At the bottom of the pyramid is testability, followed above by reliability, changeability, efficiency, security, maintainability, portability, and reusability at the top of the pyramid. Quality requirements such as “Exception Handling shall not catch Null Pointer Exception” are associated with characteristics in the pyramid and have a remediation cost. If more than one characteristic is affected by a quality requirement, then an association with the lowest characteristic is formed. The characteristics at the bottom of the pyramid represent more important quality dimensions and are meant to aid practitioners when prioritizing requirements that need refactoring in the code base. SQALE is published under the open-source license, and several vendors implement it. This case study uses the implementation of SonarQube's plug-in as it has become widely adopted by organizations.

2.4.2 Quamoco

The Quamoco quality model was developed explicitly as an extensible meta-model. Its goal was to bridge a gap between abstract concepts and measurable attributes. The central concept of the model is a factor meant to represent an attribute or property of an entity, where the latter represents an essential aspect of quality we want to measure. Two types of factors exist, quality aspects and product factors. The former represents the more abstract qualities found in theoretical models such as the ISO standards. The latter represents the measurable parts of a software component and impacts their associated quality aspect. Factors form hierarchies; where factors can further refine some aspects of quality. The meta-model is split into modules (to improve modularization), where the root module contains general quality hierarchies and fundamental product factors. This allows practitioners to extend the root module for specific purposes or technologies and to focus on the qualities relevant to their specific needs.

Quamoco's base model is an instantiation of the meta-model and uses the ISO/IEC 25010:2011 [126] (further referred to as ISO 25K) as a reference. It is the result of many years of collaboration by quality experts from industry and academia, and it is comprised of a comprehensive set of factors and measures that capture software quality assessment.

2.4.3 SIG Maintainability Model

The SIG (Software Improvement Group) Maintainability Model represents a programming language agnostic operationalized hierarchical quality model focused on evaluating maintainability [23, 55, 63, 114, 170]. The original model was based on the ISO/IEC 9126 Software Quality standard [125]. This model was further improved, based on survey results, to improve the reliability and accuracy of representation in the selection of the underlying properties. In 2011, the ISO/IEC 9126 standard was replaced by the ISO/IEC 25010 standard [126]. Eventually, the SIG Model was updated to the current definition of Maintainability and its sub-characteristics: Analyzability, Testability, Modularity, Modifiability, and Reusability [23]. The current version of the SIG Maintainability Model is in full compliance with the ISO/IEC 25010 Standard [267, 268]. The model itself is designed to be straightforward to interpret.

2.4.4 High Level Differences between Quality Models

Quamoco, SQALE, and SIG are hierarchical models. Quamoco and SQALE both link issues found in software to quality aspects and sub-aspects, and both use this information as a means to evaluate the quality of a software component. On the other hand, the SIG Maintainability Model uses metrics to form properties that are then combined to form quality aspects and sub-aspects to evaluate the quality of a software component. to The more prominent differences between these models are:

- Quamoco is defined using a meta-model that characterizes quality models defined for different circumstances. SIG and SQALE are limited to the model imposed.
- SIG is limited to a weighted summation of a distinct set of metrics. SQALE is limited to the aggregation of effort estimates through remediation functions. Quamoco is

designed to incorporate weighted aggregation, ranking, and a variety of functions to describe the influence between aspects of the quality model.

- Quamoco models are defined in separate files and are hierarchical (in the sense that one model can inherit from another). Both the SIG and SQALE models are proprietary and built into the system.

- Quamoco models are defined such that an evaluation of the current level of quality can be provided without coaxing issues or rule violations into a unit-based measure. SQALE utilizes remediation effort as an index of quality, but its proprietary nature utilizes predefined values for each issue without the ability to easily tune or parameterize those values.

- SQALE is based on ISO 9K, while SIG and Quamoco is based on ISO 25K.

- SQALE and Quamoco focus on all characteristics of internal product quality, while the SIG model focuses only on Maintainability.

2.5 Research Gaps

The management of design pattern decay forms an important component in managing software aging and Technical Debt, thus warrants further research. The following is a list of research gaps that have been identified in this area:

- Design Pattern Grime Taxonomy – Further exploration of organizational and class grime types is necessary. Initial studies into these types of grime have not yielded any significant results, but unlike modular grime, the taxonomy for these types has never been fully developed.

- Quality – The impact of grime on the quality of both software products and pattern realizations has only been subject to limited study [61, 101, 127, 128, 133].

- Technical Debt – Current research has looked into how grime plays a part in the technical debt landscape [131]. The effect of grime on the technical debt value of a software product and pattern instances has only been studied for modular grime [61].
- Relationships – The notion that different subtypes of grime can be interrelated or that subtypes of grime and design defects types can be related are another potential area of study.
- Automation – The ability to detect grime is manual and time-consuming process. This is partly due to a lack of detection tools required to identify instances of grime embedded in design patterns realizations.
- Empirical Studies – Only a small body of work concerning empirical inquiry of design pattern evolution and decay has been conducted. Only a small selection of systems have been studied of these studies. We expect to expand on the number of case studies that address design pattern-specific issues across a diverse body of software in several languages.
- Experimentation – with the exclusion of machine learning experiments, little to no research has been conducted to develop methods for experimentation in design disharmony detection and relationships to quality.

2.6 Research Contributions

Given the gaps identified in Section 2.5 this research proposes the following contributions.

1. The development of a framework for the automated collection of software artifact data, software issue data, design pattern data, and software measurement data (see Chapter 3).

2. The improvement of design pattern data collection and the identification of design pattern chains, pattern instances across versions (see Chapter 4).

3. The development of a technique for generating design pattern instances for the facilitation of design pattern experiments. This technique culminated in the development of domain-specific languages for both RBML and a pattern generation cue language (see Chapter 4).

4. A formal benchmarking approach which injects disharmonies and pattern instances into software systems to provide both a "gold standard" to compare detection algorithms against and provide the basis upon which experimentation can be conducted. Our Contribution in this area includes the definition of a formal meta-model (see Chapter 6).

5. An approach to automate the detection of design pattern grime, culminating in a tool that detects each type of grime (see Chapter 7).

6. The development of a general process for research into software artifact phenomena and their effects (see Chapter 8).

7. The development of an expanded taxonomy of design pattern grime (see Chapter 9).

8. Analysis of the impact of grime on software product quality measures. The quality measurement tool has been developed (see Chapters 5 and 10).

9. Analysis of the impact of grime on the technical debt value of a given software product (see Chapter 10).

CHAPTER THREE

THE ARC EXPERIMENTATION FRAMEWORK

The study of software engineering is a laboratory science.

–Professor Victor R. Basili

3.1 Introduction

This chapter details the inner workings of the Arc Experimental Framework. The Arc framework, simply put, is a tool execution and data aggregation framework. This framework was designed and implemented with the following goals in mind: (i) execution of internal and external tools, (ii) data model definition for tool results, (iii) aggregation of these results into the higher-level measurements, and (iv) process automation. Other tools, such as SonarQube[1], Jenkins[2], and TEDMA [80, 81], share these goals, and although they provide similar capabilities, they do not provide the level of detail and level of automation provided by the Arc framework.

Organization This chapter is organized as follows. Section 3.1 describes the underlying architecture of the Arc Framework. The framework defines a data model through which all of our data may be persisted, described in Section 3.3.2. The described data model, to be useful, is utilized by a tool execution and workflow-based process, as discussed in Section 3.3. Section 3.4 describes the integration of various Java™software analysis tools into the Arc framework. Finally, Section 3.5 concludes this chapter.

[1] `https://sonarqube.org`

[2] `https://jenkins.io`

3.2 Arc Architecture

The Arc framework provides a platform to collect software engineering artifact data. This data is to be used to conduct or prepare to conduct experiments and observational studies. These studies necessitate that we collect data related to software projects (across multiple versions), including software metrics and their measurements, results of static analysis, results of pattern detection, and results from software quality analysis. Although each of these types of data is necessary for our work, we also note that the framework must be extensible to provide the capability to add new forms of data later.

Each of these data requirements leads to the underlying architecture for the Arc Framework, as depicted in Figure 3.1. This Figure is a very high-level conceptual view, which shows the various data types contained within the *Arc Db* provided via a series of commands/tools executed using a workflow process, and interacting with a well-defined data model. The key to this framework is the workflow and command structure.

3.3 Workflows

The workflow process of the Arc Framework is based upon the metamodel depicted in Figure 3.2. The basic component of this model is the *Workflow*, which contains a sequence of *Phases*, each of which can contain a list of *Command*(s). Along with Commands is the concept of *Collectors*. Essentially, Commands are the component which executes any portion of a workflow to prepare the System for the collection of data. Data collection provided by a subtype of Command known as a Collector, which collects tool output into the data model.

The *Command* structure, as depicted in 3.2, is divided into three distinct parts. The first are the *ToolCommand*(s) which are based on an externally defined command line driven tool. Second, are the commands in the *AbstractCommand* hierarchy consisting of the five subtypes: *PrimaryAnalysisCommand*, *SecondaryAnalysisCommand*, *RepositoryCommand*,

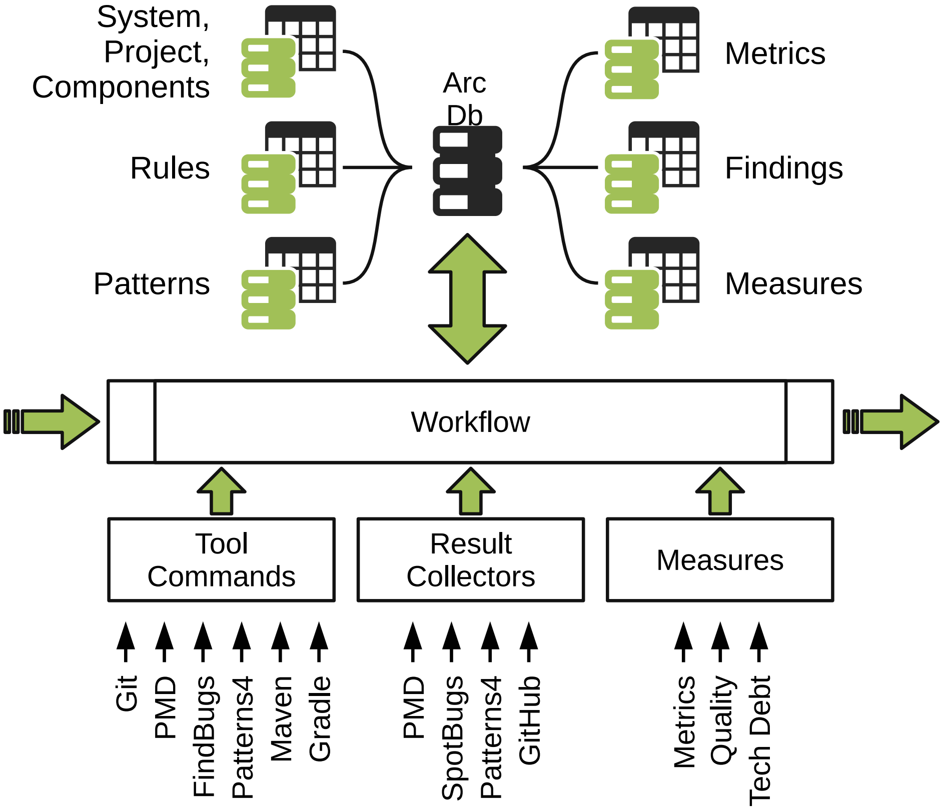


Figure 3.1: Illustration of the overall Arc conceptual framework.

ArtifactIdentifierCommand and *BuildCommand*. Finally, there are the Collectors which extract data from results files into the data model. We next further describe the *AbstractCommand* hierarchy.

The *AbstractCommand* hierarchy decomposes non-tool command types into five distinct subtypes. The first is *PrimaryAnalysisCommand* is the base conceptual class representing a tool or command which collects data related to immediately observable aspects of a software project. An example of such an aspect, are basic code metrics such as Lines of Code or Number of Methods. Thus, an instance of a PrimaryAnalysisCommand would be a metrics measurement tool or tool which extracts the underlying structural information of a software project. Next, the *SecondaryAnalysisCommand* is the base conceptual class representing a

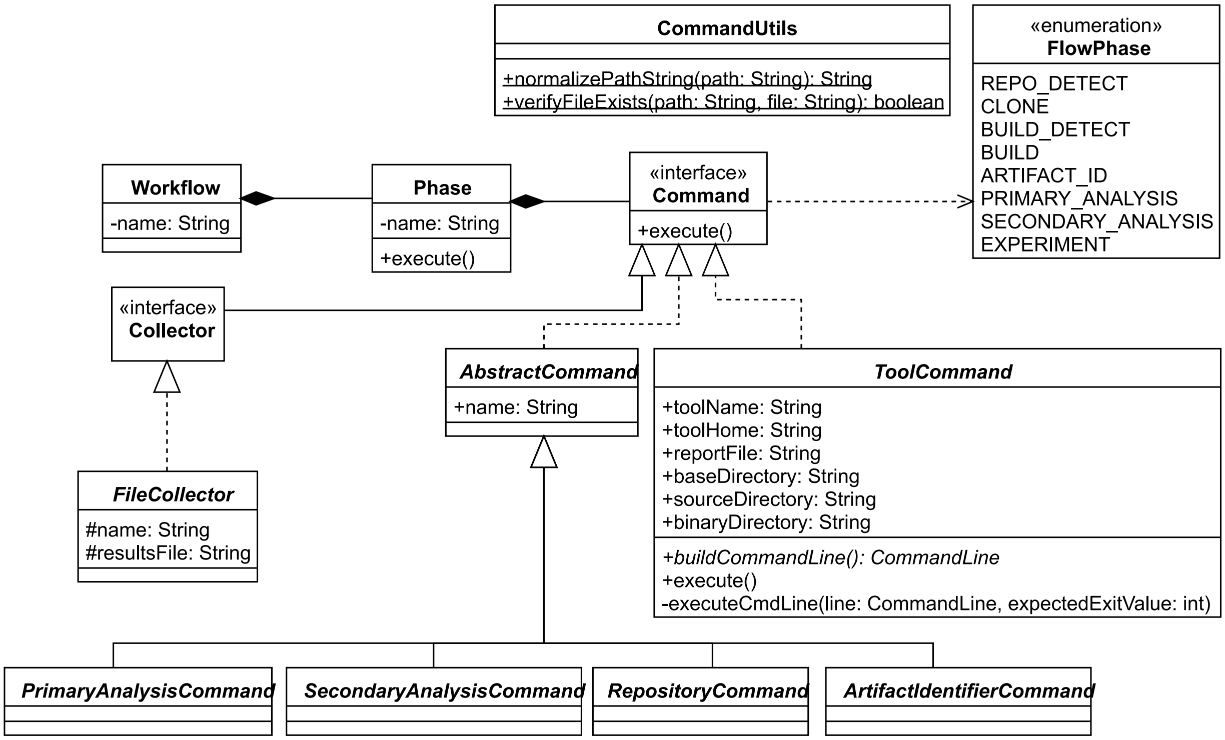


Figure 3.2: Workflow model

tool or other command which utilizes the results of PrimaryAnalysisCommands along with other data collected locally to evaluate derived measures. An example of such a command would be a command which executes a Software Quality or Technical Debt analysis. Next, *RepositoryCommand* is the base command for accessing the commands of a source code management library such as *git*, *mercurial*, or *subversion*. Next, *ArtifactIdentifierCommand* is the base class for those tools which identify file based artifacts within a project directory. Lastly, the *BuildCommand* is the base class for tools based on the libraries used to execute or analyze build and dependency management tools. Beyond the tools of the *AbstractCommand* hierarchy are the *Collectors*.

Collectors collect the output a tool produces. This output is preprocessed and then inserted into the data model. At this time, there is only one base class, *FileCollector*, which extracts information from tool output files. An example of this is the SpotBugsFileCollector. The SpotBugs tool executes across the binary class files of a project via a ToolCommand implementation. As SpotBugs executes, it writes the results of its analysis to a temporary XML file. This file contains a set of rule violations and the location in the program where the violation occurred. For each violation, the SpotBugsCollector generates a Finding. Each Finding links to a Reference representing the artifact where the violation occurred. The SpotBugsCollector stores these pairs in the data model. Once the data is stored, the collector deletes the intermediary results file.

3.3.1 Example Workflow

A workflow is simply a composition of several phases, which are then composed of several commands. These commands either read or write data to the data model. Thus, a typical Java workflow would look something like the one depicted in Figure 3.3. In the figure, the workflow is in the center surrounded by a rounded rectangle. Each phase is a rectangle containing the name of the phase. Each phase is separated by a thick green arrow indicating the flow of the phases and that separate phases must complete prior to transitioning to the next phase. Additionally, the commands of each phase interact with external entities such as files in a provided project and the database. We also note that a key representing the specific Java Tools included in the Standard Java Tooling are provided in the key surrounded by the green rounded rectangle. Data flow between phases and external entities are depicted as thin black arrows, with an arrowhead showing directionality of the data flow. Typically, such diagrams will also include numbers encircled in green which indicate the overall sequence of the process depicted. With that in mind we describe the provided example workflow.

This example defines a workflow which contains three phases: “Standard Java Tooling”, “Metrics, PMD, SpotBugs”, and “Quality/TD”. The “Standard Java Tooling” phase begins with the Java Artifact Identifier which identifies the source and binary file associated with the

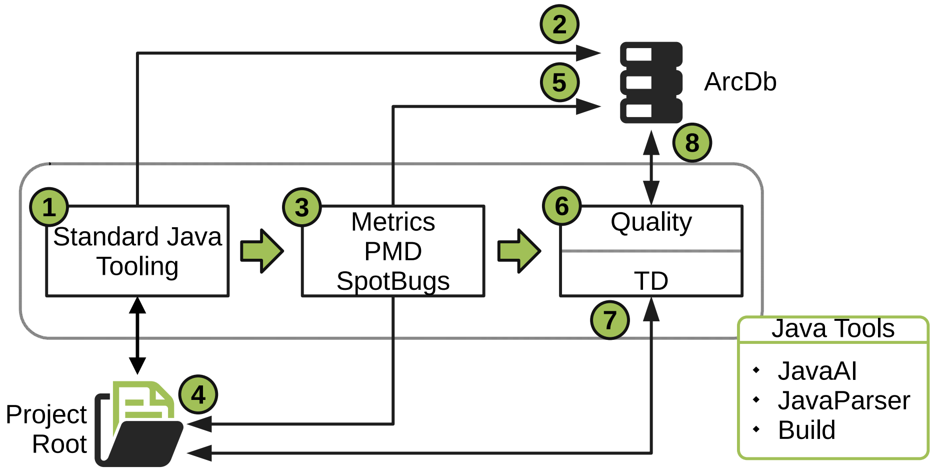


Figure 3.3: Example workflow for a Java Project.

provided project. As these items are identified it will update the appropriate ArcDb tables. Next it executes the JavaParser extracting the java components from the code. Finally it builds the project for later analysis.

The "Metrics, PMD, and SpotBugs" phase executes three primary analyses. Initially, a metrics analysis executes across the source code. Next, the PMD tool executes and its results collected. Finally, the SpotBugs tool executes, and its results collected. The results of this execution, stored in the database, can then act as input to the "Quality/TD" phase.

The "Quality/TD" phase works with prior collected data and executes the two derived analyses. The first command executes the Quamoco quality framework to evaluate the internal quality of the software system under analysis based on the aggregation of both measures and findings and storing these results as measures. The second command executes a technical debt analysis. This analysis aggregates findings and metrics values into a single index. Additionally, the technical debt analysis tools can produce an estimate of effort (to remediate the technical debt issues) based on the calculated index.

3.3.2 Data Model

The goal of this framework is the efficient and automated collection of data, in support of empirical studies, from several external and internal tools. The selected tools will differ depending on the languages in which the software studied has been implemented. Thus, we have developed an underlying data model that provides the basis and depth necessary for the collection of data from several disparate sources. The following describes this data model.

The data model has four main sections: the essential components relating to the metadata for a system under analysis, the components to contain pattern related data, the components to store information relating to static analysis results, finally, the components necessary to contain the information concerning project artifact data. The following subsubsections detail each of the sections of the model.

3.3.2.1 System Data.

Figure 3.4 defines the data model subsection concerning the System under analysis. This metamodel contains three primary data collection components. The first is the *System* class, which maintains the necessary information concerning the System under analysis. A system under analysis may have multiple versions. Each version analyzed becomes a member of the set of *Project*s. Each Project maintains the version of the System under analysis, the languages of source code contained in the project.

Additionally, the Project uses the *SCM* class to maintain information regarding source code management information. This information includes the repository URL, repository type, and the branch/tag to which the project belongs. The Project acts as the nexus point through which the remaining three sections interconnect.

3.3.2.2 Pattern Data.

The pattern section of the data model, depicted in Figure 3.5, contains six main components divided into two categories. The first category describes

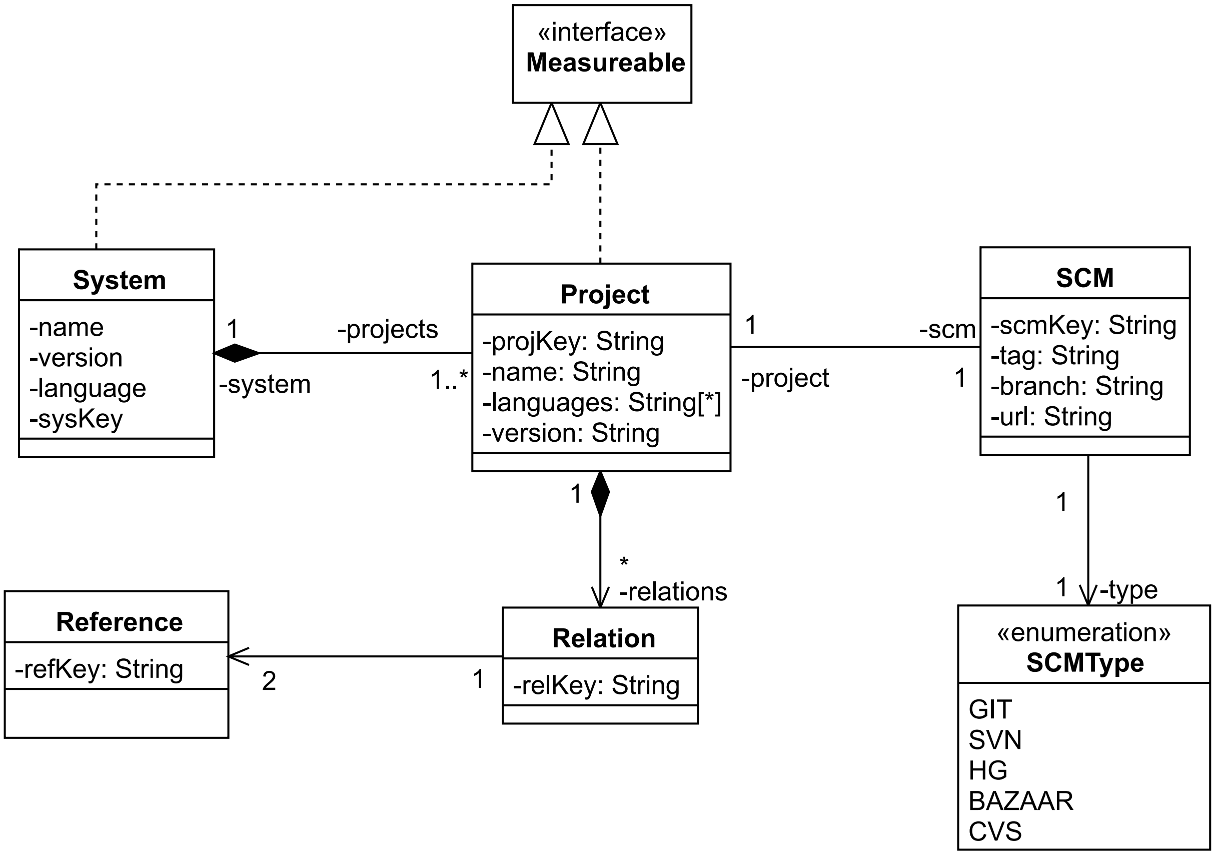


Figure 3.4: System data section of the data model.

design pattern definitions composed of three components. The first is the *Pattern* class, which contains information relevant to specific pattern types, i.e., the strategy pattern. The second component is the *Role*, which represents the individual structural components, such as classifiers and features, contained within a specific pattern type. The third component, the *PatternRepository* class, which provides a namespace for different pattern types and an overall pattern type container. An example used in the Arc Framework is the *GoF* repository containing the selected Gang of Four design patterns provided by the Pattern4 detection tool [263].

The second category, in the pattern data section, is related to pattern instances found within project artifacts and contains several primary components. The main component is

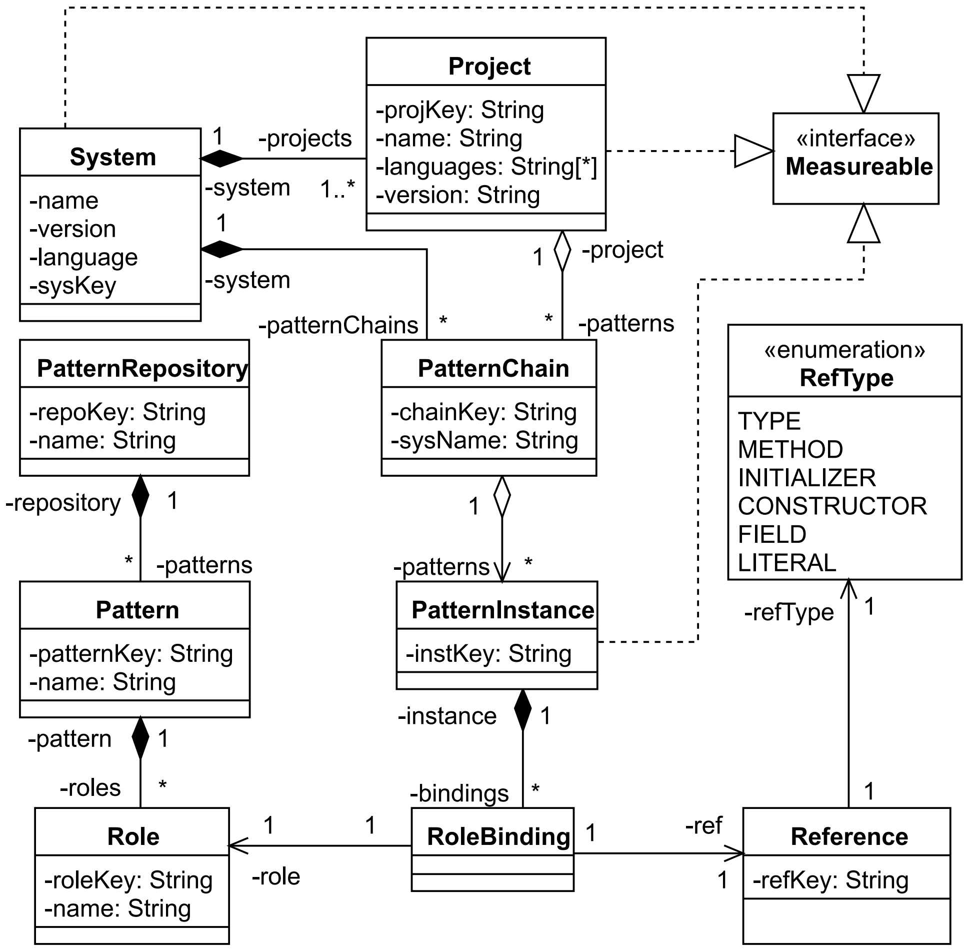


Figure 3.5: Pattern section of the data model.

the *PatternInstance* class, which represents an instance of a specific pattern type realized within the project under analysis. A project's artifacts, when needed outside of the main project hierarchy, can be accessed through the use of *Reference*s. References, in the case of PatternInstances, are linked to a given Pattern's Roles to construct *RoleBinding*s. A set of *RoleBinding*s then composes a PatternInstance. A PatternInstance, which has representation across multiple versions of a system, forms a *PatternChain*. These PatternChains are associated with the System and provide the ability to evaluate how a pattern instance changes rather than simply to analyze a pattern instance alone.

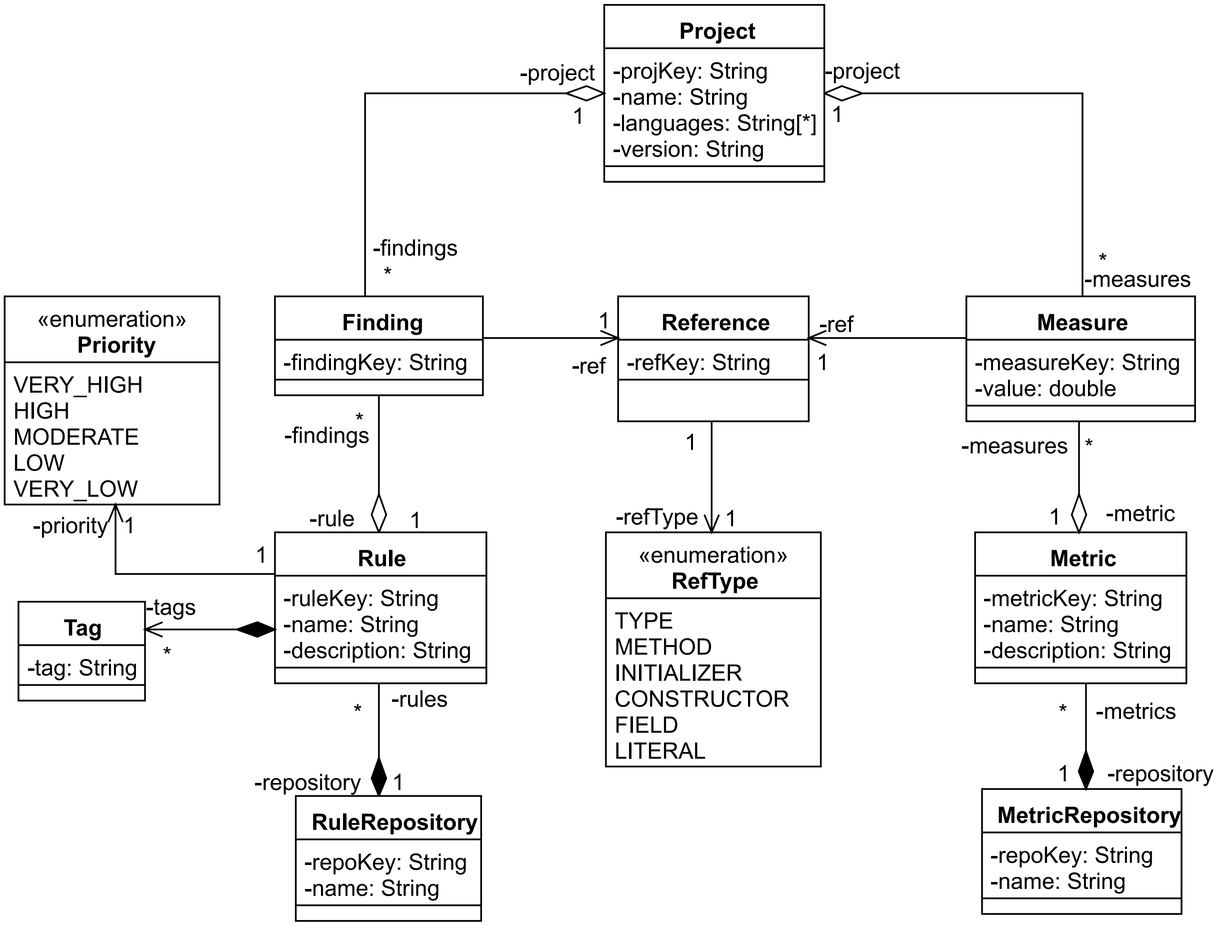


Figure 3.6: Static analysis data section of the data model.

3.3.2.3 Static Analysis Data This section of the data model, as depicted in Figure 3.6 contains data related to issues detected using static analysis tools and measures collected using metrics tools. This section is composed of the following two partitions.

The first partition concerns static analysis data and focuses on issue data in the form of a *Finding*. A Finding is a single data point relating an artifact within the System to a static analysis issue. *Rule*s represent Issues within the data model. In the model each Rule is a member of a *RuleRepository*, i.e. *pmd* or *spotbugs* for Java™ , and contains a *Priority* and a collection of *Tag*s. Each Tag provides metadata related to some more general category, i.e., security or technical debt, to which a rule belongs.

The second partition concerns measurement data. This data represented is the result of the application of a metrics tool to the System under study and is encapsulated in the *Measure* class. A Measure is a single data point, specific to a given project, evaluating some referenced artifact by a rule defined by a metric. *Metric*s, in the data model, contains related Measures and is contained within a *MetricRepository*, i.e., arc-metrics, but is not specific to any system or project.

3.3.2.4 Project Artifact Data. This is the largest but most easily understood section of the data model. This section, depicted in Figure 3.7, directly relates to logical and language constructs to which most computer scientists and software engineers are familiar. The major components of this section are the *Module*, *Namespace*, *File*, *Type* and its subclasses, and *Member* and its subclasses. Component definitions depend on the languages used within a project.

Structurally, for the Project artifact data, we have made some assumptions. First, we assume a Project has at least one Module. A Module, we assume, will contain at least one Namespace. In the case that neither of these two assumptions is true, a default Module or a default Namespace will be created. Namespaces, rather than modules, contain Files. Files contain Imports (i.e., Java import or C# using statements) and have an assigned type. A File's assigned type is selected from the *FileType* enumeration. These containment relationships simplify the overall structure to a tree form and allow a logical structure more closely representing project file structures. The remaining items from this section of the data model are related to source code artifacts.

These artifacts all derive from the base class *Component*. Components have a key, name, and *Accessibility* and a start and end line to define their location within a containing File. Furthermore, Components can be either a *Type* or a *Member*. The latter representing

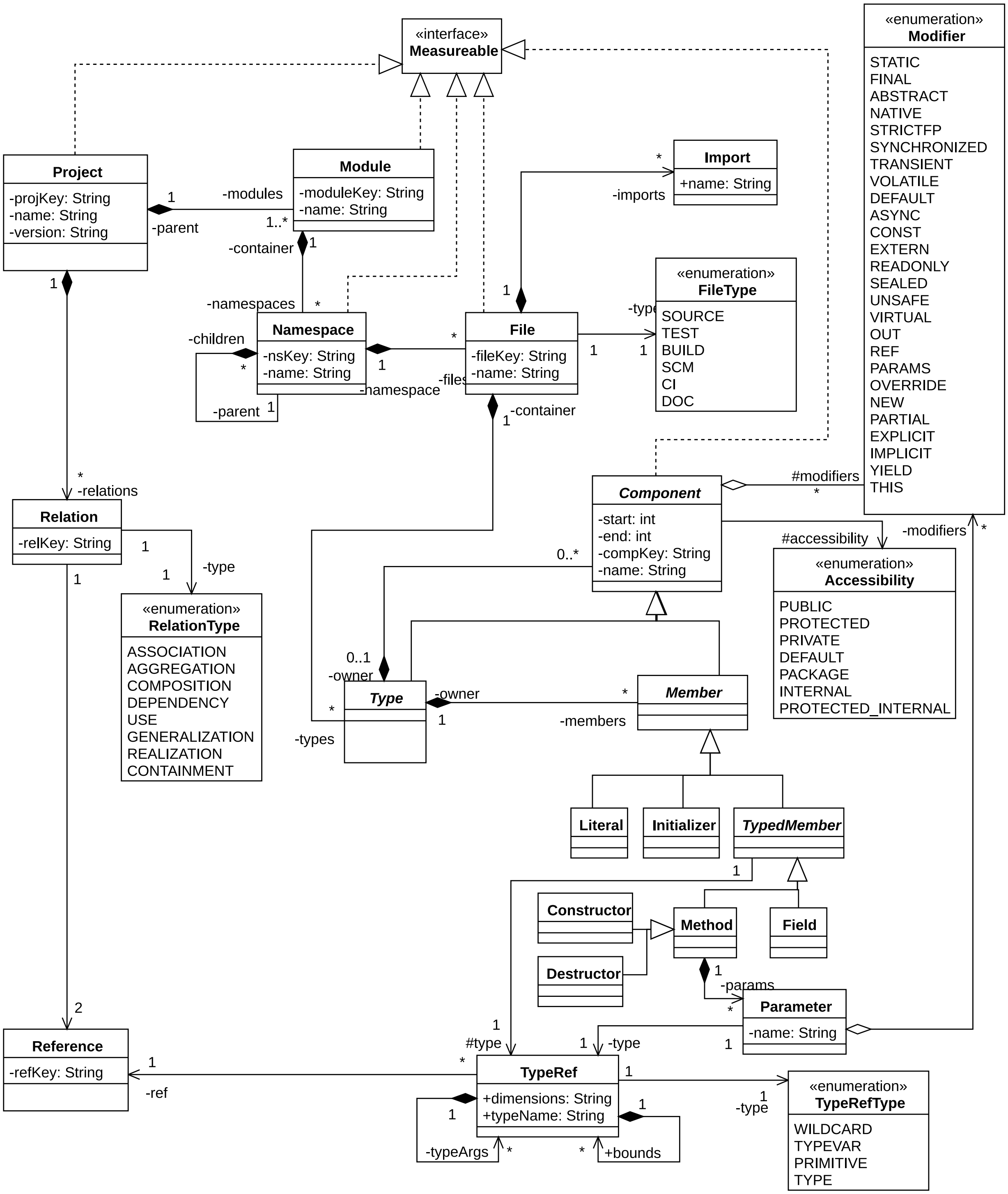


Figure 3.7: Project artifact data section of the data model.

artifacts contained within the former. The subtypes of both Member are representative of typical concepts found in most object-oriented languages and should be self-explanatory. Types maintain a field denoting whether it is an interface, class, enum, or unknown type. Unknowns are simply types referenced in the System under analysis, but which the System does not itself contain. These can be either language system provided types or library provided types.

3.4 Integration of Tools

The goal is to efficiently execute tools in the proper order. Toward this goal, each of the integrated tools is encapsulated, via Commands as depicted in Figure 3.2 and a ToolCommand using the Apache Commons Exec framework[3] to represent and execute external tools. Although tools may have specific constraints (such as requiring the project to be compiled before analysis), the framework does not explicitly enforce any such constraints. Rather, we expect that those who implement the workflows will have created phases meet these conditions. As an example, static analysis often requires a compiled software system for analysis. Understanding this issue, we have incorporated the capability to execute build tools such as Apache Maven[4] and the Gradle Build Tool[5], within sequential phases of a workflow. Besides, the previously mentioned build tools, we have implemented commands for several other tools for the Java™ ecosystem.

Currently we have implemented commands for the following tools: Maven, Gradle, SpotBugs[6] [121] , PMD[7], Git[8], the Pattern4 pattern detector [263], an implementation of the Quamoco quality measurement approach [270], an internal Metrics tool, and a Technical

[3] https://commons.apache.org/proper/commons-exec/
[4] https://maven.apache.org
[5] https://gradle.org
[6] https://spotbugs.github.io/
[7] https://pmd.github.io/
[8] https://git-scm.com/

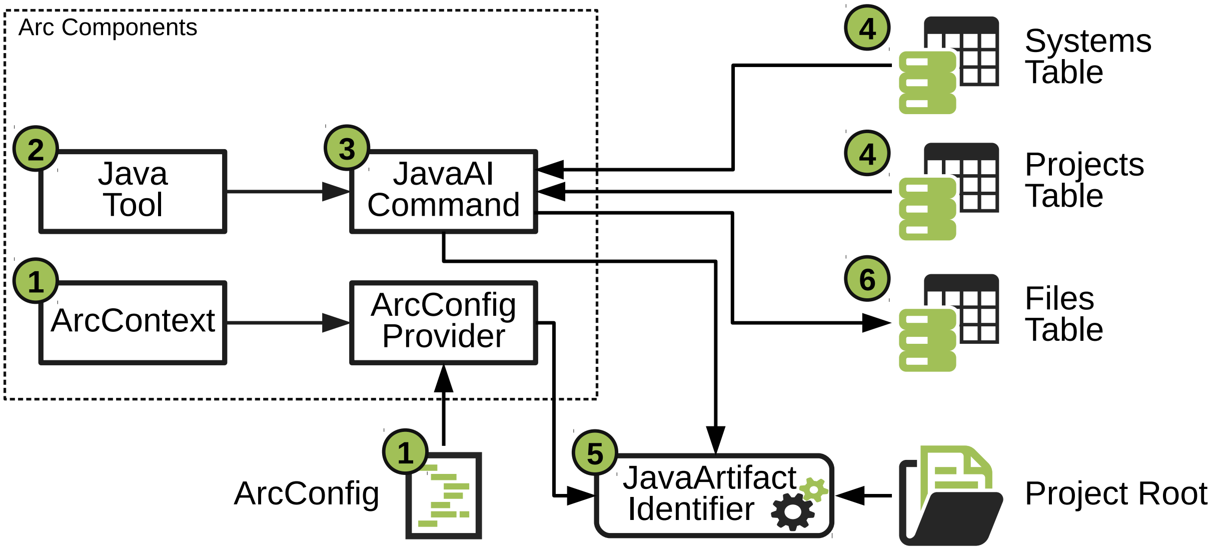


Figure 3.8: Integration of Java Artifact Identification with Arc.

Debt measurement tool. Finally, it should be noted that each tool and any required collectors are provided to Arc via an implementation of a *Tool*. Several of these tools and thier implementations and integration into the Arc framework are described in the following subsections.

3.4.1 Java™ Artifact Identification

The first tool, and arguably one of the most important, is the Java™ Artifact Identifier (JAI). JAI scans the project directory and identifies and adds those files pertinent to the analysis into the data model. Specifically, it identifies source files, binary files, and build files each needed during different stages of the analysis.

Figure 3.8 depicts the JAI tool integration execution flow. This flow, the numbers encircled in green, is as follows: 1.) The Arc Context loads the Arc configuration when the Arc system begins running. 2.) The JavaTool is loaded and provides the JavaAICommand. 3.) The JavaAICommand executes and requests 4.) the System and Project information. 5.) This provided information allows the JavaAICommand to initiate the JavaArtifactIdentifier,

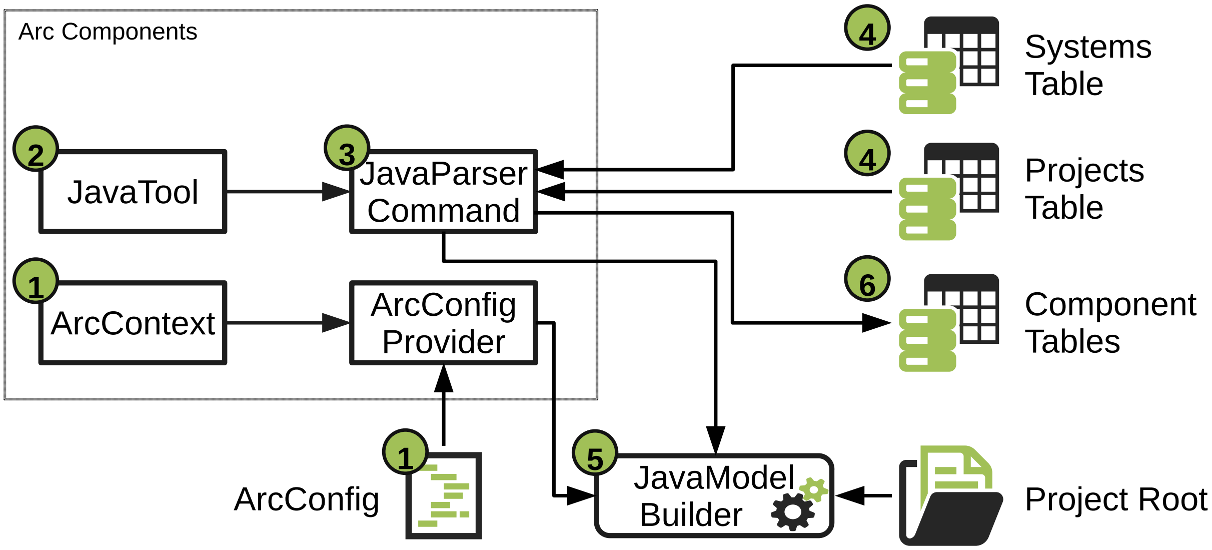


Figure 3.9: Integration of the Java Component Analysis with Arc.

which using the ArcConfig and information searches the Project directory for related Java files. 6.) The Java files found by the JavaArtifactIdentifier are then stored in the Files section of the Data Model.

3.4.2 Java Component Analysis

A key capability of the Arc Framework is the ability to provide a basis upon which measurements and issues can use in a component-level quality evaluation process. This process provided for the Java™ language, through the Java Component Analysis (Java) tool, uses an internal Java™ ANTLR[9] parser. This parser, in turn, provides a *ModelBuilder* for the Java™ language. These two components work together to populate the component level portions of the data model, providing the capability for metrics and other higher-level analyses.

The analyses provided by the Java tool and its Arc framework integration. Figure 3.9 depicts the Java tool integration and execution flow. This flow, the numbers encircled

[9] http://www.antlr.org

in green, is as follows: 1.) Initially, the Arc Context loads the Arc configuration when the Arc system begins running. 2.) Once the system is running, the JavaTool is loaded and provides the JavaParserCommand. 3.) The JavaParserCommand executes and requests 4.) the System and Project information. 5.) The JavaParserCommand executes the JavaModelBuilder, which, using the ArcConfig and information provided by JavaParserCommand, executes the ANTLR parser to extract type and member level components from the project source code files. 6.) The Components section of the data model stores this component-level information.

3.4.3 GitHub Search

In order to conduct empirical studies across open source software, we need a collection of open source projects to analyze. The projects we are limited to analyzing, due to current tooling, are Java™ based projects. Such a project can be found in several open-source repositories hosting sites. Several sites, over the years, have come and gone, but Github has become one of the most prominent sites and currently hosts a significant body of code. Thus, we selected to extract software currently hosted on their site and developed a tool to extract the necessary information.

Figure 3.10 depicts the GitHubSearch tool integration, and execution flow. This flow, the numbers encircled in green, is as follows: 1.) The Arc Context loads the Arc configuration when the Arc system begins running. 2.) Once running, the *GHSearchTool* initializes and provides to Arc the *GHSearchCommand*. 3.) The *GHSearchCommand* utilizes the GitHub REST API, provided via the GitHub API for Java[10]. 4.) This library, when parameterized with the appropriate credentials from the ArcConfig, provides the ability to access and search GitHub for suitable systems for analysis. 5.) The tool extracts System version information and repository information into Project and SCM classes, respectively.

[10] `http://github-api.kohsuke.org`

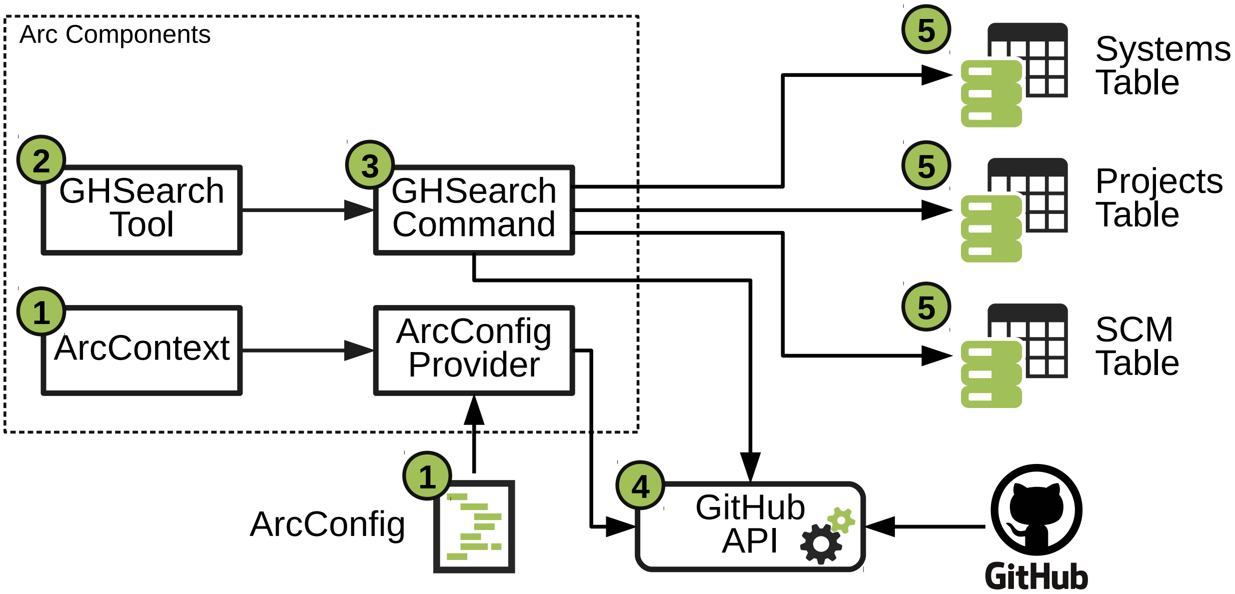


Figure 3.10: Integration of Github Search with Arc.

3.4.4 Git Execution

The previous tool provides the means to extract Software Projects from GitHub, along with metadata, which would allow these systems to be analyzed. One component of this metadata is the actual git URL to the project's repository. In order to use this, we must have a git client which would provide the means to download and extract the software project. Thus, we have implemented a tool that utilizes the JGit[11] Java library to clone the git repositories of each project to analyze. A part of this analysis the GHSearchTool extracts version information for each project from the list of tags in the repository. This information, once collected, provides the ability to conduct historical or longitudinal studies later. The Arc Frameworks facilitates these studies by allowing the use of the Git tool in workflows.

Figure 3.11 depicts the Git tool integration execution flow. The flow, the numbers encircled in green, is as follows: 1.) The Arc Context loads the Arc configuration when the Arc system begins running. 2.) Once running, the *GitTool* initializes and provides

[11] `https://www.eclipse.org/jgit/`

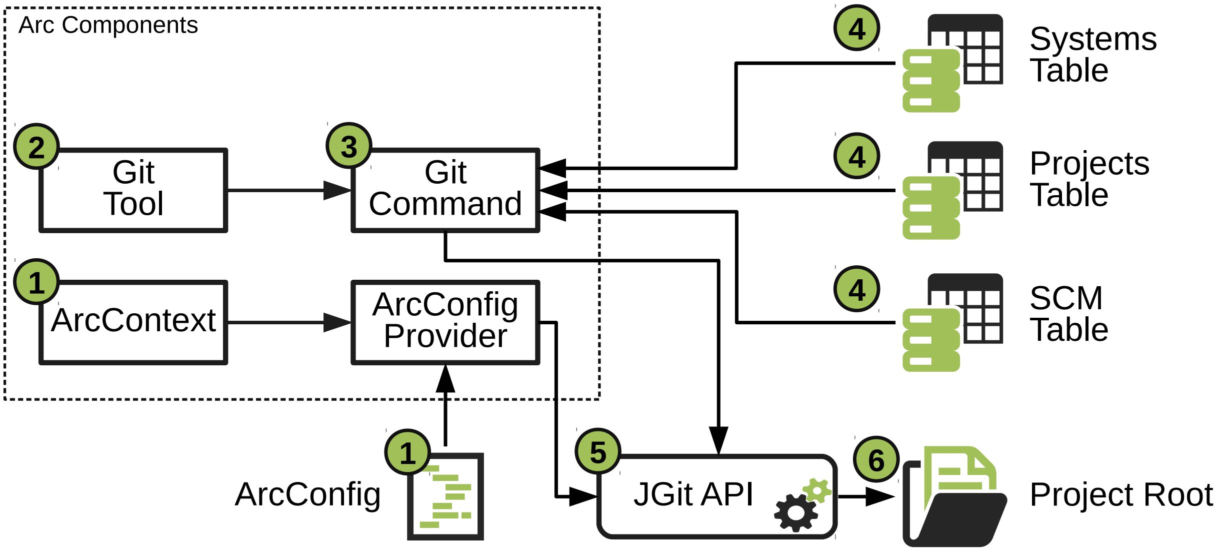


Figure 3.11: Integration of Git with Arc.

to Arc the *GitCommand*. 3.) The *GitCommand* utilizes the JGit[12] Java library to clone the Git repositories of each project to analyze. 4.) This library, when parameterized with the appropriate credentials from the ArcConfig, and the Project/System SCM information, allows Arc to clone the project properly. 5.) The project is then cloned by the GitCommand into an appropriate analysis directory, which then 6.) becomes the project root.

3.4.5 Static Analysis Tools

In software system analysis, the types of tools that detect quality-related issues fall into two main categories: static and dynamic analysis. Static analysis tools are those which evaluate the static structural elements of the software project. Precisely, they extract project artifacts such as source code, executable files, project directory structures, or build files. These artifacts provide the necessary information used to conduct analyses. These analyses include metrics measurement, identification of coding issues, identification of potential bugs, and identification of coding standard conformity issues, to name a few.

[12] `https://www.eclipse.org/jgit/`

Dynamic analysis, on the other hand, typically works by executing the software either via symbolic execution or through actual activation of the System. In the latter case, the tool monitors execution behavior in order to understand potential issues. Such an approach can evaluate test-case coverage, security issues, and parallel/concurrent programming issues (such as livelock and deadlock). Although such issues are essential, the availability of supporting tools is an issue; thus, for this research, we have limited ourselves to static analysis.

3.4.5.1 SpotBugs SpotBugs is a fork of the widely popular Java static analysis tool FindBugs developed at UMD College Park [121]. This tool uses a codification of known issues and good practices forming a set of rules. Rule violations detected by the tool are collected and output in XML format. The results provide input to the Arc framework data collectors extracted the data and entered it into the database. Although SpotBugs is a static analysis tool, it is dependent on the use of Java Reflection and thus requires the software project to be compiled into bytecode before analysis. The results of this analysis, along with the results of another tool, PMD, are used, in part, to evaluate Java project quality, as described in Chapter 5.

The SpotBugs tool integration is depicted in Figure 3.12. The Figure depicts the path of execution as numbers encircled in green. This path follows two possible routes both of which routes start by 1.) Initialization of the SpotBugsTool by the Arc system. The SpotBugsTool provides three components. The first is the *SpotBugsRuleProvider* which adds several SpotBugs related RuleRepositories to the data model. The second is the *SpotBugsCommand* which executes the SpotBugs external tool. Finally, the last item is the *SpotBugsCollector* which extracts issues from the resulting XML file. 2.a.) The SpotBugsRuleProvider, during system data model initialization, will construct the RuleRepositories and add Rule definitions to each repository. Each repository, once constructed, is (3.a.) added (along with its

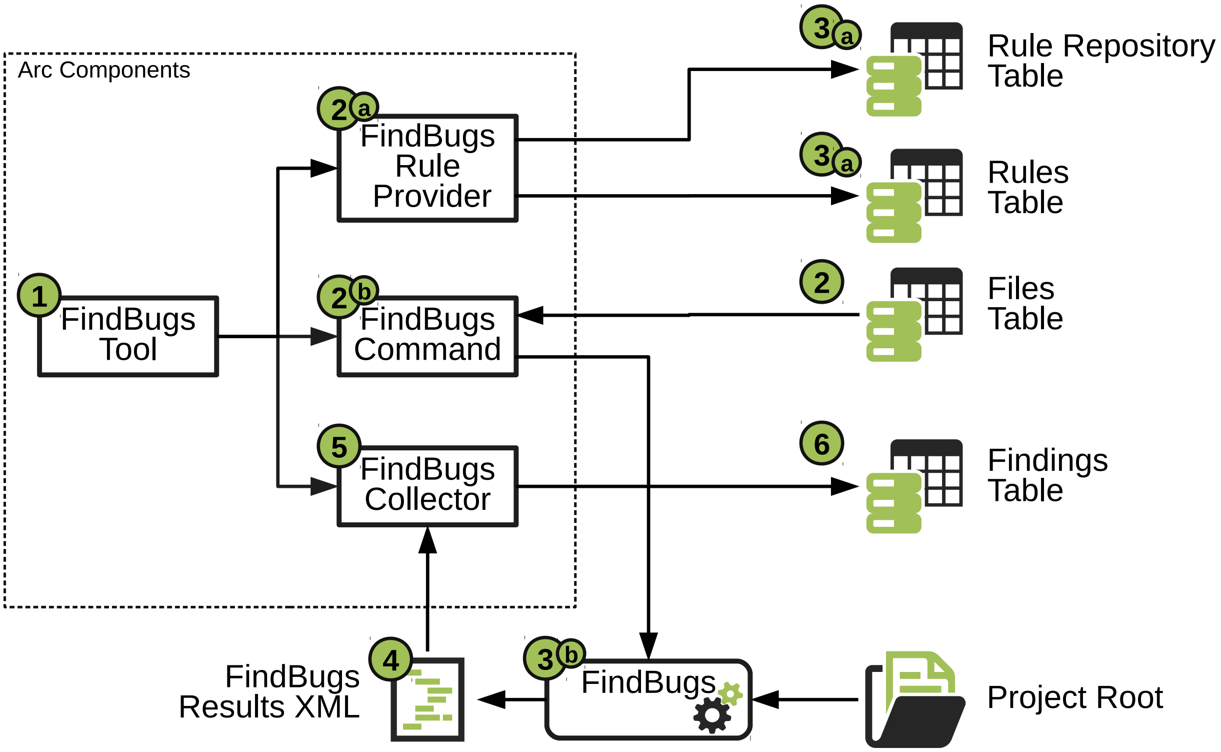


Figure 3.12: Integration of SpotBugs with Arc.

containing rules) to the data model. 2.b.) During the execution of the Arc system, the SpotBugsCommand extracts file information from the data model, and (3.b.) uses this to execute the SpotBugs external tool. 4.) SpotBugs produces a results file, which 5.) is then read by the SpotBugsCollector. The SpotBugsCollector extracts the found issues. 6.) These issues are used to construct and store Findings in the data model.

3.4.5.2 PMD PMD is another Java static analysis tool, similar to SpotBugs. Like SpotBugs, PMD uses a codification of known issues and good practices as a set of rules. The differences in rules, aside, PMD, unlike SpotBugs, does not require a compiled program for analysis. Instead, the analysis uses only source code to detect rule violations. The tool stores these violations in an XML file from which data is extracted and used in the evaluation of Java project quality. This process has been codified into the PMDToolCommand and integrated

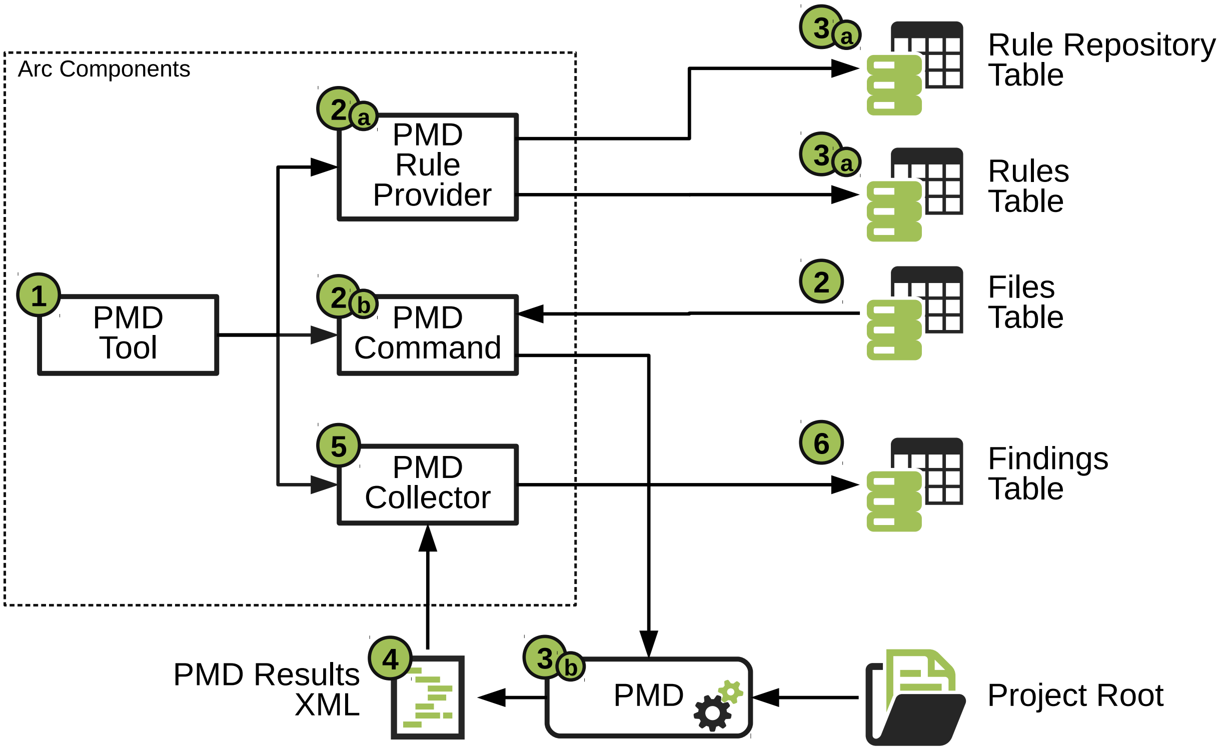


Figure 3.13: Integration of PMD with Arc.

into the Arc Framework.

Figure 3.13 depicts the PMD tool integration and its execution flow. This flow, indicated by the numbers encircled in green, follows two possible routes. Initially, both routes start with the 1.) initialization of the PMDTool by the Arc system. The PMDTool provides three components. The first is the *PMDRuleProvider*, which adds a PMD related RuleRepository to the data model. The second is the *PMDCommand*, which executes the PMD external tool. Finally, the *PMDCollector* extracts issues from the results of executing the PMD external tool. 2.a.) The PMDRuleProvider, during system data model initialization, will construct the RuleRepository, add Rule definitions to the repository, and then 3.a.) add all of these to the data model. 2.b.) During the execution of the Arc system, the PMDCommand extracts file information from the data model, and 3.b.) use this to execute the PMD external tool within the confines of the project directories. 4.) This execution results in the creation of a results XML file, which 5.) is then read by the PMDCollector. The PMDCollector then extracts the found issues 6.) to construct and store Findings in the data model.

3.4.5.3 Pattern4 Design Pattern Detector Tsantalis et al. [263] developed the Pattern4 design pattern detection tool. Chapter 4 details the use and integration of this tool and other algorithms developed (to collect and cleanse design pattern data) into the Arc framework.

3.4.5.4 Metrics Analysis Tool We have developed a tool to analyze software systems, during the primary analysis phase of the workflow, and collect several well-known metrics. Metrics analysis works by constructing measures for each component measured. The metrics tool then stores these values in the data model. Chapter 5 describes this process, the implemented metrics, and the integration into the Arc framework.

3.4.6 Build Tools

A build and dependency management system is typically employed to package a software system efficiently. Modern programming systems provide this capability through a myriad of tooling. Examples of such tools are Apache Maven[13] and Gradle[14] for Java™, Scala Build Tool[15] for Scala, and the combination of NuGet[16] and MS-Build[17] for .NET. We have focused on the two main Java build and dependency management tools: Apache Maven and the Gradle Build tool to facilitate this research. Their use and integration are as follows.

3.4.6.1 Maven The Apache Maven tool provides the capability to build nearly any type of Java application through the use of XML configuration files. The configuration provides

[13] https://maven.apache.org/

[14] https://gradle.org/

[15] https://www.scala-sbt.org/

[16] https://www.nuget.org/

[17] https://www.microsoft.com/en-us/build

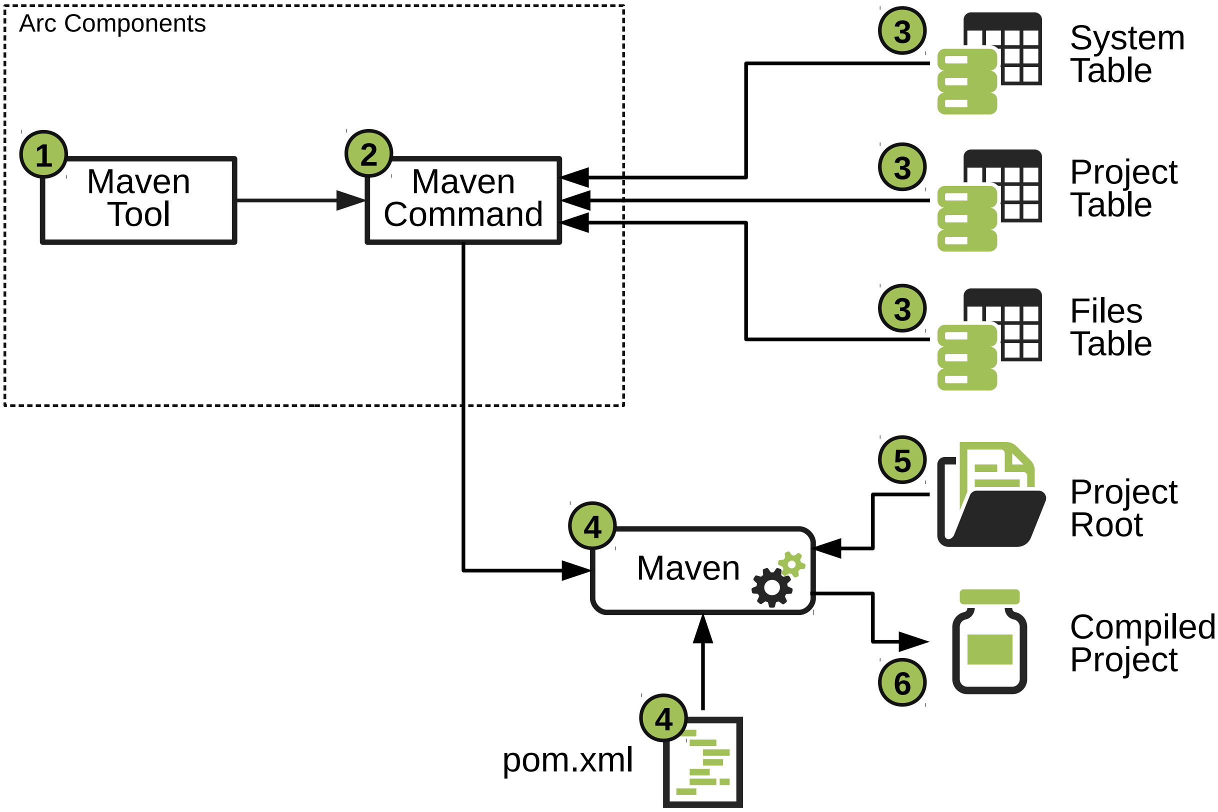


Figure 3.14: Integration of Apache Maven with Arc.

extensive capabilities for building, automated test execution, dependency management, and a default project directory structure exploitable during analysis.

Projects utilizing Apache Maven can easily be manually compiled from the command line using the Maven command-line tool. Tool execution logic has been encapsulated into an Arc ToolCommand for use within Arc workflows, as depicted in Figure 3.14. The process of the MavenTool execution is as depicted by the numbers encircled in green. The execution flow is as follows: 1.) The Arc system initializes the *MavenTool*, which then provides the *MavenCommand* to the Arc system. 2.) The Arc system then executes the MavenCommand, which 3.) extracts from the System, Project, and Files tables the necessary information needed to build the System correctly. 4.) At this point, the framework executes the Maven command-line tool (as controlled by the provided *pom.xml* build configuration file)

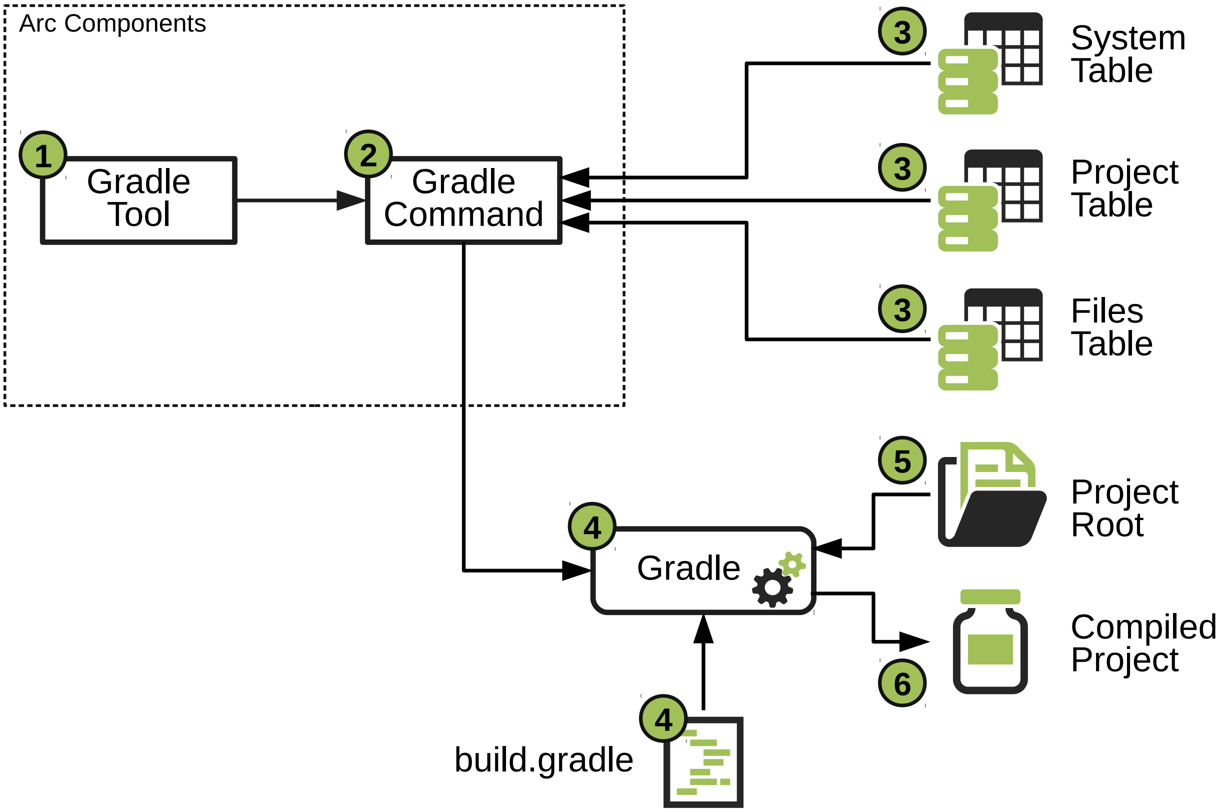


Figure 3.15: Integration of Gradle with Arc.

5.) utilizing the files in the project root. 6.) The output of this execution is the compiled project ready for analysis.

3.4.6.2 Gradle The Gradle tool provides a similarly highly configurable capability for building Java™ projects. Unlike Maven, Gradle uses a Groovy DSL to control the build process and manage dependencies. Similar to Maven, Gradle does provide the ability to utilize the existing Maven infrastructure for locating and utilizing dependencies.

Projects utilizing Gradle can easily be manually compiled from the command line using the Gradle command-line tool. The tool execution logic has been encapsulated into an Arc ToolCommand, similar to the Maven command, for use within Arc workflows, as depicted in Figure 3.15. The process of the GradleTool execution is as depicted by the numbers encircled in green. The execution flow is as follows: 1.) The Arc system initializes the *GradleTool*, which then provides the *GradleCommand* to the Arc system. 2.) The Arc system then executes the GradleCommand, which 3.) extracts from the System, Project, and Files tables the necessary information needed to build the System correctly. 4.) At this point, the framework executes the Gradle command-line tool (as controlled by the provided *build.gradle* build configuration file) 5.) utilizing the files in the project root. 6.) The output of this execution is the compiled project ready for analysis.

3.5 Conclusion

This chapter describes the basic concepts and components of the Arc Framework. This framework is the heart of the underlying method, which guides the experiments and studies found within this dissertation. In subsequent chapters, we build upon this framework and explore the remainder of the methods and techniques used to facilitate data collection and execution of our studies.

CHAPTER FOUR

COLLECTING DESIGN PATTERN DATA

Design patterns require neither unusual language features nor amazing programming tricks with which to astound your friends and managers.

–Gamma et al. [96]

4.1 Introduction

This chapter presents our approach to collecting design pattern data from a software system. One of the most difficult aspects of design pattern grime research is collecting the initial data. Unlike other forms of phenomena identification which work with artifacts of the language (i.e., static analysis issues, code smells, or antipatterns), it is difficult to detect grime directly. Grime detection subdivides into two steps. The first step involves the collection of the raw data from a design pattern detection tool, and the second step involves the cleansing of the data into a form suitable for identifying issues such as grime.

Similarly, experimentation with grime wherein we inject the grime instances is also tricky. This difficulty stems from the fact in order to inject grime that we must first have a design pattern instance. “Wild” design pattern instances (those found in open-source or industry software) do not distribute evenly (if at all) across pattern types or already contain an unknown level of grime. Thus, it is to our benefit to have the ability to generate design patterns instances to alleviate these issues.

This chapter grime detection and the two primary problems note prior. The first problem in grime detection is the need to detect design patterns. The detection tool we have selected, and its underlying operation are described in Section 4.2. This section further describes the subproblems related to pattern detection. These problems include evaluating the quality of the data and connecting pattern instances across versions of the software system. We alleviate these issues through the development of several techniques described in Section 4.2. The second problem of generating design pattern instances, for use in experimentation, is described in Section 4.3. This section further illustrates how this process integrates into the Arc Framework. Finally, this chapter is concluded in Section 4.4.

4.2 Design Pattern Detection

The first requirement to understanding and experimenting with design pattern grime is the ability to detect a software systems existing design pattern instances. Several methods have been proposed based analyzing the structural and behavioral aspects extracted from the software. Initially, techniques focused only on the structural aspects [18, 26, 29, 56, 57, 78, 119, 137, 156, 245, 252, 263, 279, 291] but as research progressed approaches began to focus on a combination of structural and behavioral analysis with or without semantic analysis [15, 33, 66, 67, 116, 117, 122, 240, 241, 259, 273, 274].

Upon reviewing the available methods and their proposed tools, we selected the Similarity Scoring Approach (SSA) as implemented by Tsantilis et al. [263] in their tool Pattern4. This tool was selected to fulfill the following requirements: (i) simple, quick execution separate from other tools such as IDEs, (ii) executable at the command line, (iii) capable of detecting a majority of the GoF design patterns, (iv) capable of analyzing Java software systems, and (v) ease of processing results. The Pattern4 tool fulfills these requirements, and next, we discuss its underlying algorithm.

The Pattern4 tool employs SSA, as shown in Algorithm 4.1, to detect design patterns given knowledge of their structure. Structural knowledge encodes a pattern definition through a set of four structural design matrices: (i) associations between class roles, (ii) generalizations between class roles, (iii) abstract class roles, and (iv) similar abstract method invocations between classes. This data is extracted by the tool using Java™ reflection and

Algorithm 4.1: Similarity Scoring Algorithm [263]

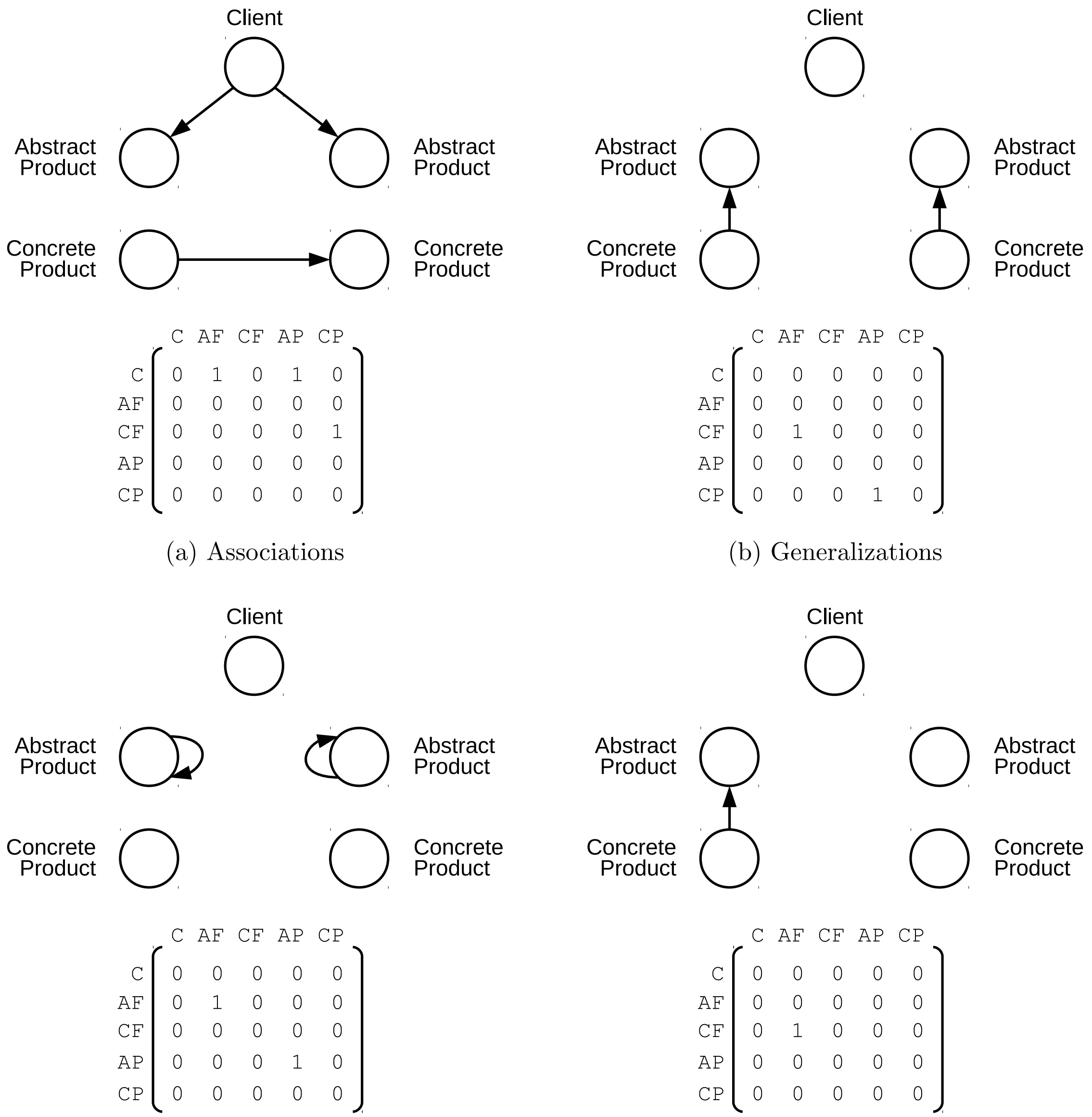
Require: A: an $n_A \times n_A$ matrix of a pattern specification graph G_A
Require: B: an $n_B \times n_B$ matrix of a system graph G_B
Ensure: $\mathcal{S}$: an $n_A \times n_B$ real valued matrix of normalized similarity scores, in range $[0, 1]$, where s_{ij} represents the similarity between vertex j in G_A and vertex i in G_B

```
function SimilarityScore($A$, $B$)
    $A \leftarrow$ adjMatrix($G_A$)
    $B \leftarrow$ adjMatrix($G_B$)
    $Z_0 \leftarrow 1$ // $n_B \times n_A$ matrix of all 1's
    repeat
        $Z_{k+1} \leftarrow \frac{BZ_KA^T+B^TZ_KA}{||BZ_KA^T+B^TZ_KA||_1}$ // for an even number of times
    until convergence
    $\mathcal{S} \leftarrow Z_K$
    return $\mathcal{S}$
end function
```

resulting in a graph of the software system.

This graph is processed using a set of heuristics (such as starting with generalization hierarchies) to locate candidate pattern instances. From these candidate pattern instances, it derives their structural matrix representations. The extracted matrices and pattern-specific matrices (as an example, see Figure 4.1) act as input to the SSA Algorithm. The algorithm compares these matrices and results in a similarity score for each pattern type. A pattern type that scores above a particular threshold, for a particular candidate instance, is assigned to that instance.

The actual pattern instances, once identified, are then encoded into an XML file composed of a single *project* tag, within which are a series of *pattern* tags (one for each of the 16 pattern types). Each pattern tag is composed of multiple *instance* tags. Each *instance* tag is composed of a series of *role* tags. *role* tags each map the named Role to an element within the software system (a class or a method). Although this process works well and is capable of identifying design pattern instances, it is not without its shortcomings. In the following section, we address these shortcomings and discuss our approaches to alleviating



$$\begin{array}{c|ccccc} & \texttt{C} & \texttt{AF} & \texttt{CF} & \texttt{AP} & \texttt{CP} \\ \hline \texttt{C} & 0 & 1 & 0 & 1 & 0 \\ \texttt{AF} & 0 & 0 & 0 & 0 & 0 \\ \texttt{CF} & 0 & 0 & 0 & 0 & 1 \\ \texttt{AP} & 0 & 0 & 0 & 0 & 0 \\ \texttt{CP} & 0 & 0 & 0 & 0 & 0 \end{array}$$

(a) Associations

$$\begin{array}{c|ccccc} & \texttt{C} & \texttt{AF} & \texttt{CF} & \texttt{AP} & \texttt{CP} \\ \hline \texttt{C} & 0 & 0 & 0 & 0 & 0 \\ \texttt{AF} & 0 & 0 & 0 & 0 & 0 \\ \texttt{CF} & 0 & 1 & 0 & 0 & 0 \\ \texttt{AP} & 0 & 0 & 0 & 0 & 0 \\ \texttt{CP} & 0 & 0 & 0 & 1 & 0 \end{array}$$

(b) Generalizations

$$\begin{array}{c|ccccc} & \texttt{C} & \texttt{AF} & \texttt{CF} & \texttt{AP} & \texttt{CP} \\ \hline \texttt{C} & 0 & 0 & 0 & 0 & 0 \\ \texttt{AF} & 0 & 1 & 0 & 0 & 0 \\ \texttt{CF} & 0 & 0 & 0 & 0 & 0 \\ \texttt{AP} & 0 & 0 & 0 & 1 & 0 \\ \texttt{CP} & 0 & 0 & 0 & 0 & 0 \end{array}$$

(c) Abstract Classes

$$\begin{array}{c|ccccc} & \texttt{C} & \texttt{AF} & \texttt{CF} & \texttt{AP} & \texttt{CP} \\ \hline \texttt{C} & 0 & 0 & 0 & 0 & 0 \\ \texttt{AF} & 0 & 0 & 0 & 0 & 0 \\ \texttt{CF} & 0 & 1 & 0 & 0 & 0 \\ \texttt{AP} & 0 & 0 & 0 & 0 & 0 \\ \texttt{CP} & 0 & 0 & 0 & 0 & 0 \end{array}$$

(d) Similar Abstract Method Invocations

Figure 4.1: Example matrix breakdown of the Abstract Factory Pattern. The circles (nodes) in each graph represent class roles, and links represent the presence of that type of connection (Association, Generalization, Abstraction, Method Invocation).

them.

4.2.1 Data Cleansing

We have identified three major shortcomings through our use of the Pattern4 tool. The tool's results produced appear, anecdotally, to consider certain pattern instances as more than one instance. That is, in cases where generalization hierarchies form a main body of the pattern, the tool appears to identify multiple pattern instances based on the subtypes, rather than finding a single instance based on the root of the hierarchy (a similar issue was noted by Reimanis and Izurieta [220] and Feitosa et al [77]). The second shortcoming of this tool is the lack of identification of several *optional roles* (c.f. 4.2.2) along with abstract roles providing generalization hierarchies in the pattern specification. The final issue is the inability of the tool to report on patterns implemented through language idioms (i.e., an observer pattern implemented in Java™ using the Observable interface). Although we cannot alleviate the third shortcoming, we alleviate the first two through an algorithmic approach based on RBML.

Our approach implements RBML using a YAML specification. These specifications provide the framework, along with mappings between RBML roles and Pattern4 instance roles, the capability to validate detected pattern instances. These instances detected using the Pattern4 tool and collected from a result XML file. The resulting roles map from their Pattern4 names to the RBML names and connect to the identified data model components extracted from the XML file. This mapping provides the necessary information to construct pattern instances in the Arc data model. These initial PatternInstances are used as input to the Pattern Coalescing Algorithm, as shown in Algorithm 4.2.

The algorithm (similar to the coalescence approach of Reimanis and Izurieta [220]) uses the initial PatternInstances to identify instances sharing a common subset of roles and pattern type that can coalesce. Coalescence uses a two-pass approach, as follows. First, the set of candidate pattern instances uses pattern type to form partitions. Once partitioned, the algorithm uses contained generalization hierarchies to control expansion (at line 5 of the

Algorithm 4.2: Pattern Instance Coalescing Algorithm

Require: $\mathcal{R}$: Result set of Pattern Instances from Pattern Detection

```
procedure COALESCE($\mathcal{R}$)
    $map \leftarrow$ PARTITION($\mathcal{R}$)        ▷ pattern instances keyed by pattern type
    for all $(k, \mathcal{P}) \in map$ do
        for all $p \in \mathcal{P}$ do
            $p \leftarrow$ EXPAND(p)        ▷ Expand each instance
        end for
    end for
    for all $(k, \mathcal{P}) \in map$ do
        $\mathcal{P} \leftarrow \mathcal{P} \setminus$ COMPAREANDCOMBINE($\mathcal{P}$)
    end for
    $\mathcal{R} \leftarrow map.values$
    return $\mathcal{R}$
end procedure
```

algorithm). Expansion occurs in both directions (up and down the generalization and across associations). The pattern instance adds each class, which is not already a member (if the class fulfills a role in the pattern, as defined by the RBML). Next, the process of combining pattern instances begins.

For each pattern type and the instances assigned to that pattern type, we call the method *compareAndCombine(. . .)*. This method takes a set of pattern instances as input and compares each pair, merging those which share a common set of classes. Finally, the algorithm updates both *map* and the set $\mathcal{R}$ to reflect the current set of complete instances, returning $\mathcal{R}$.

4.2.2 Pattern Chains

A part of the research herein is to evaluate the evolution of design patterns. The evolution of a single design pattern instance, observed at discrete instants of time (i.e., releases), forms a sequence, which we call a *pattern chain*. A pattern chain provides a useful concept for evaluating the evolution of a pattern instance but adds a layer of complexity in the process. This complexity stems from the requirement to both identify patterns across observations, a process we call *pattern tracking.*

Tracking pattern instances across observations require that we define a *pattern instance identity.* An initial definition, derived from the specification of the parent pattern type, can be simply stated as a set of Classifier Roles, Feature Roles (fulfilling the field and methods within the classifiers), and Relationship Roles. A pattern instance is then a mapping from the entities in the RBML roles to the actual artifacts in the software system, which is called a *binding.*

For a binding to work as the definition of an instance's identity, we must consider role types. A role is either optional or mandatory, where a *mandatory role* plays a role contributing to a design pattern specification with a multiplicity having a lower bound greater than or equal to 1. Having a lower bound multiplicity of 0, on the other hand, indicates an *optional role.* A *role binding* refers to the pairing of a role with a component in the system. Putting this together, we define a *Pattern Instance Identity* as *the collection of role bindings for all mandatory roles in the pattern specification.*

This definition works well for a single version of a software system. It provides a distinction between both pattern instances of the same type and pattern instances of the same type. Such a distinction is necessary, but also insufficient to distinguish among the same pattern instance across multiple versions of a software system. Thus we need to refine the notion of identity further.

Across multiple versions of a system the core identity component, the set of role bindings, may change in size as the pattern evolves. This change in size presents a significant problem in tracking the pattern. To address this we define the *pattern chaining operator*, $\prec_p$, such that for any two pattern instances P_1 and P_2, $P_1 \prec_p P_2$ indicates that P_1 precedes P_2 in the chain and that P_1 and P_2 are the same pattern instance separated by versions of the software. The semantics of this operator is defined as follows: $P_1 \prec_p P_2 \equiv [(B(P_1) \subseteq B(P_2)) \wedge (B(P_1)_m \setminus B(P_2)_m = \varnothing)] \vee [(B(P_2) \subseteq B(P_1)) \wedge (B(P_2)_m \setminus B(P_1)_m = \varnothing)]$. Where, $B(X)$ is the set of role bindings for pattern instance X and $B(X)_m$ is the set of role bindings for pattern instance X concerning only those roles which meet the previous definition of mandatory roles. This operator is designed to encompass the logic of identifying a pattern instance as it either grows or shrinks throughout its evolution.

Pattern evolution creates 5 cases to address. (i) The pattern expands by adding new role bindings. (ii) The pattern shrinks by removing role bindings. (iii) The pattern remains unchanged. (iv) The pattern no longer exists, or (v) the pattern reappears. In the first case (expansion), a chain links two versions of a pattern instance, P_1 and P_2. This link occurs iff either the set of role bindings of P_1, $B(P_1)$, is a subset of the role bindings of P_2 and the set difference between the set of role bindings for mandatory roles of P_1 and the set of role bindings for mandatory roles of P_2 is the empty set. Thus, for expansion, the prior version of the pattern instance should contain the same role bindings as the following version and thus will be a subset of the following version. Furthermore, the difference in mandatory roles (those comprising the identity of an individual version), when using symmetric set difference focusing on the following version, $P2$, the difference should be the empty set. A similar line of reasoning follows for the second case, but we swap the versions as we expect the pattern to be contracting, and thus role bindings have been removed. These two cases are indicated in the pattern chaining operator by the clauses separated by the primary or operator. Additionally, the first two cases capture the third case. The fourth and fifth cases require the tracking approach to look at all chains (including those which have ended) as possible chain candidates.

Algorithm 4.3 defines the process for constructing pattern chains. This algorithm takes as input the Project under analysis, which is associated with a particular version of a software system. As depicted in Chapter 3 Figure 3.5 each project is contained with a System, and

Algorithm 4.3: Pattern Instance Chaining Algorithm

Require: $\mathcal{P}$: Current project under analysis

```
 1: procedure ChainDetection($\mathcal{P}$)
 2:     $s \leftarrow \mathcal{P}.system$
 3:     $chains \leftarrow s.$patternChains
 4:     if $chains = \varnothing$ then
 5:         createChains($\mathcal{P}.patterns$)
 6:     else
 7:         for all $p \in \mathcal{P}.patterns$ do
 8:             $chain \leftarrow \varnothing$
 9:             for all $c \in chains$ do
10:                 if $c.matches(p)$ then
11:                     $chain \leftarrow c$
12:                     break
13:                 end if
14:             end for
15:             if $chain \neq \varnothing$ then
16:                 $chain \leftarrow p$
17:             else
18:                 $chains \leftarrow$ createChain($p$)
19:             end if
20:         end for
21:     end if
22: end procedure
```

that System also contains a collection of PatternChain(s). Each PatternChain is composed of a set of PatternInstances forming the chain across each of the Projects. Thus, we can extract all the needed information from the Project alone. Initially, the provided Project is used to extract the parent System. The System provides its set of PatternChains, to the variable *chains*. If the set is empty, then this is the first version (or Project within the System) to be processed. At this point, the algorithm constructs a new chain per individual unknown pattern instance, as detailed in Lines 6 through 21.

In the case of existing chains, for each pattern instance in the Project, the algorithm determines if it is a member of an existing chain or the start of a new chain. Thus, for each chain in the set of known pattern chains, the algorithm determines if the current pattern instance, p, matches the chain. The *matches(...)* operation is an implementation of the pattern chaining operator which tests whether $p_{-1} \prec_p p_0$, where p_{-1} is the previous version's pattern instance in the chain and the current version, p_0. In the case of a match, the algorithm sets the variable *chain* to the match, and the algorithm stops searching. Otherwise, the algorithm will continue searching for a match. If there is a match, the algorithm adds the current pattern instance, p, to that chain. Otherwise, the pattern instance starts a new chain.

In the following subsection, we discuss how the pattern detection tool, pattern coalescing algorithm, and pattern chaining algorithm work together. We also discuss the combination of these three components via their integration into the Arc Framework.

4.2.3 Integration into Arc

The goal of the described tools is to provide the capability for pattern data collection. Pattern data collection is a small part of the larger data collection framework, including related software quality measurements, metrics measurements, and issues affecting the software product, as described in Chapter 3. Thus, to further complete the Arc framework, pattern data collection is integrated as follows.

Figure 4.2 depicts the Pattern4 tool integration, and execution flow. The flow, the numbers encircled in green, is as follows: 1.) The system initializes the *DPDTool* which then initializes the components: *Pattern4Command* and *Pattern4Collector*. 2.) The *Pattern4Command* then executes the design pattern detection process embodied in *Pattern4Tool*. 3.) The *Pattern4Tool* executes the command line Pattern4 tool across the compiled version of the Project, which generates the Results XML file. 4.) The *Pattern4Collector* then reads these results. 5.) The *Pattern4Collector* utilizes the data from the results XML file and 6.) the Component Table(s) and Project Table of the

ArcDb to generate candidate PatternInstance(s). 7.) The Pattern4Collector is determines if InstanceCoalesce is to be used (8a.) or if the candidate PatternInstances are to be stored directly (8b.). In the latter case, 8b.) The instances are directly stored into the Arc data model, and the process continues at Step 10. Otherwise, 8a.) These candidate PatternInstances are then passed to the *Pattern4Command*. 9.) The *Pattern4Command* executes the *InstanceCoalesce* algorithm (see Algorithm 4.2) to reduce the set of candidate instances to the set of actual pattern instances. 10.) The final set of PatternInstances is entered into the Arc data model and passed through the *PatternChaining* algorithm (see Algorithm 4.3) to produce new pattern chains or expand existing chains. 11.) Finally, the created or expanded chains are then entered or updated in the Arc data model.

4.2.4 Summary

This section detailed the pattern data collection approach. This approach includes the ability to cleanse the data, ensuring that identified pattern instances are complete and accurate. This set of pattern instances forms pattern chains across software versions to provide the capability to observe pattern evolution. Each component of this approach is integrated into the Arc framework to provide the ability to integrate pattern data with the other forms of information collected. In the following section, we consider the issue of collecting a large enough sample of design pattern instances necessary to conduct experiments with design patterns, which leads to an approach for design pattern instance generation.

4.3 Design Pattern Generation

Design pattern generation, or a curated collection of verified and validated pattern instances, is required as the raw data necessary for design pattern experimentation. Experiments evaluating the effects of phenomena, such as design pattern grime, on pattern instance measures (such as quality or technical debt) require a large sample of each type of

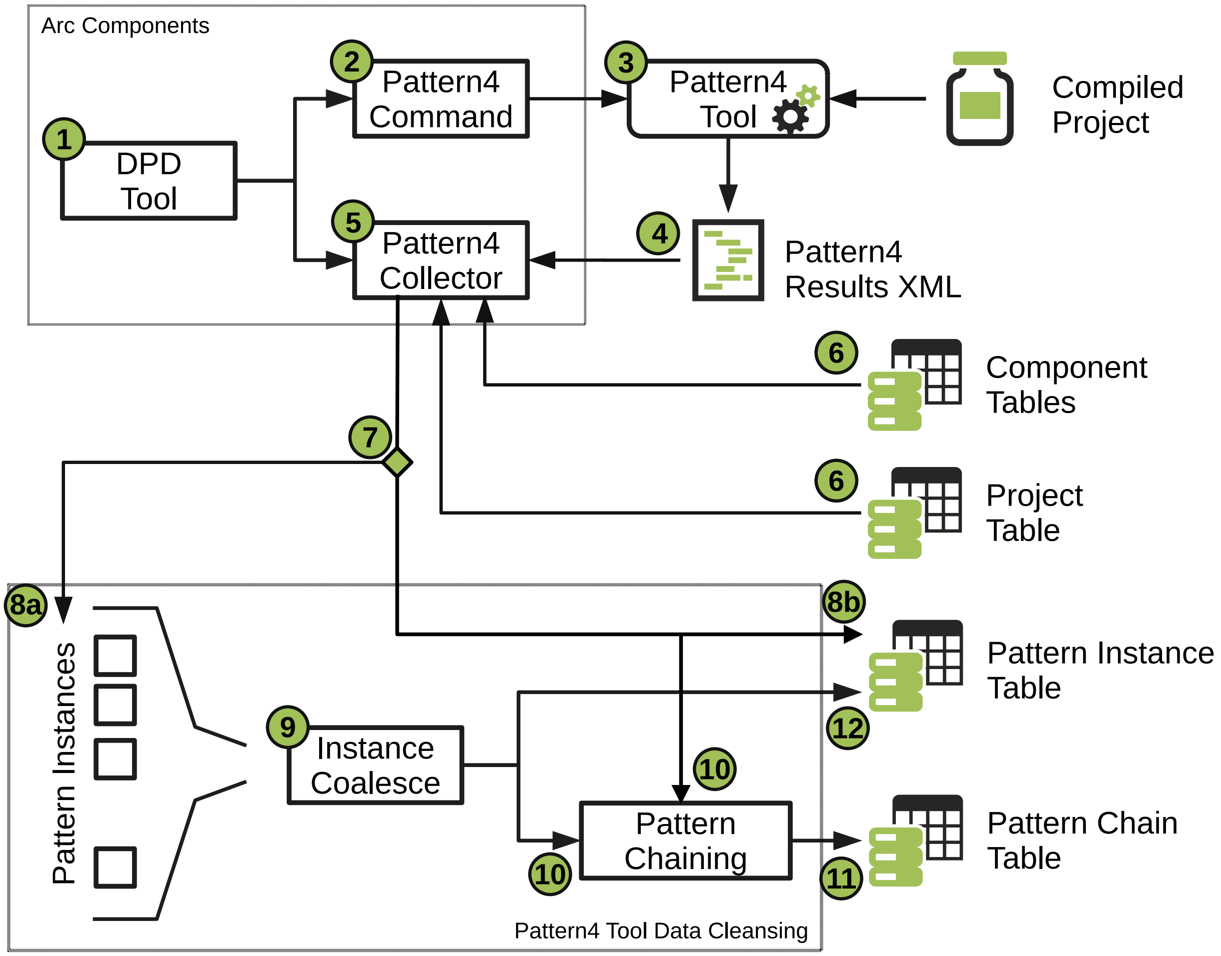


Figure 4.2: Integrating Pattern4 and Design Pattern Data cleansing into Arc.

pattern. Such a sample, in the naive approach, would require data collection across a vast number of software projects to identify the number of instances needed. One approach would be to utilize existing curated design pattern data sets such as the Percerons repository [9]. Unfortunately, even this repository is limited in the number of collected pattern instances per pattern type and is not necessarily free of error due to manual curation. To overcome these limitations, we have developed a method using RBML specifications to generate design pattern instances that integrates into the Arc framework. This integration, the supporting architecture, and the method itself described in the following subsections.

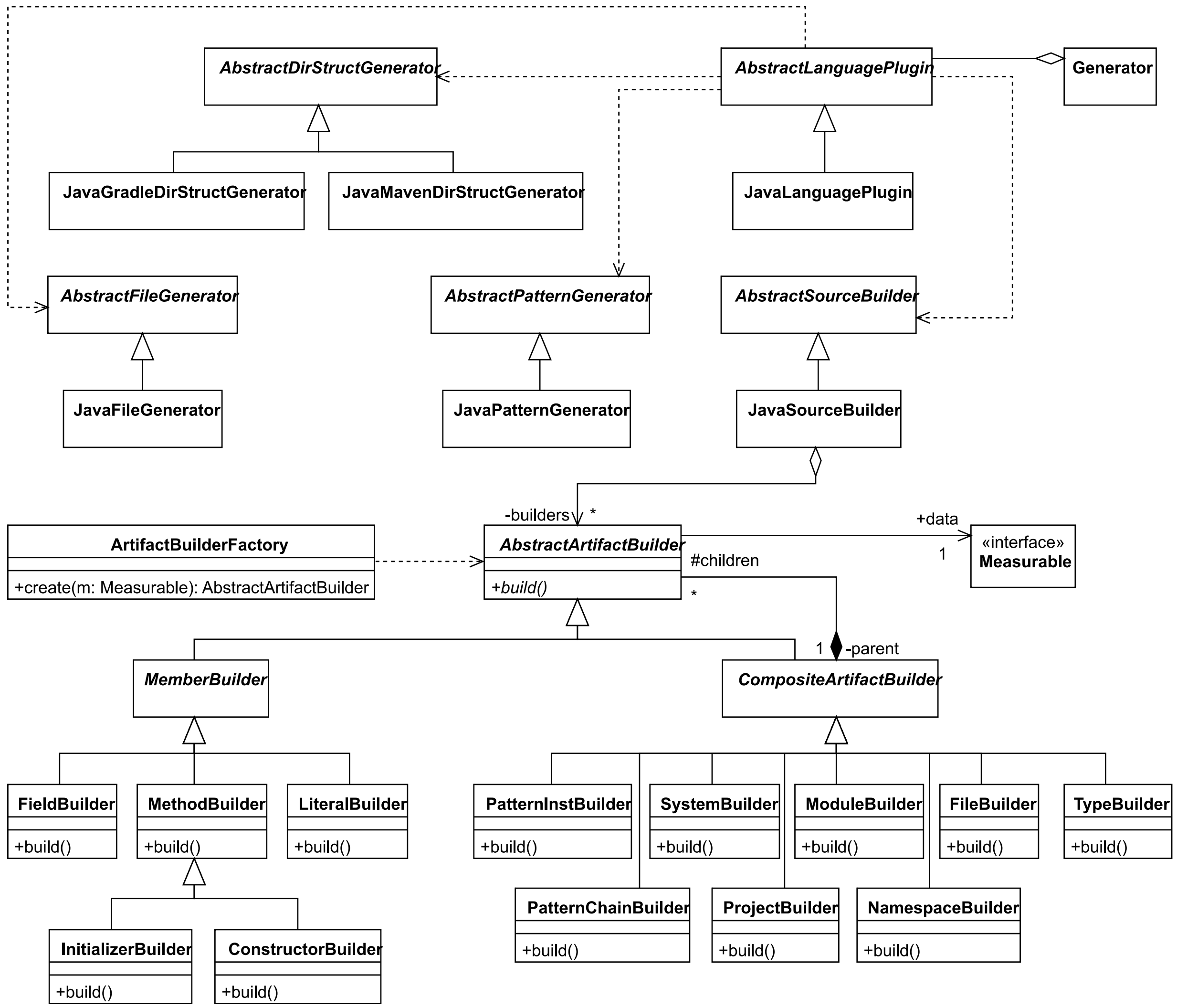


Figure 4.3: Pattern Generation class diagram.

4.3.1 Design Pattern Generation Architecture and Method

Figure 4.3 depicts the design pattern generation architecture. The architecture's controlling component is the *Generator* class. This class contains a set of *AbstractLanguagePlugin* classes. Each class implementing this abstract class, i.e. *JavaPlugin*, provide four components. The first component is the *AbstractDirStructGenerator*, which generates the project directory structure and its containing files. The generation of these files requires language-specific functionality. In the case of Java™ we have two separate implementations, one for Maven and one for Gradle. Both implementations use a language-specific implementation of *AbstractFileGenerator*, the second component, to generate basic project files (such as a “README.md” file). After file generation, the *AbstractPatternGenerator*, the third component, provides the logic for design pattern generation and is used to generate the data model constructs.

These generated constructs are then placed in the ArcDb and are used by the final component, the *AbstractSourceBuilder*. The *AbstractSourceBuilder* executes a collection of *AbstractArtifactBuilder*(s) to realize the physical file and directory contents as represented in the data model. *AbstractArtifactBuilder* is specialized into subtypes corresponding to the data model source artifact components. The *AbstractSourceBuilder* utilizes the *ArtifactBuilderFactory* to construct each of the *AbstractArtifactBuilder*(s). Currently this process is only setup for the Java™ language, but it can easily be extended to other languages. The overall generation is described by Algorithm 4.4.

This algorithm requires both the RBML pattern specification and the Pattern Cues for the pattern type to be constructed. RBML, as described in Chapter 2 is used to specify design patterns through an extension to the UML combined with OCL. Pattern Cues, specified by the Pattern Generation Cue Language (PGCL), as defined in Section 4.3.2, provide additional direction to the pattern generation process concerning language specific implementation details. A DSL provides the capability used to express these specifications as external Groovy scripts. These scripts are loaded at runtime by the generation system and provided to the algorithm. The algorithm outputs data model constructs and their physical representations in a constructed project root and source files. This process, as detailed in the algorithm, consists of two parts.

The first part initializes the generator. The algorithm selects one of the cues (at random) from the PGCL script provided. Next, the algorithm constructs the system and project by

Algorithm 4.4: Pattern Generation Algorithm

Require: $\mathcal{R}$: RBML Pattern Specification
Require: $\mathcal{L}$: PGCL Pattern Cues

```
procedure GeneratePattern($\mathcal{P}$)
    $cue \leftarrow$ selectCue($\mathcal{L}$)
    initialize($cue$)
    $sys \leftarrow$ createSystem()
    createProject($sys$)
    $map \leftarrow [\,:\,]$
    for all $j \in \mathcal{R}.joins$ do
        $map[j.shared] \leftarrow$ selectOrCreateType(j.shared)
        for all $b \in j.blocks$ do
            if $b.src = j.shared$ then
                $map[b.dest] \leftarrow$ selectOrCreateType($b.dest$)
                createRelation($j.shared$, $b.dest$, $b.rel$)
            else
                $map[b.src] \leftarrow$ selectOrCreateType($b.src$)
                createRelation($b.src$, $j.shared$, $b.rel$)
            end if
        end for
    end for
end procedure
```

calling the *createSystem()* and *createProject()* methods, respectively. Once the generator is initialized the second part of the process commences.

The second part constructs the components represented in source code. This process starts by iterating across each joined set of *role blocks* (model blocks sharing a common classifier role). A *role block* is a relationship role connecting a pair of classifier roles (which may be the same Role). When creating the source and destination types, the *selectOrCreateType(...)* method is called. This method selects an existing Type (fulfilling the required Role) that does not participate in a model block fulfilling the current relationship Role. Otherwise, this method creates a new Type (class, interface, or enumeration) as long as the multiplicity upper bound of the Role is not exceeded. Once the source and destination roles are created/selected, the algorithm constructs the relation between them using the *createRelation(. . .)* method.

These creation methods (*createOrSelectType() and createRelation()*) produces two components. The first is an instance of one or more entities from the Arc data model. The second component is the physical representation of that item (whether it be folders, files, or file contents) on disk generated using a set of language-specific templates. These templates use PGCL cues to augment their output. Thus, once this algorithm is complete for a given pattern, a buildable project exists on disk, and the necessary components exist within the ArcDb.

4.3.2 Pattern Generation Cue Language

Design patterns help software engineers address commonly occurring design problems through a set of general solutions [96]. This layer of abstraction had led to a variety of known implementation approaches of each pattern type in a given language (where some are better than others). To address language-specific implementation details and to guide generation using well-known implementation methods, we developed an templating approach. This meta-model for this approach is depicted in Figure 4.4.

The primary components defined in the meta-modelare the *Cue*, *CueContainer*, *PatternCue*, *TypeCue*, *FieldCue*, *MethodCue*, and *CueManager*. The *Cue* shares its name with a given Pattern and provides the base code generation facilities. Additionally, the *Cue* is provided with the *templateText* which when combined with a component is used to generate code. Extending these capabilities is the *CueContainer* which also maintains a collection of child cues. Directly extending this *CueContainer* are the *PatternCue* and *TypeCue* which provide specialized handling for generating either an entire pattern, or a specific type within a pattern, respectively. Next, there are the *FieldCue* and *MethodCue* specializations of *Cue* which provide the behavior for generating fields or methods within the context of a *TypeCue*.

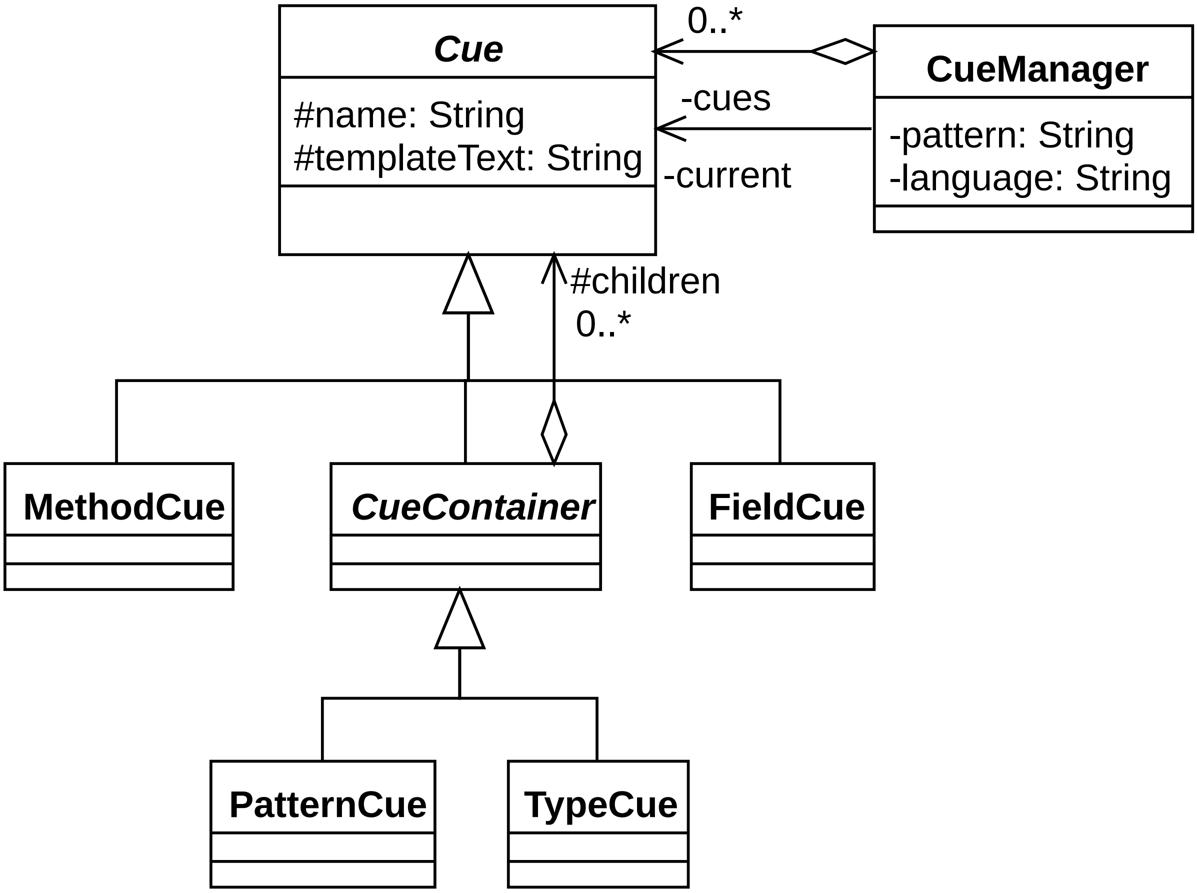


Figure 4.4: Pattern Generation Cue Language meta-model.

Finally, the *CueManager* provides the capability load cues for a given pattern and to select the next cue for that pattern.

Figure 4.5 depicts an example of the PGCL for a Java™ specific implementation of an lazy initialized Singleton. The example begins with the definition of the pattern cue within the section `start_pattern: LazyInit...end_pattern:LazyInit`. The "LazyInit" cue specifies two types with `start_type` blocks, each of which have a name matching a role defined within the RBML definition. These types specify that the system will override the default generated content using the template provided. Components of the type can be generated by adding the template for that

```
start_pattern: LazyInit
start_type: Singleton
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    protected [[InstName]]() {}

    [[methods]]
}
end_type: Singleton

start_type: ConcreteSingleton
/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: uniqueInstance
    private static [[Singleton.name]] [[uniqueInstance.name]];
    end_field: uniqueInstance
    [[fields]]

    private [[InstName]]() {
        super();
    }

    start_method: GetInstance
    public static [[Singleton.name]] [[name]]() {
        if ([[uniqueInstance.name]] == null)
            [[uniqueInstance.name]] = new [[InstName]]();
        return [[uniqueInstance.name]];
    }
    end_method: GetInstance

    [[methods]]
}
end_type: ConcreteSingleton
end_pattern: LazyInit
```

Figure 4.5: Example PGCL script for an lazy initialized singleton instance.

type within the definition of the type. For example, fields]] will generate all fields (unless any field is overridden). A field may be overridden by creating a field block

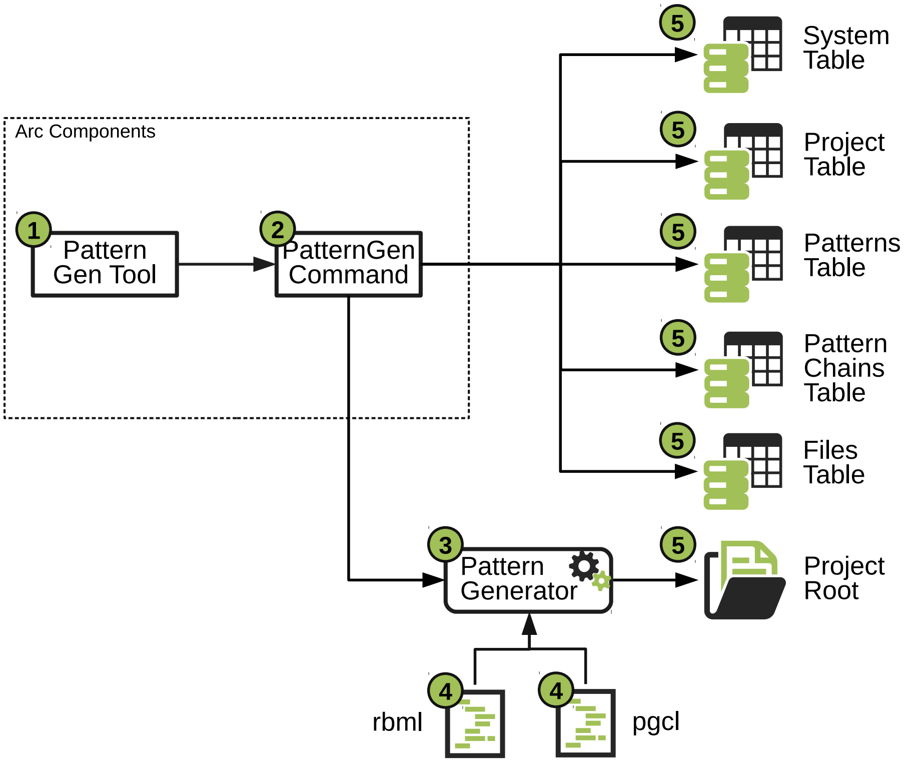


Figure 4.6: Integration of the Pattern Generator with the Arc Framework.

using the `start_field: <name> ... end_field: <name>` where name is either a field name or a role name. A roles name may be accessed using a template such as `[[Singleton.name]]` and the current instance (specific implemented type bound to the role being defined) may be accessed using InstName]]. Additional templates can be generated as shown in 4.5 including class comments (`[[ClassComment]]`), type definition header (`[[typedef]]`), and all defined methods (`[[methods]]`). This example is only one of the cues for a singleton, but the full definitions for each of the 16 design patterns (evaluated in Chapter 10) are defined in Appendix A.

4.3.3 Integration into the Arc Framework

Figure 4.6 depicts the integration of the pattern generation components and their flow of execution. This flow, the sequence of numbers encircled in green, is as follows. 1.) The

PatternGenTool provides the framework with an instance of the *PatternGenCommand*. 2.) The *PatternGenCommand* provides an interface between the pattern generation process and the ArcDb data model. It also executes the *PatternGenerator*. 3.) The *PatternGenerator* controls the execution of the pattern generation process. 4.) This process begins by loading two files for each pattern type to be created: the pattern-specific RBML and PGCL files. 5.) The results of this process are the creation of the project directory and supporting files, and the construction of pattern source code artifacts. These artifacts also have representation in the Arc data model. Steps 4 and 5 of this process repeated for each pattern instance needed.

4.4 Conclusion

In this chapter, we detailed the underlying methods and tools used to extract pattern data from software systems. Such data collection is not without its limitations. Specifically, collecting enough raw design pattern instances for experimentation is difficult and time-consuming. To alleviate this, we have detailed the development of a pattern instance generation algorithm. Though this algorithm and its results are necessary during experimentation, case studies require “wild” instances of patterns. To collect these “wild” instances from actual software systems, we identified the design pattern detection tools which will identify the raw instances. To improve the results of this and to cleanse the data set, we detailed the pattern coalescence algorithm, which also provides the ability to construct pattern chains. With these tools in place, we are capable of exploring the effects of grime on pattern instances.

CHAPTER FIVE

METRICS, QUALITY AND TECHNICAL DEBT

If you can not measure it, you can not improve it.

–Lord Kelvin

5.1 Introduction

The desire to measure the quality of software has existed nearly as long as software engineering [79]. The software industry, with the advent of better tools and processes, has been placing a higher priority on the use of quality analysis and measurement tools. The measurement of higher-level quantities such as Software Quality and Technical Debt is, typically, based on lower-level static analysis and metric aggregation. Facilitating this aggregation are models and approaches designed to operationalize the underlying quality aspects or (*-ilities*). Operationalizing quality models requires the ability to efficiently collect metrics measures and software issues in a general enough way that allows for a multitude of tools to be used. Our solution to this problem is the development of the Arc Framework, as detailed in Chapter 3.

In this chapter, we connect the components of the Arc Framework with the underlying concepts of software metrics analysis, software quality analysis, and technical debt analysis. These techniques form the basis of the organization of the sections of this chapter. The first section details our approach to integrating software metrics collection into the Arc Framework. The second section details our implementations of the Quamoco and SIG quality measurement approaches and their integrations with the Arc Framework. The third section details our approach to measuring technical debt and its integration with the Arc Framework. Finally, we conclude with a summary and segue with the upcoming chapters concerning the effects of design pattern grime on software quality and technical debt.

5.2 Metrics Analysis

A key component to an active software measurement effort is the identification of the correct metrics by which one may answer the questions at hand. This idea is central to any sophisticated software engineering enterprise in which one wishes to know what and where improvement may be gained and forms the basis of the Goal-Question-Metric paradigm [28]. Through our use of the GQM, as described in Chapter 1, we are intimately familiar with the necessity of identifying and utilizing the right metrics. Thus, this section is devoted to providing a more in-depth insight into our method of software metrics measurement.

5.2.1 Metrics Model

A software metric provides a knowledge extraction rule for encoding some software system aspects. These encoded aspects represented as either qualitative or quantitative measured values of metrics. Metrics can individually, or in combination, act as surrogates for quality attributes. Additionally, metrics provide system information useful in their own right. This capability has led researchers and industry advocates, seeking to improve both the software product and development process, to develop several metrics and measurement tools. Having used these tools, we have found that most lack a sufficient extension capability to meet our measurement needs. Thus, we have developed a metrics analysis system.

In developing this system, we divided the metrics along two axes: Direct/Derived and Source/Abstraction, as depicted in Figure 5.1. This figure depicts the subdivision of the set of all metrics into four subsets along the axes. The first axis indicates whether the Metric directly measures the artifact under analysis or if it is a metric combining other metrics. The second axis indicates whether the artifact measured is actual source code or some abstraction contained within a model of the software system (i.e., UML or CFG). Understanding this

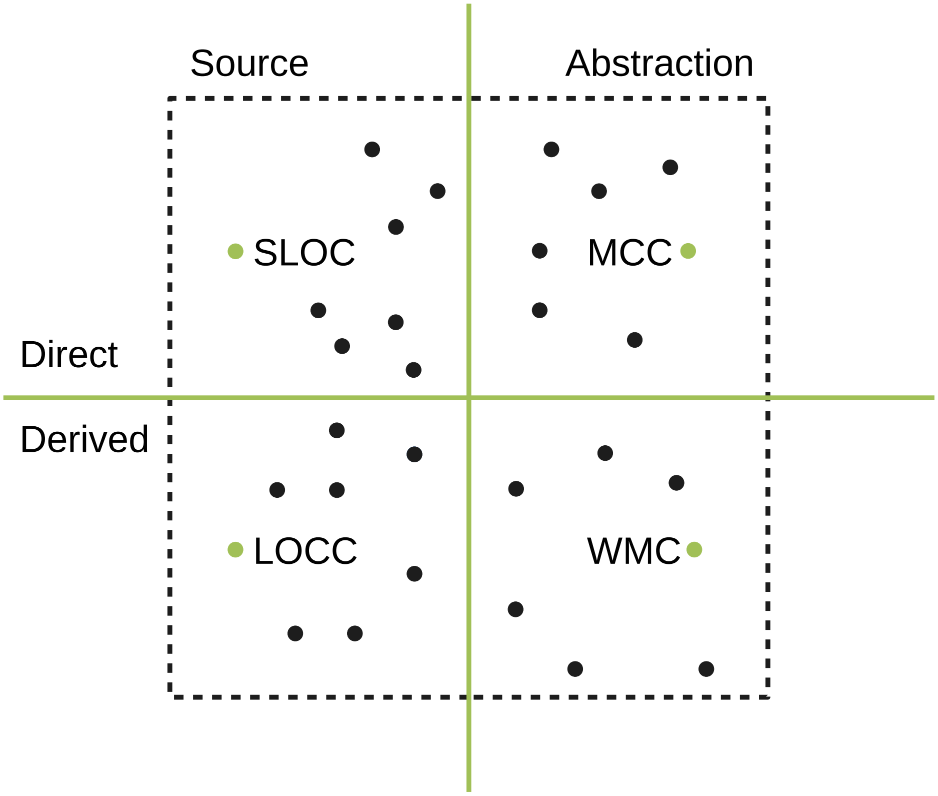


Figure 5.1: The axes of metrics division with examples shown.

division is better aided through a set of examples.

The following examples note several well-known software metrics and their location within the quadrant. The first is an example of a Direct-Source metric is *Source Lines of Code*(SLOC) [167], which is simply a count of the number of non-blank non-comment lines within a source code file. An example of a Direct-Abstraction metric is McCabe's Cyclomatic Complexity [190], which is a measure of the complexity of a method/function based on the control flow graph representation of a method's structure. An example of a Derived-Source metric would be *Lines of Code per Class* (LOCC), which is measured at the project or system level and is the average number of lines of code per class in the system. The following discusses the implementation of the metrics system.

Figure 5.2 depicts the metrics meta-model . The critical component of this model

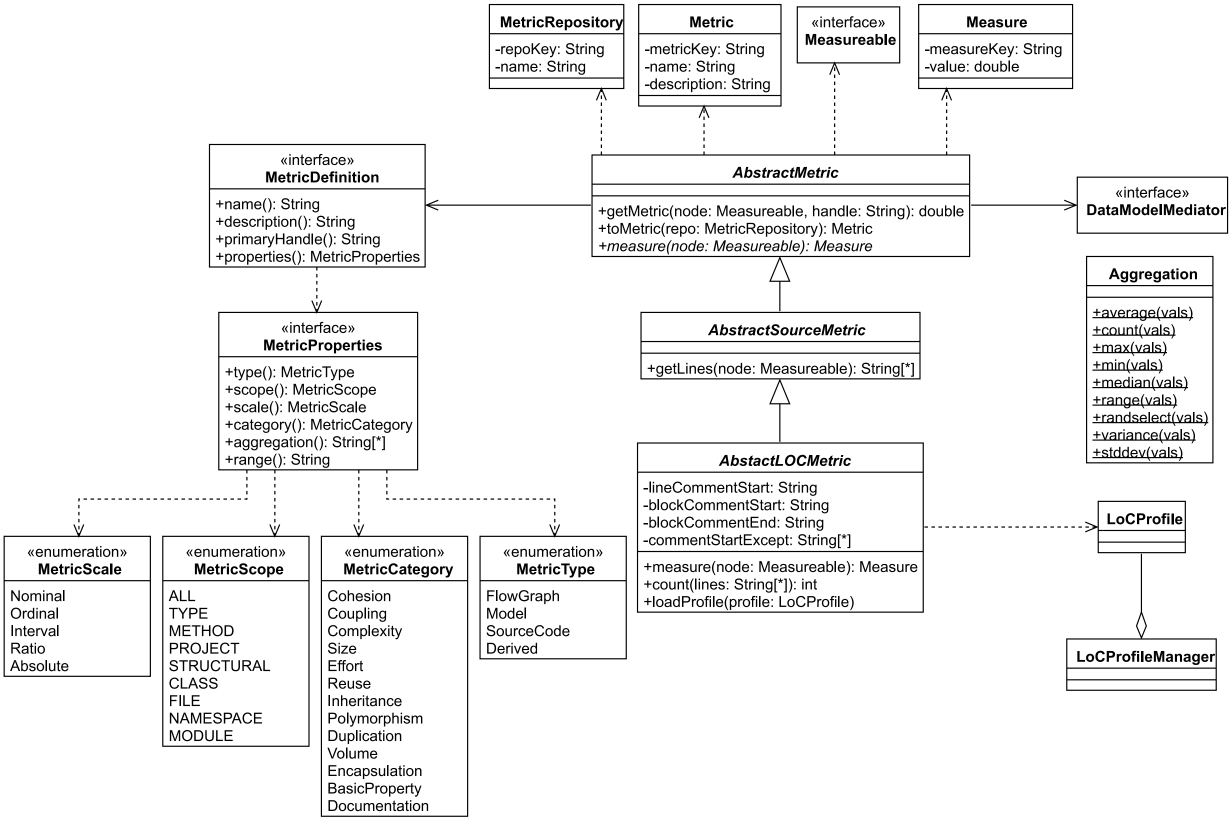


Figure 5.2: Metrics measurement system model.

is the base class *AbstractMetric*, which provides the abstract method `measure(...)` (containing the measurement logic). This method computes the Metric across a provided Component and returns this computed value encapsulated in a *Measure* object. The returned Measure is stored into the Arc data model by the Metric. Extending this base class is the *AbstractSourceMetric*, which simply provides the ability to extract the underlying source code for the provided *Measurable* item. The final main class, *AbstractLOCMetric*, provides the base capabilities to measure lines of code metrics. The latter class uses the *LoCProfile* and its manager to provide the language-specific values for the AbstractLOCMetric fields.

The final components of the meta-model link to the actual implementations. Each implemented Metric provides (through Java™/Groovy annotations) a *MetricDefinition*, which provides the metric with a name, a description, a primary handle (acronym) and a set of *MetricProperties*. The MetricProperties further describe the Metric based on the provided enumerations for the scale, scope, category, and type.

5.2.2 Implemented Metrics

The metrics analysis system currently includes implementations of several metrics needed to conduct software quality measurement and to detect design pattern grime within software systems. For the measurement of software quality (and specifically for normalizing findings) we measure the following metrics: Number of Statements (NOS) [167], Source Lines of Code (SLOC) [167], Number of Fields (NOF) [41, 48, 49], Number of Methods (NOM) [163], and Number of Classes (NC) [100]. For use when detecting Modular Grime, we have implemented the following metrics: Afferent Coupling (Ca) [182] and Efferent Coupling (Ce) [182] at the class level. For use when detecting Class Grime, we have implemented the following metrics: Tight Class Cohesion (TCC) [30] and Ratio of Cohesive Interactions (RCI) [40] at the class level. Finally, for Organizational Grime detection, we have implemented the following metrics: Instability (I) [182], Normalized Main Sequence Distance (D') [182], Common Closure (CC) and Common Reuse (CR) (both of these metrics are defined in Section 9.6) at the pattern and package level.

5.2.3 Arc Framework Integration

Figure 5.3 depicts the metrics system integration into the Arc Framework and its flow of execution. This flow follows two possible routes, the numbers encircled in green, as follows. Both routes start by 1.) initializing the ArcContext and the reading of the ArcConfig during Arc system initialization. 2.) The process then initializes the *MetricsTool* which provides both the *ArcMetricsProvider* and the *MetricsCommand*. At this point, the execution can fork. If the system is initializing the data model, then 3.a.) the ArcMetricsProvider will construct the Arc metrics repository and the Metric definitions associated with it. 4.a.)

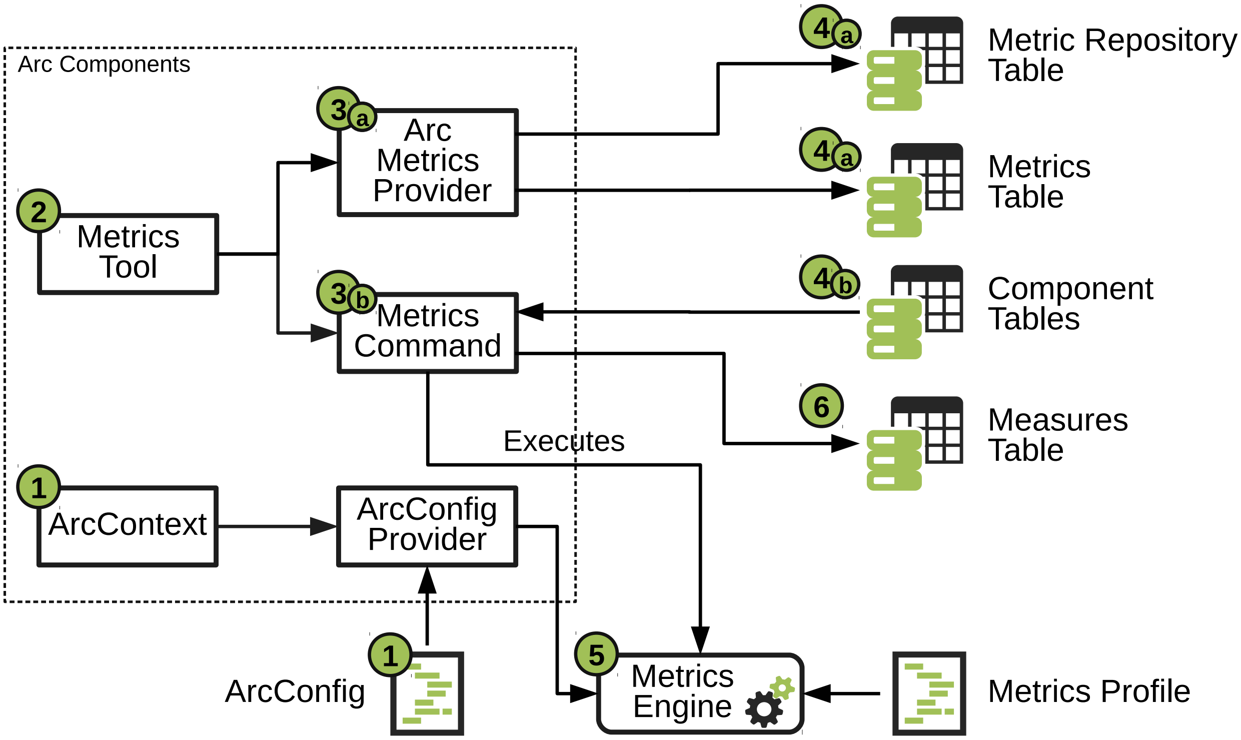


Figure 5.3: Integration of the metrics analysis system with the Arc Framework.

These values are then stored within the data model. If the system is in analysis mode, then 3.b.) the MetricsCommand extracts items from 4.b.) the Components section of the data model and utilizes the MetricsEngine to calculate metrics. 5.) The MetricsEngine using the information provided by a MetricsProfile, the ArcConfig, and the components to measure the needed metrics. A MetricsCommand invokes a Metric and encapsulates the returned value as a Measure. 6.) These Measures are then added to the data model by the MetricsCommand.

5.3 Quality Measurement

The field of Software Quality Assessment has, as described in Chapter 2, developed several descriptive quality models (such as the ISO/IEC 25010 standard). Each of these descriptive models comprises a set of characteristics and sub-characteristics. Though these characteristics and sub-characteristics define quality concepts, they do not define a means of assessment. Thus, model operationalization is a critical step in providing pragmatic solutions that can be readily adopted by software development organizations in the industry. Further, the deployment of operationalized quality models allows for continuous monitoring of the quality of an organization's software components.

The connection to higher levels of abstraction helps an organization's decision-makers assess potential economic impacts of breakdowns in quality in a holistic manner. To facilitate an understanding of quality issues that affect decision-makers as well as developers, we focused on the comparison between quality models [129]. This study led to our implementation of the Quamoco quality modeling and assessment approach within the SonarQube™ system. SonarQube™ś limitations required a shift towards the development of the Arc Framework. In the following section, we describe, in detail, the Quamoco architecture, method of assessment, and the integration within the Arc Framework.

5.3.1 Quamoco Quality Modeling

5.3.1.1 Quamoco Architecture As described in Chapter 3 we have developed a framework to meet the needs of our research and more generally the requirements for software measurement via an extensible architecture. Additionally, this architecture extends to incorporate the Quamoco quality modeling and assessment approach. This extension, depicted in Figure 5.4, has four key components: the *QuamocoTool*, the *QuamocoMetricProvider*, the *QuamocoCommand*, and the *QuamocoConfig*. The following describes the execution of Quamoco by the Arc system.

Figure 5.4 depicts the execution of Quamoco as one of two distinct paths, as indicated by the numbers encircled in green, as follows. Both paths start by 1.) the initialization of the *QuamocoTool*. At this point, the execution is dependent on the current mode of operation of the Arc system. 2.a.) During initialization mode, the *QuamocoMetricProvider* constructs the required Metrics and containing MetricRepository for the Quamoco implementation used

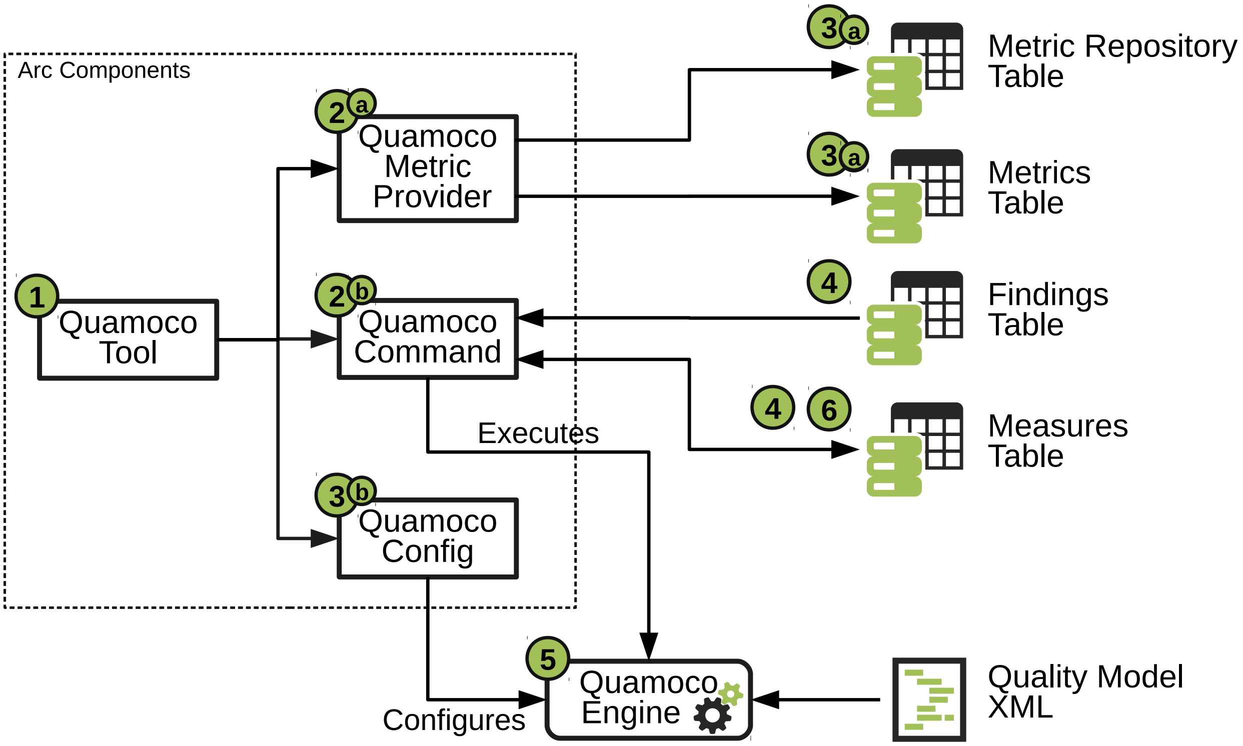


Figure 5.4: Integration of the Quamoco quality measurement approach with the Arc Framework.

in the QuamocoEngine. 3.a.) The QuamocoMetricProvider stores the MetricRepository and contained Metrics in the data model.

The *QuamocoCommand* initializes and operates the QuamocoEngine during system analysis mode. 3.b.) In this mode, a local *QuamocoConfig* configures the QuamocoEngine used by the QuamocoCommand. 4.) The QuamocoCommand extracts Findings (as provided by prior static analysis) and low-level measures (acquired during metrics analysis) and provides these to the QuamocoEngine to facilitate quality analysis. 5.) The QuamocoEngine, once configured and supplied with the necessary Findings and Measures, then loads the applicable QualityModel and calculates the system quality (as described in the following sections). 6.) Once the calculations are complete, Measures for each quality attribute produced by the QuamocoEngine are then stored in the data model by the QuamocoCommand.

5.3.1.2 Quamoco Processing The QuamocoEngine utilizes an external definition of a quality model, encoded in XML (possibly across multiple files). These files are instances of the Quamoco meta-model as defined by Wagner et al. [270, 271]. The meta-model , although useful for describing a quality model, provides far more detail than necessary to assess the quality of a software system accurately. Therefore we extract a reduced-form representation called the *processing graph*.

The processing graph is a directed acyclic graph distilled from a language-specific combined quality model. The model is processed to form a graph composed of four types of nodes, as depicted in Fig. 5.5. FactorNodes represent the higher-level abstractions related to quality characteristics and sub-characteristics. MeasureNodes correspond to lower-level issues (i.e. FindBugs rule ME_ENUM_FIELD_SETTER which detects methods within a Java™ enum, which sets the value of one of its fields[1]) applicable to entities found within source code (e.g., types, methods or fields). Finding and Value Nodes correspond to static analysis tool rules or metric values, respectively.

Each FactorNode has an attached Evaluator which handles the evaluation of afferent (incoming) measures through finding the mean of the normalized value of the findings set or value set, or through a weighted sum of afferent factors. Similarly, MeasureNodes each have an attached Aggregator applicable to the type of aggregation necessary: *union* or *intersection* for finding sets (propagated from attached finding nodes or other finding based measures) or *mean*, *min*, or *max* for ValueNodes. FindingNodes and ValueNodes provide the ability to collect either Findings (for named issues) or Values (for named metrics), respectively. Edges connect these different nodes and provide the path for aggregation.

Edges between FactorNodes provide the necessary afferent weights (i.e., coefficients of

[1] `http://findbugs.sourceforge.net/bugDescriptions.html#ME_ENUM_FIELD_SETTER`

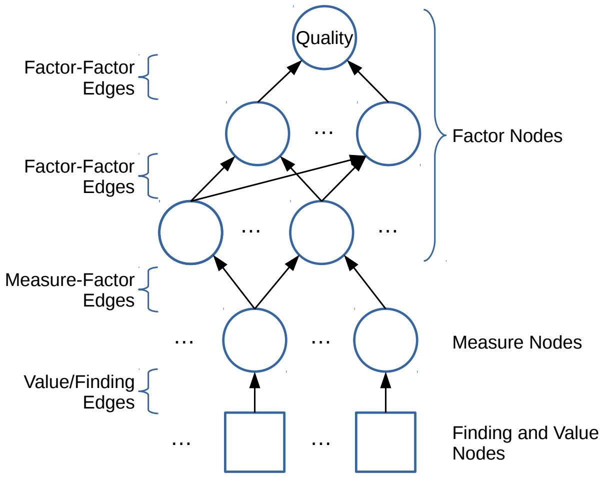


Figure 5.5: Representation of the processing graph.

source FactorNodes) used to aggregate the values at the destination FactorNode. Edges between FactorNodes and MeasureNodes, which convey sets of findings, provide a means to normalize the Finding set using an associated Normalization Measure and Range. These edges also provide a linearly increasing or decreasing function used to constrain the normalized value between 0.0 and 1.0.

5.3.1.3 Collecting Findings As shown in Figure 5.4 the QuamocoCommand extracts, from the Arc data model, Findings and Measures. A set of Findings provided to each FindingNode specific representing the Finding's Rule in the QualityModel. Thus, each provided Finding from the data model is added to the processing graph at a FindingNode with a matching Rule name and Repository name.

5.3.1.4 Evaluation of Quality The Quamoco model evaluates the quality of a system by aggregating the measures and issues affecting the system. These values form the lowest level of a Quamoco model hierarchy and provide input to the measure level. Each Measure refines another measure or is an input to a factor. A factor uses either the combination of measures or factors, but not both to compute its value. This value is always in the range

[0.0, 1.0] and represents the presence of that Factor within the software system. On the other hand, Measures pass up the hierarchy sets of Findings. Once these Findings reach a factor, they must be normalized into a value in the range [0.0, 1.0] representing the presence, in the system, of the underlying issue represented by the Finding.

Finding sets are normalized by summing a normalization measure (such as SLOC) across the entities (i.e., a method, class, or file) where the Findings occur. This sum, reduced by dividing by the system-level summation of the same Metric, acts as input to a linear increasing/decreasing function. This function converts the cardinality of the Finding set to a value in the range [0.0, 1.0] suitable for use by a Factor.

A Factor that is evaluated by a set of other Factors calculates its value using a weighted sum. The weights, assigned to each incoming Factor, derive from that Factor's assigned rank. This derivation uses the *Rank-Order Centroid* method [27] and *Swing* approach [70], such that the generated weights then sum to 1.0. The model stores the weights along the edges to facilitate a simplified graph processing algorithm for quality evaluation.

Quality evaluation occurs through a simple recursive depth-first search based algorithm. The algorithm starts at the sink Factor, "Quality", then requests the values for each incoming edge. This process continues recursively requesting the values of the source side node for each incoming Factor. When the algorithm reaches a measure to factor edge, it either requests the set of findings or the set of values from the source (depending on the type of Measure the source side is). The recursion stops upon reaching either a FindingNode or ValueNode. The values/finding sets are propagated back up the graph. During the propagation stage, the algorithm aggregates these values/finding sets as they pass through each processing graph node, stopping at the original start node.

Publishing Quality Information Once the processing graph has completed the evaluation, the QuamocoCommand extracts and encapsulates high-level quality attribute values

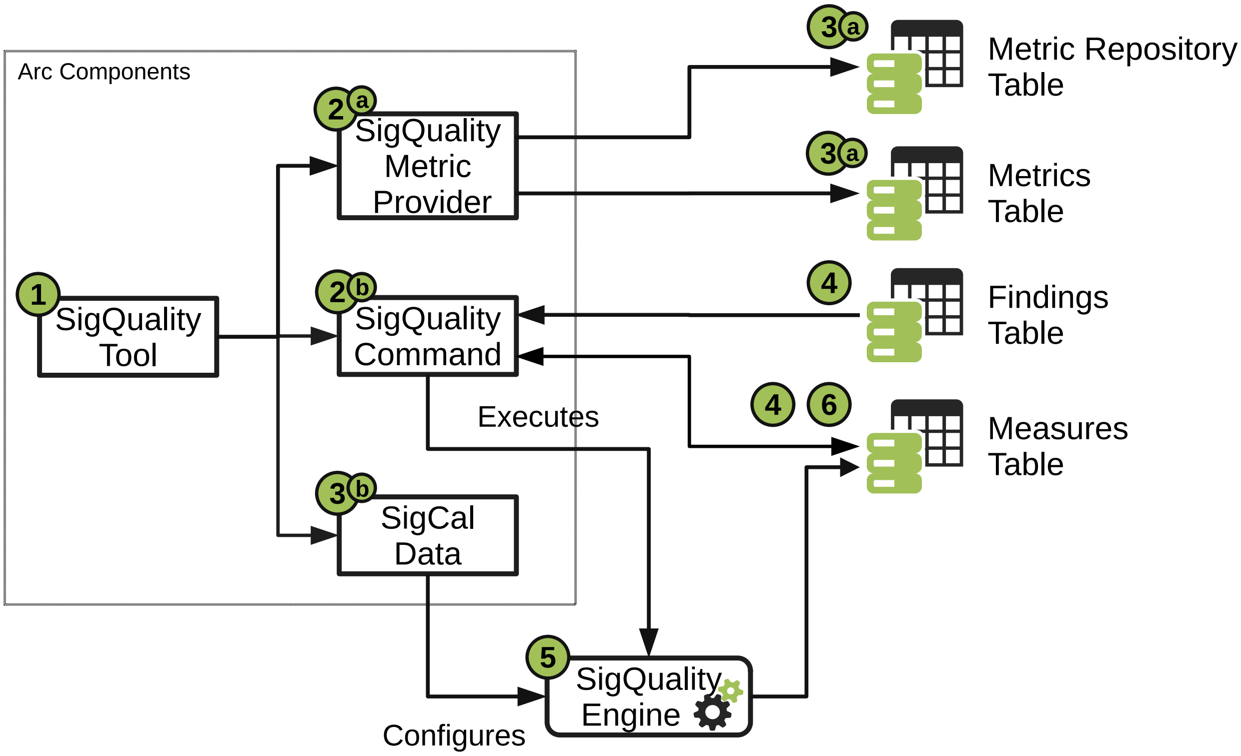


Figure 5.6: Integration of the SIG Maintainability Model quality measurement approach with the Arc Framework.

as Measures in the data model. Thus, the data model provides a simple means to access quality information through a request for the specific Measure of concern.

5.3.2 SIG Maintainability Model

In addition to the Quamoco quality model we have also incorporated the SIG Maintainability Model. This extension, depicted in Figure 5.6, has four key components: the *SigQualityTool*, the *SigQualityMetricProvider*, the *SigQualityCommand*, and the *SigCalData*. The following describes the execution of the SIG Model by the Arc system.

Figure 5.6 depicts the execution of the SIG Maintainability Model as one of two distinct paths, as indicated by the numbers encircled in green, as follows. Both paths start by 1.) the initialization of the *SigQualityTool*. At this point, the execution is dependent on the current

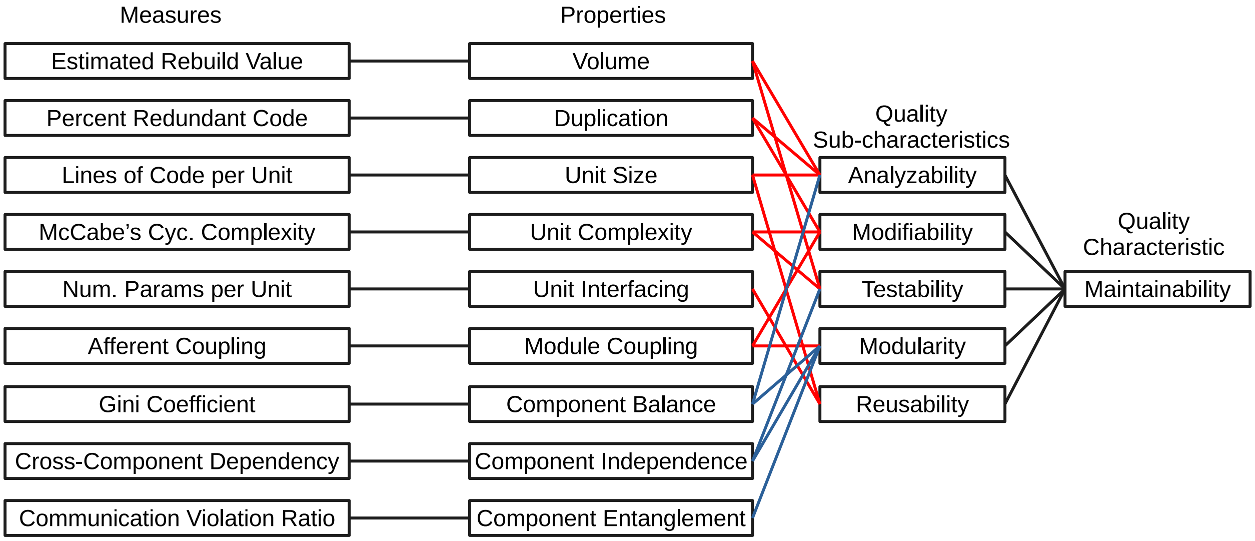


Figure 5.7: SIG Maintainability Model [268].

mode of operation of the Arc system. 2.a.) During initialization mode, the *SigQualityMetricProvider* constructs the required Metrics and containing MetricRepository for the SIG implementation used in the SigQualityEngine. 3.a.) The SigQualityMetricProvider stores the MetricRepository and contained Metrics in the data model.

The *SigQualityCommand* initializes and operates the SigQualityEngine during system analysis mode. 3.b.) In this mode, *SigCalData* configures the SigQualityEngine with calibration data used by the SigQualityCommand. 4.) The SigQualityCommand executes the SIG quality measures and provides these to the SigQualityEngine to facilitate quality analysis. 5.) The SigQualityEngine, once configured and supplied with the Measures, then performs the ratings and calculates system quality or produces calibration data (depending on the type of analysis being conducted). 6.) Once the calculations are complete, the Measures for each quality attribute produced by the SigQualityEngine are then stored in the data model by the SigQualityEngine.

5.3.2.1 SIG Maintainability Model Quality Measurement The SIG Maintainability Model, depicted in Figure 5.7, is comprised of four layers. The Quality Characteristic, Maintainability, represents the target of quality assessment. With that in mind, it should be noted that the Quality Characteristic and Quality SubCharacteristics are directly taken from the ISO/IEC 25010 descriptive quality model. The premise underlying this model is found throughout software quality modeling literature. It is the basis of each of the models described herein. The idea is that quality characteristics cannot be directly measured [126]. Thus, we require surrogate measure(s) to represent the quality characteristic. We then aggregate the values of these underlying factors or properties to provide a meaningful measure of the quality characteristic of concern. Furthermore, Siavvas et al.have suggested that identifying simpler models which are both limited in depth and have only a single measure per property/factor is ideal [242]. From this perspective, the SIG Maintainability Model seems ideal, at least for Maintainability, as each property is directly connected to a single measure. The properties are defined as follows:

- **Volume** - The overall size of the source code of the software product. Size is determined from the number of lines of Code per programming language normalized with industry-averaged productivity factors for each programming language. Volume should be rated on a scale that is independent of the type of software product.

- **Duplication** - The degree of duplication in the source code of the software product. Duplication concerns the occurrence of identical fragments of source code in more than one place in the product.

- **Unit Complexity** - The size of the source code unit (methods or functions) in terms of the number lines of code.

- **Unit Size** - The degree of complexity in the units of source code.

- **Unit Interfacing** - The size of the interfaces of the units of the source code in terms of interface parameter declarations

- **Module Coupling** - The coupling between modules (classes) in terms of the number of incoming dependencies for the modules of the source code. The notion of module corresponds to a grouping of related units.

- **Component Balance** - The product of system breakdown, which is a rating for the number of top-level components (packages containing classes) in the system, and the component size uniformity, which is a rating for the size distribution of those top-level components. The notion of top-level components corresponds to the first subdivision of the source code modules of a system into components, where a component is a grouping of source code modules.

- **Component Independence** - A rating for the percentage for code in modules that have no incoming dependencies from modules in other top-level components. The choice of top-level components will affect which dependencies will be considered to be from the outside of the top-level component.

- **Component Entanglement** - Indicates the percentage of communication between top-level components in the system that are part of commonly recognized architecture anti-patterns.

Having discussed the layers and underlying ideas, we now need to discuss how the layers work together. One of the most challenging issues in software metrics is that nearly all metrics are on different scales and have very different ranges of values. Thus, most quality models transform the values from metrics in some fashion to a suitable range of values consistent across the model. In the SIG Maintainability Model, the properties serve this convenient purpose. The idea is that the Measures provide a *raw* value, which is then rated and converted into an integer rating quality. In addition to providing the rating, the properties also consider how they affect the related quality characteristics.

In Figure 5.7 these relationships are shown as either red or blue links between the Properties and the Quality Sub-characteristics. The red links indicate that the property negatively impacts the connected characteristic, and the blue links indicate that the property positively affects the connected characteristic. For example, Analyzability is negatively affected by Volume, Duplication, and Unit Size but is positively affected by Component Balance. When considering these relationships, the rating approach needs to ensure that the standard aggregation technique will continue to work. Thus, the interpretation for rating is typically one in which properties that negatively affect quality will be rated lower for higher values of the raw measure. In comparison, those which positively affect quality will be rated higher for larger values of the raw measure. The SIG approach for aggregating Properties into Quality Sub-characteristics is a simple weighted average of the rated values, as shown in Table 5.1.

5.3.2.2 Rating Raw Values In the SIG Maintainability Model, all measures are rated on a scale of 0 - 5 stars. The raw values are mapped to these ratings, using one of two approaches. The first is direct rating, in which a calibrated rating table is used to assign a rating of stars to the observed value. The second approach is a risk profile-based approach which is a two-step approach. The first step uses the raw value to look up the risk category. Where the risk category is one of: *Low*, *Moderate*, *High*, or *Very High*. In the second step, each entity's volume (i.e., lines of code) measured in the first step is summed for each risk category. These sums are then divided by the system's total volume to find the percentage by volume for each risk category. These values form what is known as the *risk profile* for that measure for the system. These values are used to find the rating by comparing against a table of risk profiles linked to star ratings. The mapping of Property and associated Measures to

Table 5.1: Calculation of quality characteristics in the SIG Maintainability Model.

Characteristic	**Calculation**
Analyzability	$0.25 * Volume + 0.25 * Duplication + 0.25 * UnitSize$ $+ 0.25 * ComponentBalance$
Testability	$0.33 * Volume + 0.33 * UnitComplexity$ $+ 0.33 * ComponentIndependence$
Modularity	$0.25 * ModuleCoupling + 0.25 * ComponentBalance$ $+ 0.25 * ComponentIndependence$ $+ 0.25 * ComponentEntanglement$
Modifiability	$0.33 * Duplication + 0.33 * UnitComplexity$ $+ 0.33 * ModuleCoupling$
Reusability	$0.5 * UnitSize + 0.5 * UnitInterfacing$
Maintainability	$0.2 * Analyzability + 0.2 * Testability + 0.2 * Modularity$ $+ 0.2 * Modifiability + 0.2 * Reusability$

the type of rating used is shown in Table 5.2. In addition to this base implementation of the SIG Maintainability Model we made an adjustment to how the rating is created. Instead of simply mapping to a whole star, the rate will assign the whole star number. Additionaly, it will linearly interpolate between the rating min and max values for the value to be rated and add the decimal value to the assigned star value. In the case that the rating is 1-star, interpolation is not performed as there often is no max value for 1-star ratings.

Let us consider an example for rating the Volume of a system that uses a direct rating approach. First, we need a raw value for the Estimated Rebuild Value of the system. In this example, we will consider the system to have an Estimated Rebuild Value of 31.5 man-years. Next, we need to find the rating for this value. This transformation requires a calibrated

Table 5.2: SIG Maintainability Model Property and Measure rating types.

Property	**Measure**	**Rating Type**
Volume	Estimated Rebuild Value	Direct
Duplication	Percent Redundant Code	Direct
UnitSize	Lines of Code per Unit	Risk Profile
UnitComplexity	McCabe's Cyclomatic Complexity	Risk Profile
UnitInterfacing	Number of Parameters per Unit	Risk Profile
Module Coupling	Afferent Coupling	Risk Profile
Component Balance	Gini Coefficient	Direct
Component Independence	Cross-Component Dependency	Direct
Component Entanglement	Communication Violation Ratio	Direct

Table 5.3: Example rating table for Volume

Rating	**Man-Years**
☆☆☆☆☆	0 - 8
☆☆☆☆	8 - 30
☆☆☆	30 - 80
☆☆	80 - 160
☆	>160

rating table that converts man-years (or even KLOC) to stars. For this example, we will use Table 5.3. We then identify the correct rating for our raw value, which in the case of 31.5 man-years is a rating of 3 stars without interpolation and 3.03 with interpolation. Finally, the rating is assigned to the value of Volume.

Next, let us consider an example for rating the UnitSize for a system. Before we start

Table 5.4: LOC per Unit to Risk Category mapping.

LOC per Unit	**Risk Category**
0 - 15	Low Risk
15 - 30	Moderate Risk
30 - 60	High Risk
>60	Very High Risk

the process, we must understand that in the terminology of SIG, a *unit* is a method, even though we are rating the property at the system level. Thus, the first step is to calculate the raw value for Lines of Code per Unit for each unit in the system. For example, let us suppose that we have a system composed of two classes, each with three methods and each method having the given sizes in Table 5.5.

Using the values mapping of LOC per Unit to Risk Category in Table 5.4 we can determine the risk category for each method, as shown in Table 5.5. Using this, we then calculate the risk profile for the measure for the system. This calculation requires summing the LOC for each risk category and dividing by the total system size. Thus, the risk profile for this system is as follows: (LOW = 10.34%, MODERATE = 31.03%, HIGH = 58.62%, VERY HIGH = 0.0%).

Next, we use this risk profile to rate the system for Unit Size. This rating requires us to have a calibrated rating table, such as the one in Table 5.6. Using this table, we compare the risk profile, finding the rating level for which the worst category for which a value exists is not exceeded. In the case of the example system, this would be the HIGH risk category. As a result, we have a Relative Volume Percentage of 58.62%, which exceeds the maximum for all categories and places this system at a 1-star rating for Unit Size.

Table 5.5: Example system characteristics.

Class	**Method**	**LOC per Unit**	**Risk Category**
ClassA	methodA	25	Moderate
	methodB	50	High
	methodC	10	Low
Class A Size		85 LOC	
ClassB	methodD	5 LOC	Low
	methodE	35 LOC	High
	methodF	20 LOC	Moderate
Class B Size		60 LOC	
System Size		145 LOC	

Table 5.6: Example risk profile rating table for Unit Size

	Maximum Relative Volume		
Rating	**Moderate**	**High**	**Very High**
☆☆☆☆☆	30.0%	5.0%	0 %
☆☆☆☆	41.6%	18.2%	5.2%
☆☆☆	50.0%	25.0%	7%
☆☆	60.0%	30%	10%
☆	–	–	–

5.3.2.3 SIG Maintainability Model Calibration As noted in Section 5.3.2.2, to rate the raw measurements and provide values for the Properties in the SIG Maintainability Model, we need calibrated rating tables. The SIG process for this is to evaluate an expert-curated set of software projects. This set of projects should contain projects from a variety of

Table 5.7: Calibration distribution

Rating	**% of Projects**
☆☆☆☆☆	5% of projects
☆☆☆☆	30% of projects
☆☆☆	30% of projects
☆☆	30% of projects
☆	5% of projects

categories, sizes, and programming languages (if considering more than one) and have a total volume of at least 10 million lines of code [268]. For this research, we utilized the *Qualitas.Class Corpus* which is a curated collection of compiled open source Java™software systems [256]. The *Qualitas.Class Corpus* is a derivative of the *Qualitas Corpus* [255]. The projects in the corpus meet all of the requirements for calibration, with the exception of multiple programming languages.

In calibrating our implementation of the SIG Maintainability Model, we randomly selected 106 of the 111 systems from the corpus. We then collected the raw values for each measure direct rating measure and the percentage relative volume values for each risk profile-based rating measure. To then construct the rating tables, according to SIG's documentation, the values must follow the distribution defined in Table 5.7.

5.3.3 Selecting a Quality Model

Both the Quamoco and SIG models can evaluate a software system's quality following the ISO/IEC 25010 standard. However, both of these models have their pros and cons. Specifically, the Quamoco approach does provide the ability to evaluate all characteristics of the ISO 25010, while the SIG model is limited to Maintainability and its sub-characteristics. Additionally, the SIG model is simpler and easier to interpret, while the Quamoco model is complex. Finally, the Quamoco model was developed with engineers in mind, aggregating detected issues, thereby facilitating the identification of exact locations in the software that can be remediated. Alternatively, the SIG model is a metrics base model which gives a more general sense of the quality issues but does not identify the same components of the software with quality issues. Although, comparatively, from an engineer's perspective, this may suggest that the Quamoco model is preferred. Unfortunately, this is not the perspective necessary for our research. We are concerned with understanding how design disharmonies affect quality. This very fact suggests that the tools which would identify such issues are not included in the model. Therefore a model such as Quamoco is less desirable, while a more general metrics-based model is preferred. Thus, we have selected to use the SIG model over the Quamoco model (or similar issues-based approaches, i.e., QATCH [242]).

5.4 Technical Debt Measurement

The growing concern for technical debt and its lasting effects, has prompted the development of several methods of estimating a software system's level of technical debt [50, 59, 60, 99, 160, 161, 181, 203]. To date, there have been few studies which evaluate the effect of issues considered technical debt on software quality indicators. This relationship is key to understanding software disharmonies. First, though, we must have an approach to estimate a system's current level of technical debt.

5.4.1 Calculating Technical Debt

In prior work [103], we evaluated the connection between several technical debt estimates and a known quality model. We found that of all the technical debt estimation approaches evaluated, the CAST method [59, 60] was the most accurate (of those methods evaluated) concerning the current definition of technical debt. However, there are potentially more accurate methods available, such as the approach proposed by Nugroho et al. [203], at

Table 5.8: Values for models of TDE as proposed by Curtis, Sippidi, and Szynkarski [60,103].

	Severity	**Model 1**	**Model 2**	**Model 3**
Percent of Findings to be Fixed	High	50%	100%	100%
	Medium	25%	50%	—
	Low	10%	—	—
Time to Fix	High	1 hr	2.5 hrs	10% – 1 hr 20% – 2 hrs 40% – 4 hrs 15% – 6 hrs 10% – 8 hrs 5% – 16 hrs
	Medium	1 hr	1 hr	—
	Low	1 hr	—	—
Cost to Fix	All	$75	$75	$75

the time of our prior study we did not have the empirical data nor was an implementation of Nugroho et al.'s approach available. In the following subsections, we describe the implementation of both the CAST and Nugroho approaches to technical debt estimation.

5.4.1.1 CAST TD Principal Estimation The CAST approach focuses on estimating the technical debt principal (effectively the cost/effort to remediate the underlying issue) using a static analysis based parameterized cost model. Three key parameters guide this model's operation: (i) Percent of Finding to be Fixed per Finding Severity Level, (ii) Hours to Fix per Finding Severity Level, and (iii) The Cost per Hour per Finding. Parameters values are prescribed by Curtis, Sippidi, and Szynkarski [60] defining the three models shown in Table 5.8.

The values of the parameters *time to fix* and *cost to fix* calculate a monetary value based on the percentage of findings to fix. These values combine in the following equation to estimate the Technical Debt principal using the values from Table 5.8 and the counts of collected Findings:

$$\begin{aligned} TDE &= (\Sigma HS * \%HS * HS_{fix} * HS_{cost}) \\ &+ (\Sigma MS * \%MS * MS_{fix} * MS_{cost}) \\ &+ (\Sigma LS * \%LS * LS_{fix} * LS_{cost}) \end{aligned} \tag{5.1}$$

Where ΣHS, ΣMS, and ΣLS are the counts of *high severity*, *medium severity*, and *low severity* violations, respectively. The variables $\%HS$, $\%MS$, and $\%LS$ represent the percentages of high, medium, and low severity violations intended to be fixed. The variables HS_{fix}, MS_{fix}, and LS_{fix} represent the average time (in hours) required to fix per instance of each severity level. Finally, the variables HS_{cost}, MS_{cost}, and LS_{cost} represent the monetary cost per hour to perform each fix.

5.4.1.2 Nugroho et al.'s Method to Estimate TD Principal and Interest Nugroho et al. [203] developed an empirical model of Technical Debt Principal and Interest founded upon the SIG Maintainability Model. Unlike the CAST approach, this method is directly related to a validated ISO/IEC 25010 Quality Model for maintainability, the quality characteristic most closely associated with Technical Debt [164]. To the best of our knowledge, there is no other implementation of this approach. The following describes the methods of calculating Technical Debt Principal and Interest and our implementation of this technique into the Arc Framework.

Technical Debt Principal In this approach, TD Principal is considered to be the cost of improving a software system to the ideal level [203]. The ideal level, in this case, would be a system with a 5-star rating using the SIG Maintainability Model. Thus, the TD Principal becomes the estimated Repair Effort needed to conduct the improvements. The Repair Effort (RE) can be calculated as follows:

$$RE = RF * RV * RA$$

Where RF is the Rework Fraction, which estimates the percent change in system size needed to reach the ideal level. This value is determined using the table provided by Nugroho et al. (reproduced in Table 5.9 for the reader's convenience). RF can then be estimated by using the table to identify the column representing the current rating of the system's maintainability and then finding the row associated with the desired level to be achieved. The intersecting cell then contains the appropriate RF. In our implementation of the SIG Maintainability Model, we have applied a linear transform to allow for continuous values in the range of 1.0 to 5.0 for quality ratings. Thus, the source value will not fall directly on any one category. Therefore, to calculate the correct RF, a similar linear transformation is applied, as follows:

$$RF = \frac{R_t - R_s}{R_t - \lfloor R_s \rfloor} * RF(\lfloor R_s \rfloor, R_t)$$

Where R_t is the target rating value, R_s is the source rating value for Maintainability, and $RF(s,t)$ is the value of the cell in Table 5.9. RV is the Rebuild Value, which estimates the effort (in man-months) required to reconstruct the system using a given technology. RV can be calculated as follows:

$$RV = SS * TF$$

Where SS is the System Size in KLOC and TF is the Technology Factor. The Technology

Table 5.9: Rework Fraction table [203].

	Source				
Target	☆	☆☆	☆☆☆	☆☆☆☆	☆☆☆☆☆
☆					
☆☆	60%				
☆☆☆	100%	40%			
☆☆☆☆	135%	75%	35%		
☆☆☆☆☆	175%	115%	75%	40%	

Factor is a language-dependent productivity factor that provides a means to convert a line of code into man-months. TF is derived from tables used for ”back-firing“ LOC to Function Points and then converting Function Points to man-months based on the average function points per month in the given language. In the case of Java™ (used in the studies describe in Chapters 10 and 11) the value is 0.00136 [203]. Finally, RA is the Refactoring Adjustment, a discounting factor applied for teams using advanced development tools to reduce required repair effort. In the case of our system, we have selected to use the value a value of 0.10, as suggested by Nugroho et al.

Technical Debt Interest In Nugroho et al.'s approach Technical Debt Interest is the extra maintenance cost incurred due to Technical Debt. Thus, TD Interest, TDI, becomes the difference between the maintenance effort at the current quality level, ME_C, and the maintenance effort at the ideal quality level (5-stars), ME_I. Thus, TDI is calculated as follows:

$$TDI = ME_I - ME_C$$

The Maintenance Effort in man-months, ME, can be calculated as follows:

$$ME = \frac{MF * RV}{QF}$$

Where RV is the Rebuild Value (calculated as it was for Repair Effort), the next component is the Maintenance Fraction, MF, which is the yearly maintenance effort of the system. While this value may be estimated from historical data, this is an incredibly challenging problem when working with open-source systems. Nugroho et al. note that SIG estimates that 15% of yearly changes to a system's code are due to maintenance. Thus, in our implementation, we have assumed this value. The last component in the calculation of ME is the Quality Factor, QF, which accounts for the current quality level. The underlying assumption is that as the quality level decreases the amount of maintenance effort increases. Thus, QF acts as a penalty increasing the value of the maintenance effort for lower values of $QualityLevel$. QF is calculated using the following formula:

$$QF = 2^{((QualityLevel-3)/2))}$$

This formula results in a value of 2 for a current quality level of 5 stars. However for a value of 1 for the current quality level, this results in a value of 0.5 which increases the maintenance effort. Furthermore, the values that can be generated using the QF equation are in line with prior empirical results from Bijlsma [32].

5.4.1.3 Selecting a Method Arguably both methods of calculating TD Principal are based on the competing empirical analysis of systems. Both SIG and CAST collect data on software systems and, using this information, have constructed models representing technical debt. However, there are two readily apparent differences between these models. The first is that Nugroho et al.'s approach is directly based upon the SIG Maintainability Model and hence comes closer to accepted definitions of technical debt. Though this model is perhaps more accurate in evaluating the level of technical debt a system has, it does little for the engineer tasked with remediating the technical debt and for managers needing to make decisions regarding technical debt issues to include in upcoming releases. That said, it does provide the ability to monitor the technical debt level and to make high-level decisions regarding project direction. On the other hand, the CAST method utilizes well-known issues and can be directly traced back to individual items and their location in code. Thus, decisions at the code level made by team leads and engineers regarding planning and system health can be made if the effect of those items is known. For this research, as existing tools such as PMD and SpotBugs do not identify grime directly, the CAST method would not necessarily provide the best method for evaluating grime's effect on technical debt. This is similar to the reasoning used in selecting a quality measurement approach.

5.4.2 Technical Debt Measurement Architecture

The evaluation of a Software System's Quality and Technical Debt is one of the primary motivations of the Arc Framework. Thus, the Arc Framework integrates the above method of technical debt estimation. This integration and its flow of execution are depicted in Figure 5.8. The execution flow, the numbers encircled in green, follows two possible paths. Both paths begin with 1.) the initialization of the Arc system which then provides an *ArcContext* initialized with a *ArcConfig* provided via an *ArcConfigProvider*. 2.) Next, the system initializes the TechDebtTool. The TechDebtTool provides two major components: the *TechDebtMetricProvider* and the *TechDebtCommand*. At this point, the execution path forks and depends upon the operational mode of the Arc system.

When the Arc system in is the data model initialization mode 3.a.) the TechDebtMetricProvider constructs the MetricRepository and Metrics and, 4.a.) adds them to the data model. When the Arc system is in the system analysis mode 3.b.), the TechDebtCommand

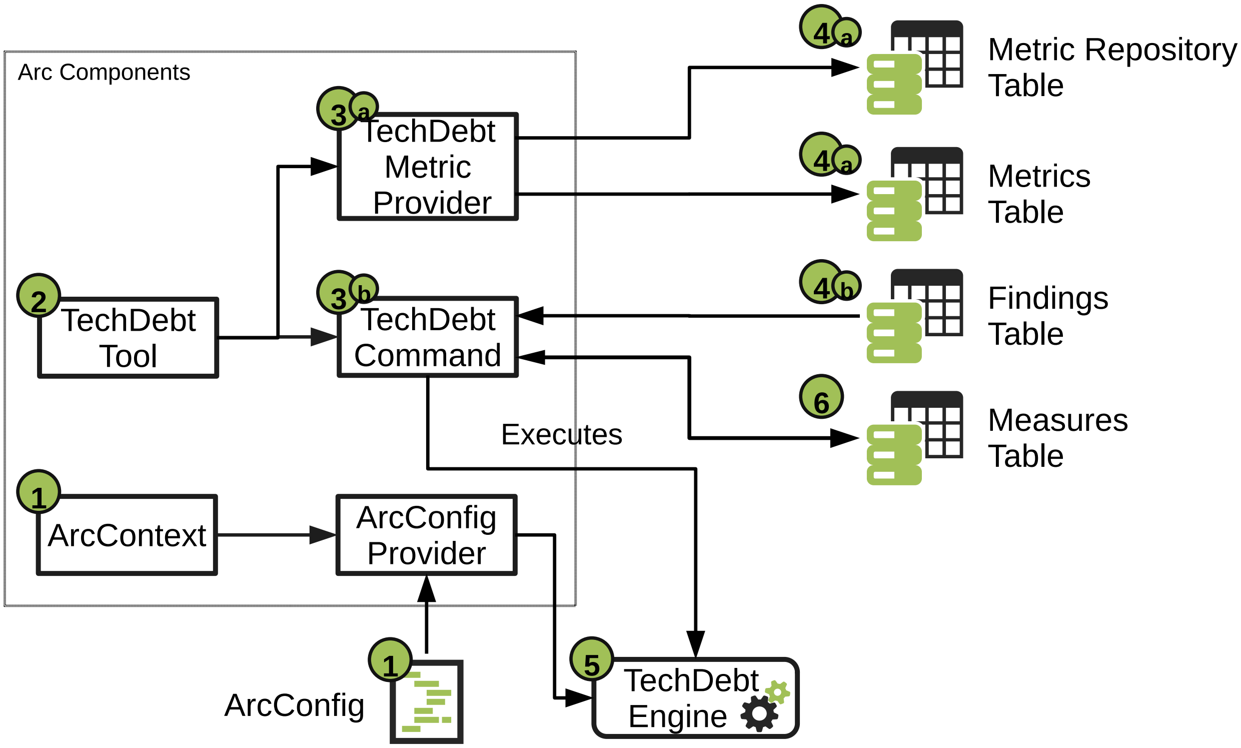


Figure 5.8: Integration of technical debt measurement system with the Arc Framework.

controls the technical debt analysis. 4.b.) This analysis extracts static analysis findings and measures for the system under analysis and passes this information to the TechDebtEngine. 5.) The TechDebtEngine uses these findings, measures, and ArcConfig information to calculate the system's technical debt value. 6.) The calculated value is returned to the TechDebtCommand, wherein it is encapsulated into a Measure and added to the data model.

5.5 Conclusion

This chapter provides the details concerning the measurement of both software quality and technical debt necessary for the experiments and case studies found in Chapters 10 through 11. This chapter further develops the reasoning for the construction of the Arc Framework, while also describing the integration of the metrics, quality, and technical debt analysis framework components. These components along with those defined in Chapters 3, 4, 6, and 7 implement the central ideas of the method presented in Chapter 8 which lays out this research's guiding principles and processes.

CHAPTER SIX

SOFTWARE INJECTION

Program testing can be used to show the presence of bugs, but never to show their absence!

–Edsger Dijkstra

6.1 Introduction

Currently, design disharmony research lacks (excluding code smells and antipatterns) automated identification and verification techniques. Without such techniques, the ability to cultivate design disharmony datasets is quite limited. These limitations, in turn, have slowed the progress in evaluating the effects that grime has on quality and technical debt and restricting research to observational studies. Such studies reduce the scope of analysis and preclude the ability to evaluate causal relationships. Thus, to introduce causal analysis, through experimentation, this Chapter describes a framework for the injection of design disharmonies. Injection allows for the controlled introduction of design disharmonies instances into software artifacts, thereby removing the original limitations.

This Chapter is organized as follows. Section 6.2 describes the architecture of the process governing our software injection system. Section 6.3 provides definitions for the injection of design pattern grime into design pattern instances. Section 6.4 details potential applications beyond design pattern grime for this framework. Finally, Section 6.5 concludes this chapter.

6.2 Software Injection Architecture

Software Injection is a method to include entities of study through a software system transformation. These transformations occur through the serial application of a set of operators. Each operator either modifying or creating artifacts to exemplify affliction with the disharmony injected. The Software Injection meta-model, depicted in Figure 6.1, contains the process' necessary components and basic architecture. The following subsections describe this architecture.

6.2.1 Software Injection Metamodel

The meta-model consists of two main sections: (i) the components on the left that enact the operators described by (ii) the components on the right, and the *InjectionContext* separates both sections. The section on the right describes the transformation operators necessary to effect the injection of a disharmony. The primary injection component, the interface *SourceTransform*, provides the base interface and logic from which concrete realizations derive. These realizations provide the necessary logic to inject a disharmony into both the data model and source code artifacts. The SourceTransform hierarchy forms the basis of the Command pattern implemented.

The base class, *AbstractSourceTransform*, provides common SourceTransform functionality. Many of these operations involve simple update operations maintaining consistency between source files and the data model. The *AbstractSourceTransform* also provides an association with an *InjectionContext*, to provide access to the components managing the injection process. Finally, the *AbstractSourceTransform* contains a set of *Condition*(s) used to validate the current state of the model permits the transform operator execution. Beyond these basic operations and components, the AbstractSourceTransform has two specializations.

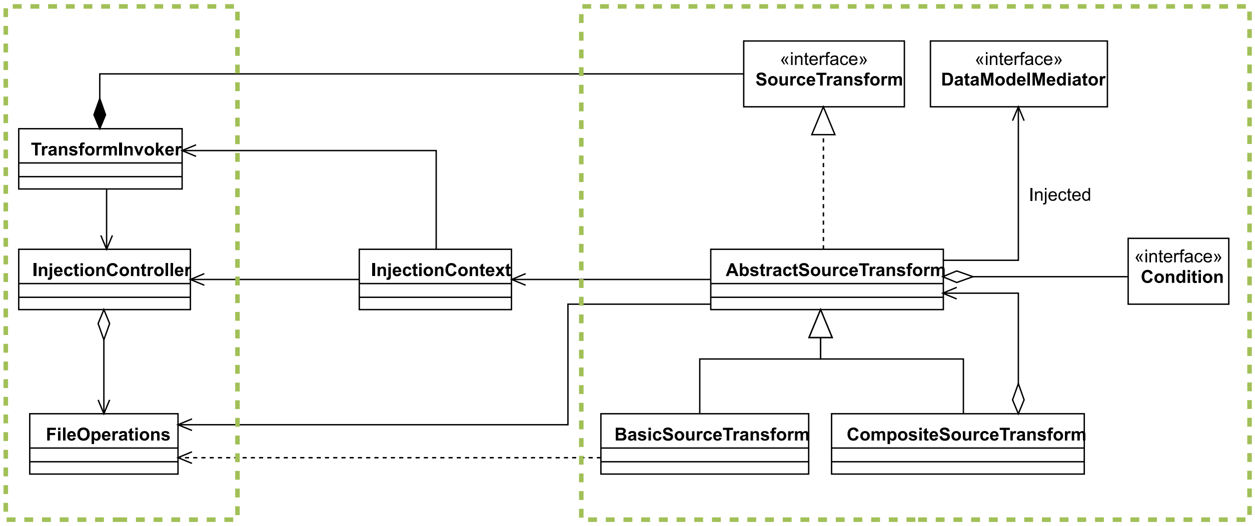


Figure 6.1: Software Injection meta-model.

These specializations include: (i) *BasicSourceTransform* and (ii) *CompositeSourceTransform*. The *BasicSourceTransform* is the base class for transforms, providing only basic source transformation operations. Transformation operations include, but are not limited to, the creation of a file, addition of a field, and addition of a constructor. This is in contrast to the *CompositeSourceTransform*. *CompositeSourceTransform* extensions provide both the base logic for generating code and the ability to divide their operation into combinations of other transforms. An example of such a transform is *CreateEncapsulatedField*, which creates a new Field and provides a getter and setter method for that field. The remaining classes in the meta-model define the operational components for the injection process.

The main component of the operational section of the meta-model is the *TransformInvoker* class. It provides the logic to control the correct application of transforms. As each *SourceTransform* is constructed it is stored until processing begins, via a concurrent queue, in the *TransformInvoker* controlled by the *InjectionController*. The *InjectionController* also controls, for each active file, a set of *FileOperations*. The *FileOperations* class provides the logic to handle SourceTransform required the file operations (includes basic file I/O and the

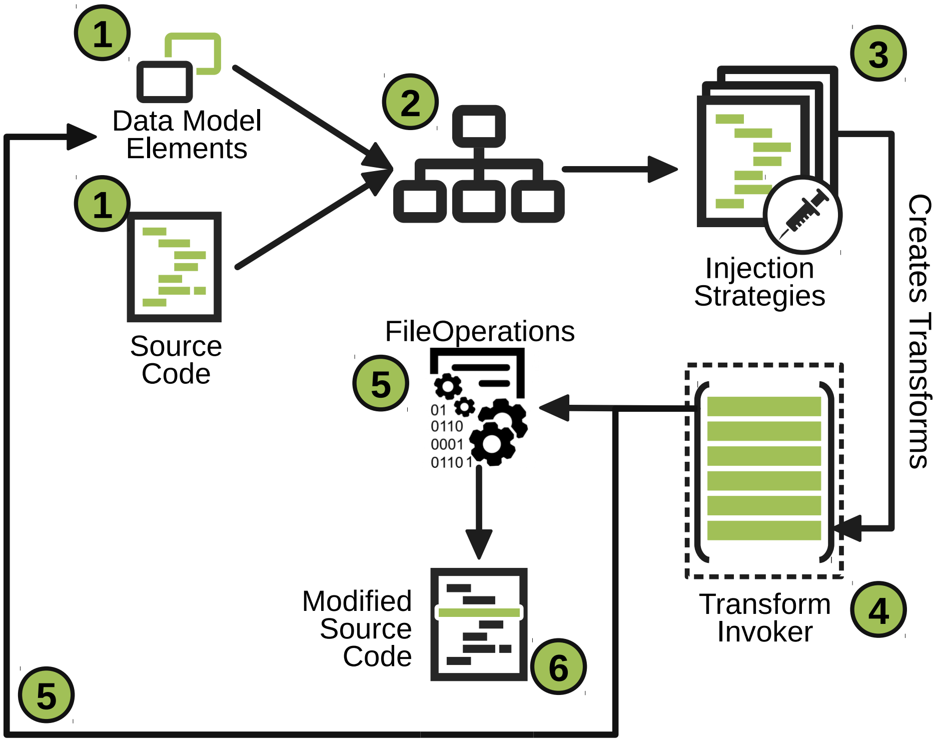


Figure 6.2: High-level overview of the software injection process.

injection of new file content). Figure 6.2 depicts the process of combining these components, which the following subsection describes.

6.2.2 The Injection Process

The Injection Process uses a combined model and direct source code manipulation approach, as opposed to a similar concept using bytecode manipulation developed by Dale [61], as depicted in Figure 6.2. This figure depicts the flow of execution, numbers encircled in green, as follows: 1.) Initially, the process extracts the data model components and their associated source code locations from the data model and file system. 2.) The data, extracted into a hierarchical model, via the InjectionController acts as input to the injection strategies. 3.) As each InjectionStrategy executes, it constructs SourceTransforms, which are passed to the TransformInvoker and added to its transform queue. 4.) Each queued transform executes to construct injection operators. 5.) These operators modify or construct data model elements, and then execute operations within the FileOperations entities, 6.) resulting in modified source code.

The novelty of the process described here is in the introduction of artifacts such as code smells, antipatterns, design pattern grime, and even design patterns using defined and validated injection strategies that control the injection process. This process modifies source code using a model-driven approach independent of any language-specific features. Furthermore, the ability to generate source code escapes the problem of simulation, common to this type of approach. Finally, this approach facilitates the ability to inspect the generated code to validate the production of these entities, a process that bytecode injection prohibits. Key to our approach is the *SourceInjector*, as depicted in Figure 6.3

The *SourceInjector* encodes the logic necessary to generate the sequence of transforms needed to inject one or more types of disharmonies into one or more software components, what we term *Injection Strategies*. Currently, we have developed strategies for modular grime (c.f. 6.3.1), class grime (c.f. 6.3.2), organizational grime (c.f. 6.3.3), and a *NullInjector* for the case of an unknown type. These injectors' base class, *GrimeInjector*, contains the common operations for each of its subtypes. Subtypes of this class are generated and provided to the *InjectionController* via the *GrimeInjectorFactory*. Finally, the three interfaces *ClassGrimeTypes*, *OrgGrimeTypes*, and *ModularGrimeTypes* are used simply to hold constants representing the names of known subtypes of the respective major-type of grime.

6.2.3 Integration into the Arc Framework

The injection process, described in this Chapter is a necessary component of the experimentation method defined in Chapter 8. This method of software injection, to be useful, must be integrated into the Arc Framework alongside the methods and tools we

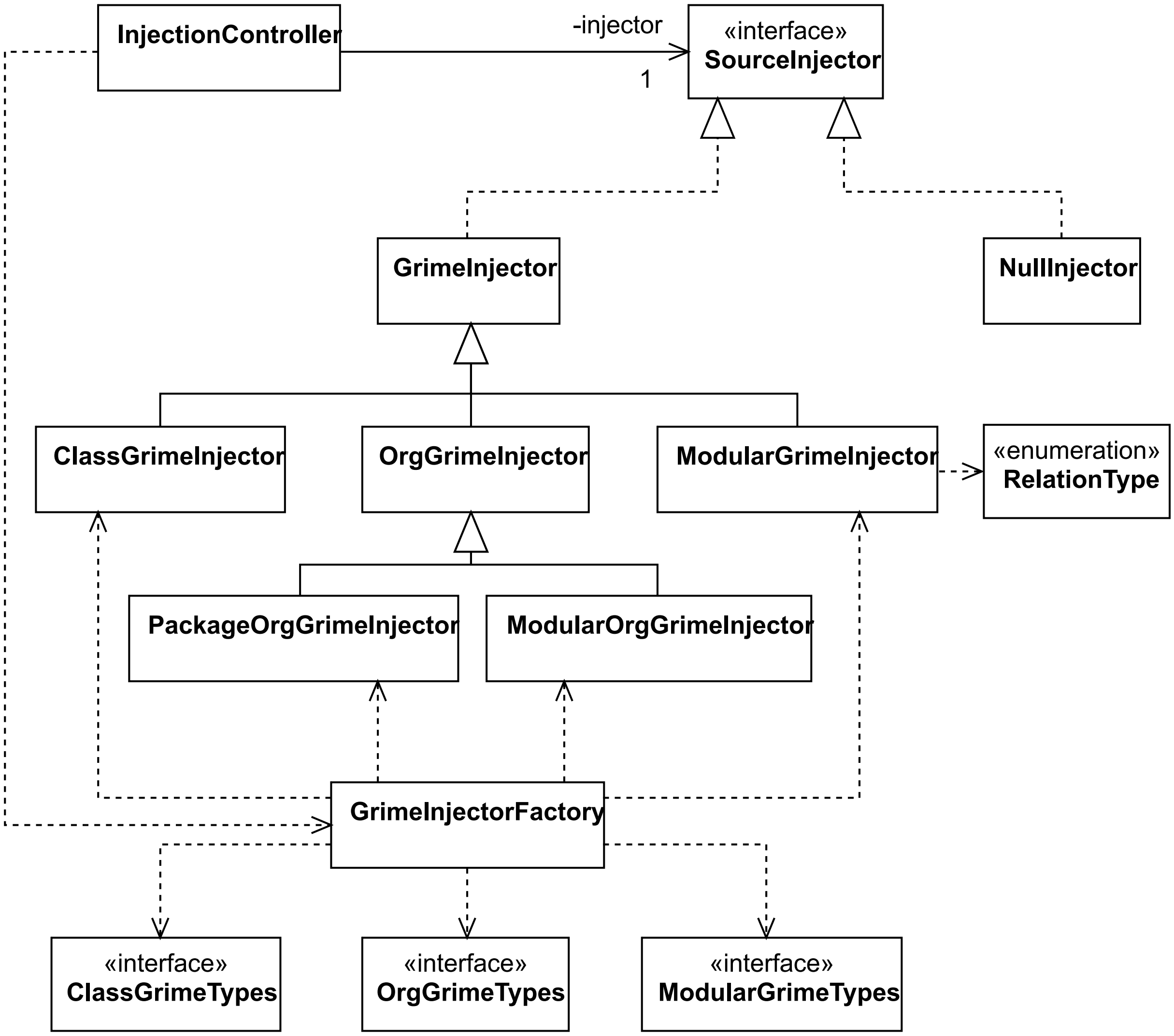


Figure 6.3: Software Injection Injectors.

described in Chapter 5. Figure 6.4 depicts the integration of Software Injection into the Arc framework.

Figure 6.4 shows the Software Injection execution path as the numbers encircled in green. This path is as follows: 1.) During system initialization the *SourceInjectorTool* is initialized to provide the Arc framework with the *InjectorCommand*. 2.) The *InjectorCommand* controls the execution of the *SourceInject* tool. The *InjectorCommand*

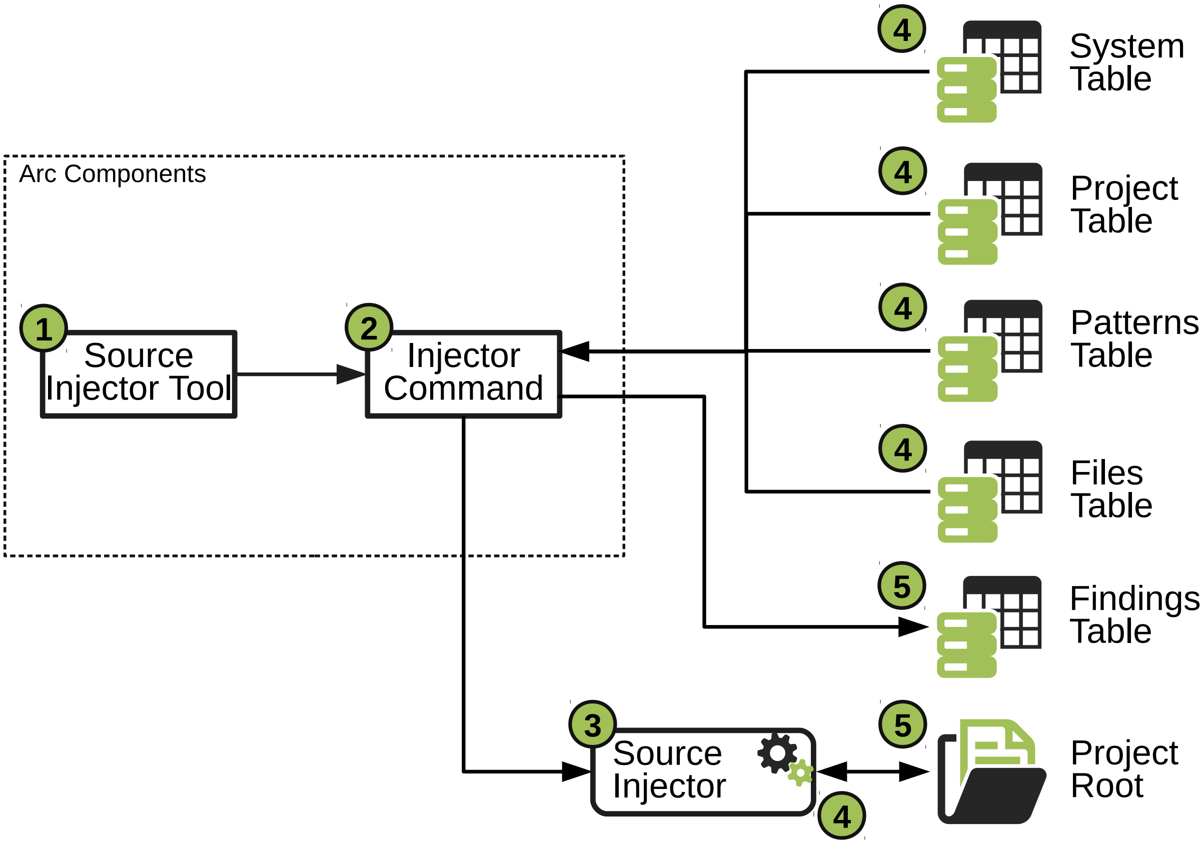


Figure 6.4: Source Injector integration with the Arc Framework.

also provides the *SourceInjector* with data from the data model 3.) The *SourceInjector* tool executes the source injection process according to selected detection strategies. 4.) The *SourceInjector*, via the *InjectorCommand*, extracts necessary data from the data model and reads in information from the source files in the project root. 5.) As the *SourceInjector* executes, it modifies both the physical files in the project root and the components in the data model. That is, the *InjectorCommand* adds basic artifacts that are created and added to the data model, along with, Findings for any added disharmonies.

Algorithm 6.1: Modular Grime Injection Strategy

```
procedure Inject($persist$, $extern$, $efferent$)
    $src \leftarrow \varnothing$
    $dest \leftarrow \varnothing$
    $rel \leftarrow \varnothing$
    if $extern$ then
        if $efferent$ then
            $src \leftarrow$ SelectOrCreateExternClass()
            $dest \leftarrow$ SelectPatternClass()
        else
            $src \leftarrow$ SelectPatternClass()
            $dest \leftarrow$ SelectOrCreateExternClass()
        end if
    else
        $src, dest \leftarrow$ Select2PatternClasses()
    end if
    if $persist$ then
        $rel \leftarrow$ SelectPersistentRelationship()
    else
        $rel \leftarrow$ SelectTempRelationship()
    end if
    CreateRelationship($src$, $dest$, $rel$)
end procedure
```

6.3 Design Pattern Grime Injection

This section details the injection strategies defined for each of the three types of design pattern grime: Modular Grime, Class Grime, and Organizational Grime.

6.3.1 Modular Grime

This section describes the basic strategy for injecting modular grime into an existing software system. This strategy, defined by the pseudocode in Algorithm 6.1, has three control parameters. These parameters correspond to the components defining the modular grime taxonomy [234]). These components are the strength, scope, and direction of the grime and exist in the strategy as the Boolean flags *persist.*, *extern*, and $efferent$, respectively.

Table 6.1: Value table for the Modular Grime Injection Strategy parameters. T indicates $true$, F indicates $false$, and – indicates N/A

Grime	Parameters		
	$persist$	$extern$	$efferent$
PIG	T	F	–
TIG	F	F	–
PEEG	T	T	T
TEEG	F	T	T
PEAG	T	T	F
TEAG	F	T	F

Parameter combinations specify the known types of modular grime, as shown in Table 6.1. Specifically, this works by controlling the algorithm using the following interpretations of each variable's possible values. When $persist$ is true, this indicates a form of persistent modular grime, and, otherwise, indicates a form of temporary modular grime. When $extern$ is true, this indicates a form of external modular grime, and, otherwise, indicates a form of internal modular grime. When $efferent$ is true, this indicates a form of efferent modular grime, and, otherwise, indicates a form of afferent modular grime. With this understanding in mind, the following describes the inner workings of this strategy.

The injection strategy, based upon the definition of Modular Grime from Chapter 9, generates grime inducing relationships using a set of three variables. These variables are as follows: (i) src, the source side of the grime inducing relationship, (ii) $dest$, the destination side of the grime inducing relationship, and (iii) rel, the type of the grime inducing relationship. Initially, the strategy sets the values for src, $dest$, and rel to be null. The variable value selection corresponds to the grime type specified by the input parameters. The following describes the assignment process for each of the variables.

Modular Grime injection variable assignment considers the following cases. If both *extern* and $efferent$ are true, then the value of the *src* will be the selection of some class external to the pattern instance, and *dest* selects from the set of pattern class. If *extern* is true and $efferent$ is false, the opposite selection occurs. If *extern* is not true, both *src* and *dest* are assigned pattern classes.

Once both *src* and *dest* are assigned, the algorithm selects the relationship type. The strategy determines relationship type, as follows. If $persist$ is true, then the value of *rel* is set to be persistent (one of generalization, realization, or a form of association), and otherwise is set to be temporary (a form of dependency). Finally, once selected, the variables *src*, *dest*, and *rel* act as input to the respective relationship constructing transform, which is created and passed to the *TransformInvoker*.

6.3.2 Class Grime

This section details the basic strategy for injecting class grime into an existing software system. This strategy, defined by the pseudocode in Algorithm 6.2, has three parameters. These parameters correspond to the components defining the class grime taxonomy. These components are the strength, the scope, and the direction components of the taxonomy (c.f. 9.5) and exist in the strategy as the Boolean flags *direct*, *internal*, and *pair*, respectively.

Parameter combinations specify the known types of class grime, as shown in Table 6.2. Specifically, this works by controlling the algorithm using the following interpretations of each variable's possible values. When *direct* is true, this indicates a form of direct class grime, and, otherwise, indicates a form of indirect class grime. When *internal* is true, this indicates a form of internal class grime, and, otherwise, indicates a form of external class grime. When *pair* is true, this indicates a form of pair class grime, and, otherwise, indicates a form of singular class grime. With this understanding in mind, the following describes the inner workings of this strategy.

Algorithm 6.2: Class Grime Injection Strategy

```
procedure Inject(direct, internal, pair)
    clazz ← SelectPatternClass()
    field ← SelectField(clazz)
    method1 ← $\varnothing$
    method2 ← $\varnothing$
    if internal then
        method1 ← SelectPatternMethod()
    else
        method1 ← SelectOrCreateMethod()
    end if
    if pair then
        method2 ← SelectOrCreateMethod()
    end if
    if direct then
        AddFieldUse(method1, field)
        if pair then
            AddFieldUse(method2, field)
        end if
    else
        mutator ← SelectOrCreateMutator(field)
        AddMethodCall(method1, mutator)
        if pair then
            AddMethodCall(method2, mutator)
        end if
    end if
end procedure
```

The injection strategy, based upon the definition of Class Grime from Chapter 9, generates grime inducing field and method use relationships using a set of four variables. These variables are as follows: (i) *clazz*, the class injected with grime, (ii) *field*, the field to be connected to a method forming the grime relationship, (iii) *method*1, and (iv) *method*2, the methods which will form the source side of the relationship. Initially, the strategy assigns to *clazz* the value of a randomly selected pattern instance class and assigns to *field* a randomly selected field from within that class. In the case that no available field exists, the strategy creates one. The strategy then sets the value of *method*1 and *method*2 to null.

Table 6.2: Value table for the Class Grime Injection Strategy parameters. T indicates *true* and F indicates *false*.

Grime	Parameters		
	direct	*internal*	*pair*
DIPG	T	T	T
DISG	T	T	F
DEPG	T	F	T
DESG	T	F	F
IIPG	F	T	T
IISG	F	T	F
IEPG	F	F	T
IESG	F	F	F

The variable value selection corresponds to the grime type specified by the input parameters. The following describes the assignment process for each of these variables.

Class Grime injection variable assignment considers the following cases. If *internal* is true, then *method*1 is set to a method within the selected class matching a pattern feature role. Otherwise, the strategy assigns *method*1 to any other method or a newly created method. If *pair* is true, then the strategy assigns *method*2 to any method or a newly created method. If *direct* is true, then the strategy constructs a transform to create a field use relationship between the *field* and *method*1 and adds it to the TransformInvoker. If both *direct* and *pair* are true, then the strategy constructs a transform to create a field use relationship between *field* and *method*2 and adds it to the TransformInvoker. If *direct* is false, then the strategy constructs an indirect relationship through a selected or created mutator of the *field*, rather than constructing direct relationships.

Algorithm 6.3: Package Organizational Grime Injection Strategy

```
procedure Inject($internal$, $closure$)
    $pkg \leftarrow$ SelectPatternNamespace()
    $type \leftarrow \varnothing$, $rel \leftarrow \varnothing$, $dest \leftarrow \varnothing$, $other \leftarrow \varnothing$
    if $internal$ then
        $type \leftarrow$ SelectPatternClass($pkg$)
    else
        $type \leftarrow$ selectOrCreateExternalClass($pkg$)
    end if
    $rel \leftarrow$ SelectRelationship()
    if $closure$ then
        $other \leftarrow$ SelectUnreachableNamespace($pkg$)
        $dest \leftarrow$ SelectExternalClass(other)
    else
        $other \leftarrow$ SelectNamespace()
        $dest \leftarrow$ SelectExternalClass(other)
    end if
    if $other$ and $dest$ then
        CreateRelationship($type$, $dest$, $rel$)
    end if
end procedure
```

6.3.3 Organizational Grime

This section details the basic strategies for injecting organizational grime into an existing software system. Unlike both Class and Modular Grime, Organizational Grime subdivides into two distinct types: Package and Modular Organizational Grime.

6.3.3.1 Package Organizational Grime This section details the strategy for injecting package organizational grime into an existing software system. This strategy, defined by the pseudocode in Algorithm 6.3, has two parameters corresponding the components of the taxonomy (c.f. 9.6) and are identified as the Boolean flags *internal* and *closure*.

Parameter combinations specify the known types of package organizational grime, as shown in Table 6.3. Specifically, this works by controlling the algorithm using the following interpretations of each variable's possible values. When *intern* is true, then this indicates

Table 6.3: Value table for the Package Organizational Grime Injection Strategy parameters. T indicates *true* and F indicates *false*.

Grime	Parameters	
	internal	*closure*
PICG	T	T
PIRG	T	F
PECG	F	T
PERG	F	F

a form of internal package organizational grime, and when false, this indicates a form of external package organizational grime. When *closure* is true, this indicates a form of closure package organizational grime, and, otherwise, indicates a form of reuse package organizational grime. The following describes the inner workings of this strategy.

The injection strategy uses the definition of Package Organizational Grime from Chapter 9. This strategy focuses on the creation of grime inducing relationships using the following five control variables: (i) *pkg*, the namespace containing elements of the pattern instance, (ii) *type*, the type selected from the pattern instance within the namespace referenced by *pkg*, (iii) *rel*, the relationship type to be injected, (iv) *dest*, the type representing the destination end of the relationship injected, and (v) *other*, the other namespace containing the type referenced by *dest*. Initially, the strategy assigns *pkg* to a randomly selected namespace from those containing elements of the pattern instance and the remaining four variables to null. The control variable values correspond to the values of the three input parameters.

This correspondence is a direct connection between the values specified in Table 6.3 to the definition of Package Organizational Grime; the strategy considers the following cases. If *internal* is true, then the strategy assigns *type* to a pattern instance class within the namespace referenced by *pkg*. Otherwise, the strategy assigns *type* to a class internal to the namespace or one created internal to the namespace but external to the pattern instance. The strategy then selects the type of relationship to create and assigns it to the variable *rel*. The strategy then selects the destination end of the relationship. If *closure* is true, then the strategy assigns to *other* a namespace currently unreachable from the namespace referenced by *pkg* and assigns to *dest* a type (external to the current pattern instance) found within the namespace referenced by *other*. Otherwise, the strategy assigns to *other* any namespace and assigns to *dest* a type (external to the current pattern instance) found within the namespace referenced by *other*. Finally, if both *other* and *dest* have a value set, the strategy constructs a transform to create a relationship between *type* and *dest* with the type or *rel* and adds it to the TransformInvoker.

6.3.3.2 Modular Organizational Grime This section details the strategy for injecting modular organizational grime into an existing software system. This strategy, defined by the pseudocode in Algorithm 6.4, has in three parameters corresponding to the components of the taxonomy (c.f. 9.6) and identified by the Boolean flags *persistent*, *internal*, and *cyclical*, respectively.

Parameter combinations specify the known types of modular organizational grime, as shown in Table 6.4. Specifically, this works by controlling the algorithm using the following interpretations of each variable's possible values. When *persist* is true, this indicates a form of persistent modular organizational grime, and, otherwise, indicates a form of temporary organizational modular grime. When *internal* is true, this indicates a form of internal modular organizational grime, and, otherwise, indicates a form of external modular organizational grime. When *cyclical* is true, this indicates a form of cyclical modular organizational grime, and, otherwise, indicates a form of unstable modular organizational grime. With this understanding in mind, the following describes the inner workings of this

Algorithm 6.4: Modular Organizational Grime Injection Strategy

```
procedure Inject($persistent$, $internal$, $cyclical$)
    $pkgs \leftarrow$ PatternNamespaces()
    $ns_1 \leftarrow \varnothing, ns_2 \leftarrow \varnothing$
    if internal then
        if $|pkgs| > 1$ then
            $(ns_1, ns_2) \leftarrow$ SelectNamespaces($pkgs$)
        else
            $ns_1 \leftarrow$ SelectNamespace($pkgs$)
            $(ns_1, ns_2) \leftarrow$ SplitNamespace(ns1)
        end if
    else
        $ns_1 \leftarrow$ SelectPatternNamespace()
        $ns_2 \leftarrow$ SelectOrCreateExternNamespace()
    end if
    if $persistent$ then
        $rel \leftarrow$ SelectPersistentRelationship()
    else
        $rel \leftarrow$ SelectTempRelationship()
    end if
    if $cyclical$ then
        CreateCyclicalDep($ns_1$, $ns_2$, $rel$)
    else
        AddInstabilitiy($ns_1$, $ns_2$, $rel$)
    end if
end procedure
```

strategy.

The injection strategy, based upon the definition of Modular Organizational Grime from Chapter 9, generates grime inducing relationships using a set of three variables. These variables are as follows: (i) *pkgs*, namespaces containing elements of the pattern instance, (ii) ns_1, the source-side namespace of an injected grime forming dependency, (iii) ns_2, destination side namespace of an injected grime forming dependency. Initially, the strategy assigns to *pkgs* the set of namespaces containing the pattern instance's elements. The variable selection corresponds to the grime type specified by the input parameters. The following describes the assignment process for each of these variables.

Table 6.4: Value table for the Modular Organizational Grime Injection Strategy parameters. T indicates *true* and F indicates *false*.

Grime	Parameters		
	persistent	*internal*	*cyclical*
MPICG	T	T	T
MPIUG	T	T	F
MPECG	T	F	T
MPEUG	T	F	F
MTICG	F	T	T
MTIUG	F	T	F
MTECG	F	F	T
MTEUG	F	F	F

Modular Organizational Grime injection variable assignment considers the following cases. If *internal* is true and the size of *pkgs* is greater than 1, then the strategy assigns both ns_1 and ns_2 to randomly selected members of *pkgs*. In the case that the size of *pkgs* is 1, then the strategy assigns both ns_1 and ns_2 to the namespaces created through a split of the namespace currently referenced by ns_1. Otherwise, if *internal* is false, then the strategy assigns to ns_1 any one of the pattern instance containing namespaces, and to ns_2 an external namespace if one exists (otherwise it creates one). If *persistent* is true, then the strategy assigns to *rel* a persistent relationship type (one of generalization, realization, or a form of association). Otherwise, the strategy assigns to *rel* a dependency relationship type. Finally, if *cyclical* is true, then a set of transforms are added to the TransformInvoker which will create relationships between ns_1 and ns_2 of type *rel*. These transforms result in the formation of a cyclical relationship between the namespaces represented by ns_1 and ns_2. On the other hand, if *cyclical* is false, then the set of transforms are added to the TransformInvoker create relationships between ns_1 and ns_2 of type rel, such that the instability of ns_1 increases.

6.4 Applications

This Chapter describes a technique to inject design disharmonies into existing source code programmatically. Within this research, this approach facilitates the evaluation of the effects of design pattern grime through experimentation. Although this technique is effective in accomplishing its objective, it is capable of much more. In the following subsections, we describe three potential applications of software injection.

6.4.1 Application to Experimentation

The use of injection allows for the controlled creation of design disharmonies through the creation of injection strategies. Furthermore, injection strategies allow the injection of any number or type of software entity into software artifacts. This capability, when combined with proper experimental design and parameterization, provides the ability to evaluate the various effects of the injected entities on the system. This combination enhances experimentation capability, as injection allows for randomization in assignment and selection of treatment groups, thus providing a means to evaluate causal relationships. Beyond traditional experimentation, software injection applies to case studies and single-case mechanism experiments from Design Science [275].

6.4.2 Application to Benchmarking

Injection strategies also allow the development of proper benchmarking datasets such as code smell, antipattern, and design rule violation detectors. Beyond just a simple true/false identification, injection strategies will allow for the fine-tuning and calibration of such tools. Fine-tuning and calibration allow for the possibility of identifying a range of detection for each design disharmony a tool supports. Improving tools in this manner allows for the identification and evaluation of the systematic error associated with these tools. Additionally, injection strategies can provide tools with the capability to detect rare or theoretical design disharmonies.

6.4.3 Application to Design Patterns

Beyond simply injecting disharmonies, injection strategies can inject more complicated and more useful concepts as well. For instance, as a part of the experimentation process, rather than requiring the software product studied to have existing instances of each type of design pattern, we could inject the required patterns into the system as needed. These injected design pattern instances provide a base for additional disharmony injection or other experiments.

6.5 Conclusion

This Chapter presented current work on the software injection process and framework. We presented the architecture, meta-model, and integration with the Arc framework underlying this approach. Furthermore, we defined the concept of *injection strategy*, which provides an algorithmic approach to inserting design disharmonies into the software. Finally, we presented several potential applications for this technique, which our future work will explore.

CHAPTER SEVEN

DESIGN PATTERN GRIME DETECTION

Up to a point, it is better to just let the snags [bugs] be there than to spend such time in design that there are none.

–Alan M. Turing

7.1 Introduction

Design pattern grime research, up to now, has been stagnated due to a limitation of manual identification processes. This has further limited our ability to understand the effects of grime within the context of software systems, and has become a significant issue worthy of study in and of itself. Similarly, research into other design disharmonies, such as code smells and antipatterns, has also suffered a similar problem. For these disharmonies the problem of automated detection has been addressed through a variety of methods [82, 83, 97, 110, 111, 113, 135, 139–142, 144, 165, 166, 171, 172, 179, 189, 193, 194, 196–198, 212, 217, 229, 232, 260, 262, 265, 266, 272, 277]. This chapter details efforts to address this problem for design pattern grime through the adoption of techniques known to work for other design disharmonies.

These approaches use the properties and measures of the components of a system understudy to effectively construct a hierarchy of filters used to identify artifacts afflicted with design disharmonies. Our approach implements a similar approach to facilitate the detection of grime.

This chapter is organized as follows. Section 7.2 describes how the automated detection of different forms of grime. Section 7.3 describes the integration of the detection system into the Arc Framework. Finally, Section 7.4 concludes this chapter.

7.2 Detection Framework

Automated grime detection strategies are directly based upon the definitions defined in Chapter 9. Using these taxonmies we have created a detection strategy per major type of grime. Each of these strategies filters software artifacts to identify the occurrence of a specific subtype of grime. The following subsections describe the detections starting with Modular Grime.

7.2.1 Modular Grime Detection

The detection strategy for Modular Grime is depicted in Algorithm 7.1. This strategy takes in a single parameter, *pattern*, representing a design pattern instance. This strategy has two basic phases. The first (lines 2 – 7) gathers the necessary information needed to determine if grime exists. The second is the actual detection phase (lines 8 – 37).

Phase 1: Data Gathering Phase The first major step in the data gathering phase is constructing the underlying graph using the call to `ConstructGraph` (line 3). This step creates a directed graph where the nodes are types comprising the pattern instance and any other type in the system directly coupled to the pattern instance types.

Next, the function `markInternalOrExternal` (line 4) considers each type (node) in the graph to determine whether it is internal or external to the pattern instance. Again, this is a trivial process that determines if the particular type is part of a RoleBinding in the provided pattern instance or not.

Next, the function `martTemporaryOrPersistent` (line 5) checks all of the edges within the graph. This is a trivial step as each edge maintains the type of coupling it represents. Furthermore, the graph should allow for both self-loops (a node has a connection to itself) and the ability to have type information for the edge to allow multiple edges between

Algorithm 7.1: Modular Grime Detection Strategy

```
procedure Detect($pattern$)
    $findings \leftarrow \varnothing$
    $graph \leftarrow$ ConstructGraph($pattern$)
    markInternalOrExternal($graph$,$pattern$)
    markTemporaryOrPersistent($graph$)
    markValidOrInvalid($graph$,$pattern$)
    calculateDeltas($graph$)
    for all $e \in edges(graph)$ do
        if $invalid(e)$ then
            $src \leftarrow source(e)$
            $dest \leftarrow dest(e)$
            if $internal(src) \wedge internal(dest)$ then
                if $persistent(e)$ then
                    $findings \leftarrow$ "PIG"
                else
                    $findings \leftarrow$ "TIG"
                end if
            else if $internal(src) \vee internal(dest)$ then
                if $persistent(e)$ then
                    if $increases(e, `Ca')$ then
                        $findings \leftarrow$ "PEAG"
                    end if
                    if $increases(e, `Ce')$ then
                        $findings \leftarrow$ "PEEG"
                    end if
                else
                    if $increases(e, `Ca')$ then
                        $findings \leftarrow$ "TEAG"
                    end if
                    if $increases(e, `Ce')$ then
                        $findings \leftarrow$ "TEEG"
                    end if
                end if
            end if
        end if
    end for
    return $findings$
end procedure
```

nodes. At that point, determining whether an edge is *temporary* or *persistent* is a simple table lookup or similar query.

Next, the function `markValidOrInvalid` (line 6) evaluates each edge to determine if it is valid/invalid according to the RBML specification for the pattern. Again, this should be a trivial implementation based on a simple query to the RBML implementation. Each of these marking functions sets a property in the corresponding structure evaluated.

Finally, the constructed and marked graph is used to calculate the metrics necessary for evaluating modular grime. In this case, we are concerned with *Afferent Coupling (Ca)*, which is the count of the number of incoming couplings of a class, and *Efferent Coupling (Ce)*, which is a count of the number of out-going couplings of a class [183]. The `calculateDeltas` function (line 7) calculates both metrics for each type when considering the exclusion of each coupling in which the class takes part. The information gathered in this phase is used in the detection phase to identify any occurrences of grime.

Phase 2: Detection Phase This phase iterates across all edges in the graph and evaluates the properties extracted during the data gathering phase. Each property gathered is evaluated using the predicates found within lines 9 – 30. The predicate $invalid(e)$ returns true if the edge was marked as invalid and false otherwise. The predicate $internal(n)$ returns true if the type n was marked as internal to the pattern and false otherwise. The predicate $persistent(e)$ returns true if the edge e represents a persistent type of coupling between the types. Finally, the predicate $increases(e, str)$ returns true if the provided edge creates a positive change to the named metric. Additionally, we can extract the source or destination types from the directed edge using the $source(e)$ and $dest(e)$ functions, respectively. If an edge meets the criteria indicating grime, that type of grime is added to the $finding$ list. After all edges in the graph are processed, this list is returned from the detection strategy.

Algorithm 7.2: Class Grime Detection Strategy

```
 1: procedure Detect($pattern$)
 2:     $findings \leftarrow \varnothing$
 3:     for all $t \in types(pattern)$ do
 4:         $graph \leftarrow$ constructGraph($t$)
 5:         markMethods($graph$,$pattern$)
 6:         $methodPairs \leftarrow$ markMethodPairs($graph$)
 7:         calculateDeltas($graph$)
 8:         DetectPairGrime($methodPairs$, $findings$)
 9:         DetectSingularGrime($graph$, $findings$)
10:     end for
11:     return $findings$
12: end procedure
```

7.2.2 Class Grime Detection

The detection strategy for Class Grime is depicted in Algorithms 7.2 – 7.4. Similar to the Modular Grime detection strategy, this strategy uses a single parameter, *pattern*, representing a design pattern instance. This strategy is composed of three phases. The first, similar to Modular Grime, is the Data Gathering phase and is defined in Algorithm 7.2. The second phase handles the detection of the method pair forms of Class Grime and is described in Algorithm 7.3. Finally, the third phase handles the detection of the singular method forms of Class Grime and is described in Algorithm 7.4. The following describes each phase in detail.

Phase 1: Class Grime Data Gathering The Class Grime detection process executes across each type bound to a role within a pattern instance. Thus, line 3 of Algorithm 7.2 starts by iterating across all types of the pattern. Phase 1 consists of lines 4 - 7 in Algorithm 7.2.

The first step of this phase is the construction of a method-attribute graph. The graph is a directed graph created by the `constructGraph` (line 4) call. This function creates a graph in which methods and attributes of the provided type are the nodes. The edges represent uses from methods to attributes or attribute accessor/mutator methods.

Next, the `markMethods` (line 5) call marks each method as either internal or external to the pattern instance. Performing this marking is trivial, as an internal method is bound to a role in the pattern instance. Additionally, this call also marks for which the method is the source as direct or indirect. A direct edge is between a method and an attribute, and an indirect edge is between a method and an accessor/mutator of an attribute.

Next, the `markMethodPairs` (line 6) call identifies each pair of methods accessing the same attribute as either direct or indirect. A pair is only identified as direct if both methods for a given attribute are individually direct. Otherwise, the pair is marked as indirect. Thus, this call results in a 4-tuple consisting of a method pair, an attribute, and a boolean value indicating direct or indirect.

Finally, the `calculateDeltas` (line 7) call calculates the change in two cohesion metrics. The first metric, Tight Class Coupling (TCC) [30], measures the cohesion for each method pair and a given attribute. The second metric, Ratio of Cohesive Interactions (RCI) [40], considers the cohesion based on how individual methods use attributes. Both metrics handle the cases of direct and indirect attribute use. The information gathered in this phase is used in the following detection phases to identify any occurrences of Class Grime.

Phase 2: Class Grime Detection for Method Pairs Class Grime Detection Phase 2 is defined by Algorithm 7.3. This phase works similarly to the Modular Grime Detection Phase. In this phase, method pair tuples and a list of findings are provided. Each tuple provides the necessary data to identify Class Grime occurrences attributable to method pairs. Just as in the Modular Grime Detection Strategy, several predicates provide the filtering mechanism providing this identification. In this phase, each tuple is evaluated, extracting out the method pairs as m_1 and m_2 along with determining whether the pair is *direct* or not. The predicate

Algorithm 7.3: Class Grime Detection Strategy - Pair Types

```
function DetectPairGrime($tuples$, $findings$)
    for all $tuple \in tuples$ do
        $m_1 \leftarrow tuple[1]$
        $m_2 \leftarrow tuple[2]$
        $a \leftarrow tuple[3]$
        $direct \leftarrow tuple[4]$
        if $direct$ then
            if $(internal(m_1) \wedge internal(m_2)) \wedge decreases(m_1, m_2, a, `TCC') \wedge (calls(m_1) = \varnothing \vee calls(m_2) = \varnothing)$ then
                $findings \leftarrow$ "IIPG"
            else if $(\neg internal(m_1) \vee \neg internal(m_2)) \wedge decreases(m_1, m_2, a, `TCC') \wedge (calls(m_1) = \varnothing \vee calls(m_2) = \varnothing)$ then
                $findings \leftarrow$ "IEPG"
            end if
        else
            if $(internal(m_1) \wedge internal(m_2)) \wedge decreases(m_1, m_2, a, `TCC') \wedge (calls(m_1) = \varnothing \vee calls(m_2) = \varnothing)$ then
                $findings \leftarrow$ "DIPG"
            else if $(\neg internal(m_1) \vee \neg internal(m_2)) \wedge decreases(m_1, m_2, a, `TCC') \wedge (calls(m_1) = \varnothing \vee calls(m_2) = \varnothing)$ then
                $findings \leftarrow$ "DEPG"
            end if
        end if
    end for
end function
```

$internal(m)$ is true when the provided method is bound to a role in the pattern instance other it is false. The $decreases(m_1, m_2, a, met)$ predicate is true if the method pair's use of an attribute, a, reduces the provided metric, met, compared to a base that excludes those uses. The $calls(m)$ function returns the set of methods calling the provide method. Using this knowledge, following the logic should be relatively straightforward. Each pair that meets any criteria will add new grime findings to the $finding$ list.

Phase 3: Class Grime Detection for Singular Methods Class Grime Detection Phase 3 is described by Algorithm 7.4. This phase works similarly to the last phase. In this phase, Algorithm 7.4: Class Grime Detection Strategy - Singular Types

```
function DetectSingularGrime($graph$, $findings$)
    for all $rel \in edges(graph)$ do
        $m \leftarrow source(rel)$
        if $direct(rel)$ then
            if $internal(m) \wedge decreases(rel`RCI') \wedge calls(m) = \varnothing$ then
                $findings \leftarrow$ "DISG"
            else if $\neg internal(m) \wedge decreases(rel, `RCI') \wedge calls(m) = \varnothing$ then
                $findings \leftarrow$ "DESG"
            end if
        else
            if $internal(m) \wedge decreases(t, m, `RCI') \wedge calls(m) = \varnothing$ then
                $findings \leftarrow$ "IISG"
            else if $\neg internal(m) \wedge decreases(t, m, `RCI') \wedge calls(m) = \varnothing$ then
                $findings \leftarrow$ "IESG"
            end if
        end if
    end for
end function
```

rather than tuples as input, this phase takes in the method-attribute graph. The graph provides the necessary data to identify Class Grime occurrences attributable to singular methods. The predicates used in this phase are the same as those in the last phase, with two exceptions. The $direct(r)$ evaluates to true if the provided relationship is marked direct and false otherwise. The $decreases(r, met)$ predicate evaluates to true if the provided relationship causes a decrease in the provided metric when included versus when excluded. Additionally, a new function, $source(r)$, extracts the source method of the relationship. In this phase, each relationship in the method-attribute graph is evaluated. Using the knowledge and properties extracted during Phase 1 the relationships and their source methods are filtered to detect any occurrences of grime. Those occurrences detected are then added as new findings to the $finding$ list.

Algorithm 7.5: Organizational Grime Detection Strategy

```
1: procedure DETECT($pattern$)
2:     $findings \leftarrow \varnothing$
3:     $(nsGraph, typeGraph) \leftarrow$ CONSTRUCTGRAPHS($pattern$)
4:     MARK($pattern$)
5:     CALCULATEDELTAS()
6:     DETECTMODULARORGGRIME($findings$,$nsGraph$)
7:     return $findings$
8: end procedure
```

7.2.3 Organizational Grime Detection

The detection strategy for Class Grime is depicted in Algorithms 7.2 – 7.4. Similar to the Modular Grime detection strategy, this strategy uses a single parameter, *pattern*, representing a design pattern instance. This strategy is composed of three phases. The first, similar to Modular Grime, is the Data Gathering phase and is defined in Algorithm 7.2. The second phase handles the detection of the method pair forms of Class Grime and is described in Algorithm 7.3. Finally, the third phase handles the detection of the singular method forms of Class Grime and is described in Algorithm 7.4. The following describes each phase in detail.

Phase 1: Oganizational Grime Data Gathering The Organizational Grime Data Gathering Phase is described in Algorithm 7.5. This process, similar to the Modular and Class Grime Data Gathering Phases, is provided a pattern instance. Phase 1 consists of lines 3 - 6 of Algorithm 7.2.

The first step of this phase is the construction of a namespace and type graphs. Both are directed graphs created by the `constructGraphs` (line 3) call. This function creates a type graph similar to the graph created in Modular Grime Phase 1. The namespace graph is a directed graph consisting of all the namespaces in the system connected by dependencies extracted from the couplings between classes in separate namespaces.

Algorithm 7.6: Organizational Grime Detection Strategy - Package Types

```
function DetectPackageOrgGrime($findings$, $typeGraph$)
    for all $n \in nodes(typeGraph)$ do
        if $\neg internal(n) \wedge nsInternal(n) \wedge decreases(n, `CohesionQ')$ then
            $findings \leftarrow$ “PECG”
        else if $internal(n) \wedge nsInternal(n) \wedge decreases(n, `CohesionQ')$ then
            $findings \leftarrow$ “PICG”
        else if $\neg internal(n) \wedge nsInternal(n) \wedge decreases(n, `CouplingQ')$ then
            $findings \leftarrow$ “PERG”
        else if $internal(n) \wedge nsInternal(n) \wedge decreases(n, `CouplingQ')$ then
            $findings \leftarrow$ “PIRG”
        end if
    end for
end function
```

Next, the `mark` (line 5) call marks all types bound to a role in the provided pattern instance as internal. This function also marks namespaces containing any such types as internal. Additionally, for internal namespaces, types contained by those namespaces are marked internal if not already.

Finally, the `calculateDeltas` (line 6) call calculates the change in the following metrics. The first metric, CohesionQ, measures the closure quality of a package [3]. The second metric, CouplingQ, measures the reuse quality of a package. The final metric, Normalized Distance (D’), measures the instability of a set of packages [183]. The information gathered in this phase is used in the following detection phases to identify any occurrences of Organizational Grime.

Phase 2: Organizational Grime Detection for Package Types The Organizational Grime Detection Phase 2 is defined by Algorithm 7.6. This phase works similarly to the Modular Grime Detection Phase. In this phase, the type graph and a list of findings are provided. Like the Modular Grime Detection Strategy, several predicates provide the filtering mechanism providing this Organizational Grime identification. In this phase, each type in the type graph is iterated across. Several predicates are then used to filter the types until any occurrences of Organizational Grime are identified. The predicates and functions used include the following. The predicate $internal(t)$ evaluates true when the provided type, t, is bound to a role in the pattern instance. The $nsInternal(t)$ predicate evaluates to true if the provided type, t, is contained within a namespace that has also been marked internal. Finally, the predicate $decreases(t, met)$ evaluates to true if the provided type, t, reduces the provided metric, met when comparing the evaluation of the metric when the type is excluded to when it is included. Using this knowledge, following the logic should be relatively straightforward. Each type that meets the criteria for any subtype of Organizational Grime defined in Algorithm 7.6 will result in an addition of new grime findings to the $finding$ list.

Phase 3: Organizational Grime Detection for Modular Types Organizational Grime Detection Phase 3 is described by Algorithm 7.7. This phase works similarly to the last phase. In this phase, rather than tuples as input, this phase takes in the namespace graph. This graph provides the necessary data to identify modular type Organizational Grime occurrences. As in other detection strategies, a series of predicates and functions are used to filter the data and identify a set of grime findings. The $source(r)$ function extracts the source namespace from a namespace dependency of the namespace graph. The $dest(r)$ function operates similarly to $source(r)$ but extracts the destination side of the relationship. The persistent(r) predicate evaluates to true if the relationship, r, is marked persistent. The $iternal(n)$ predicate evaluates to true if the provided namespace, n contains types of bound to roles within the pattern instance. The $cycle(r)$ predicate evaluates true if the provided relationship is part of a cycle between namespaces. The $dropInstability(r)$ predicate evaluates to true if the provided relationship, r reduces the Instability metric for the system. In this phase, each relationship in the namespace graph, $nsGraph$, is evaluated. For each relationship within the graph, the source and destination namespaces are extracted.

Algorithm 7.7: Organizational Grime Detection Strategy - Modular Types

```
function DetectModularOrgGrime($findings$,$nsGraph$)
    for all $rel \in edges(nsGraph)$ do
        $src \leftarrow source(rel)$
        $dest \leftarrow dest(rel)$
        if $persistent(rel) \wedge ((internal(dest) \wedge \neg internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge cycle(rel))$ then
            $findings \leftarrow$ “MPECG”
        else if $\neg persistent(rel) \wedge ((internal(dest) \wedge \neg internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge cycle(rel))$ then
            $findings \leftarrow$ “MTECG”
        else if $persistent(rel) \wedge ((internal(dest) \wedge internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge cycle(rel))$ then
            $findings \leftarrow$ “MPICG”
        else if $\neg persistent(rel) \wedge ((internal(dest) \wedge internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge cycle(rel))$ then
            $findings \leftarrow$ “MTICG”
        else if $persistent(rel) \wedge ((internal(dest) \wedge \neg internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge dropInstability(rel))$ then
            $findings \leftarrow$ “MPEUG”
        else if $\neg persistent(rel) \wedge ((internal(dest) \wedge \neg internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge dropInstability(rel))$ then
            $findings \leftarrow$ “MTEUG”
        else if $persistent(rel) \wedge ((internal(dest) \wedge internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge dropInstability(rel))$ then
            $findings \leftarrow$ “MPIUG”
        else if $\neg persistent(rel) \wedge ((internal(dest) \wedge internal(src)) \vee (internal(src) \wedge \neg internal(dest)) \wedge dropInstability(rel))$ then
            $findings \leftarrow$ “MTIUG”
        end if
    end for
end function
```

This pair of namespaces and the associated relationship are combined with the knowledge and properties extracted during Phase 1 to filter the relationships and identify Organizational Grime. Those occurrences detected are then added as new findings to the $finding$ list.

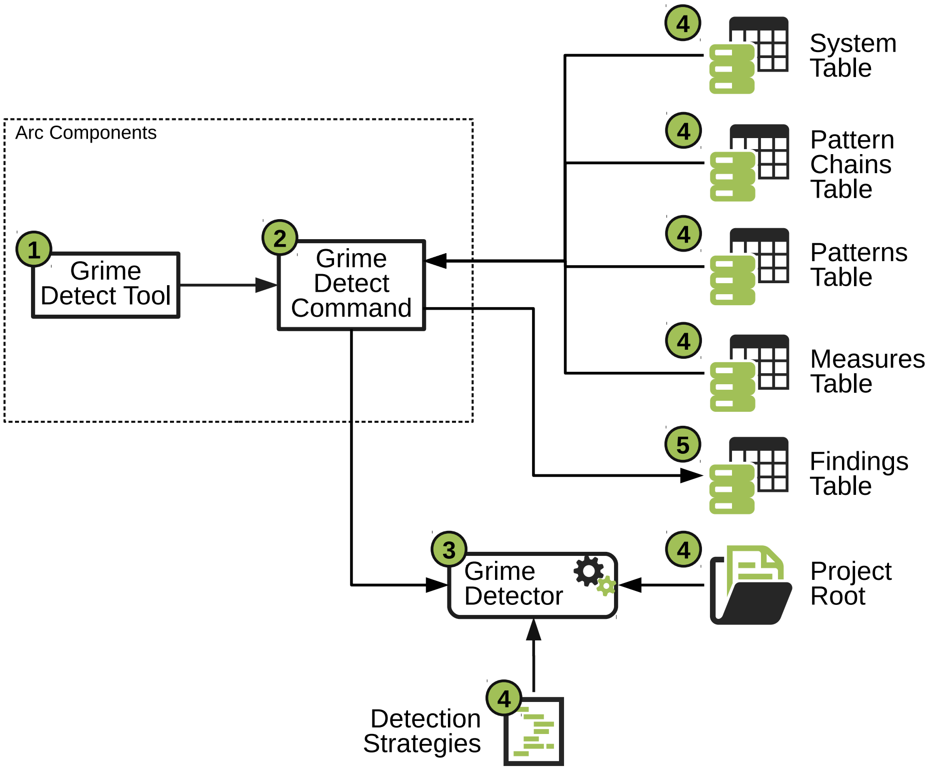


Figure 7.1: Grime Detection integration with the Arc Framework.

7.3 Arc Framework Integration

To make the detection strategy framework useful, it needs to be integrated into the Arc Framework, as depicted in Figure 7.1. The execution of this integration follows the execution path depicted by the numbers encircled in green, as follows: 1.) When the Arc System is initialized, it initializes the *GrimeDetectTool*, which provides the framework with and instance of the *GrimeDetectCommand*. 2.) The *GrimeDetectCommand* performs two functions: (i) it controls the operation of the Grime Detector and (ii) provides access to the ArcDb via the ArcContext. 3.) The *GrimeDetectCommand* executes the *GrimeDetector* which utilizes the detection strategy framework, described in Section 7.2. 4.) The *GrimeDetector* executes all three grime detection strategies across applicable pattern instances. 5.) As instances of design pattern grime are detected, each instance is provided to the *GrimeDetectCommand* to be encoded into Findings and added to the Findings Table of the ArcDb.

7.4 Conclusion

This chapter defined our approach for the automated detection of design pattern grime. This approach uses implementations of detection strategies [179] and Moha et al. [196] for each primary type of Design Pattern Grime. Each of these detection strategies was defined using the taxonomy definitions from Chapter 9. Additionally, we describe the integration of this system into the Arc experimentation framework making the validation study defined in Chapter 11 possible.

CHAPTER EIGHT

PUTTING IT ALL TOGETHER: THE METHOD

Truth can only be found in one place: the code.

–Robert C. Martin

In the field of software engineering, there is an incredible rate of change in the methods and tools used, leading to a call for more experimentation in software engineering [276]. Furthermore, this expediency of change has led to the proposal of an ever-increasing number of issues purporting to affect the quality of software. However, these claims are typically only supported by anecdotal data. The lack of empirical support underlying such claims leads to a need to gain more in-depth insight into the nature of these phenomena. To accomplish this, a process which provides the logical and scientific methods is required.

The development of such a generalized process would be too broad to be useful due to the numerous and profoundly different contexts found within the software engineering field. Thus, when considering the nature of design disharmonies and other software issues, we have opted to constrain this approach to the context of the software itself. Here we consider only the underlying source code, design documents, build files, and other artifacts that comprise a software system.

Supporting this process are the frameworks and techniques documented in Chapters 4–7. The remainder of this chapter is organized as follows: Section 8.1 describes the aspects of software engineering to study in both isolation and within live systems. Section 8.2 describes the general method and its application. Section 8.3 concludes with a summary and the implications of this method on further study of design disharmonies.

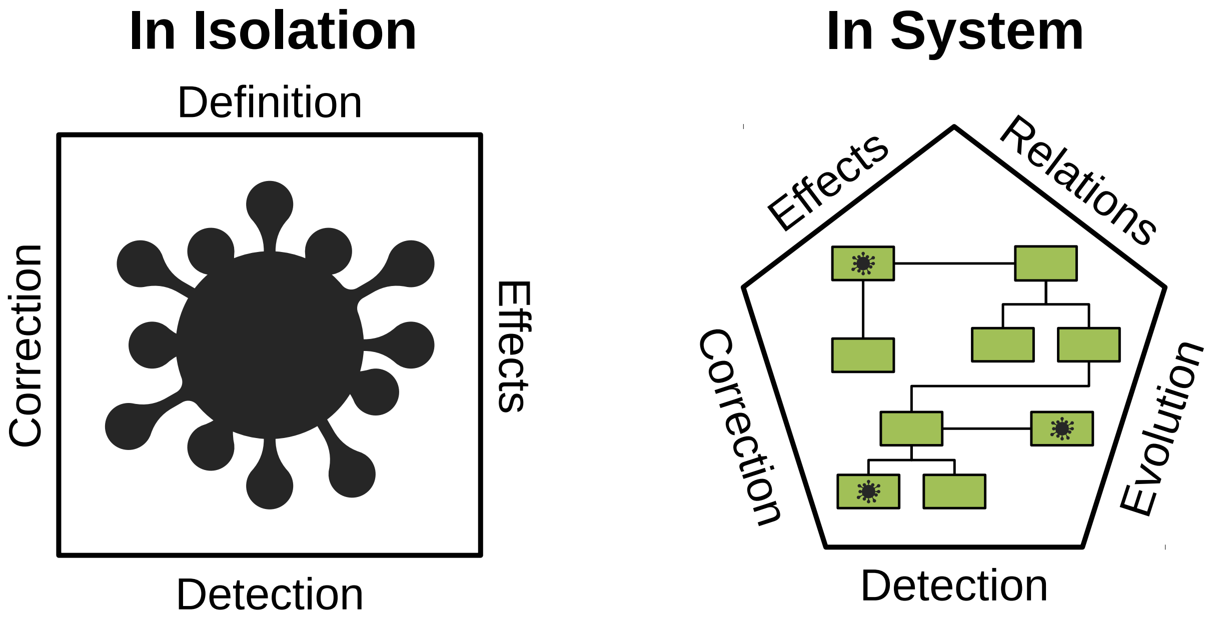


Figure 8.1: Software engineering phenomena aspects of study.

8.1 Aspects to Study

The goal of this process is to increase our understanding of software engineering phenomena. Before describing this method, we first describe the aspects of the phenomena we wish to study and the phenomena's observational context. We consider two main contexts of study, as depicted in Figure 8.1: (i) In Isolation (corresponding to *in vitro* studies) and (ii) In System (corresponding to *in vivo* studies). These contexts provide multiple lenses through which to study such software engineering phenomena and are analogous to the study of biological phenomena. Biological phenomena such as pathogens are studied both in isolation in larger biological systems such as the human body.

In the case of the phenomena in isolation, we are concerned with the following four aspects, as depicted in Figure 8.1: (i) Definition, (ii) Effects, (iii) Detection, and (iv) Correction. In the case of the phenomena in system, we are concerned with the following five aspects, as depicted in Figure 8.1: (i) Relations, (ii) Effects, (iii) Evolution, (iv) Correction and (iv) Detection. The goals of each of these aspects in their respective context are as follows:

- *Definition:* The goal is to explore the connections between the phenomena studied and the greater body of software engineering knowledge. This connection relates well-known concepts such as design principles, metrics, and quality within an overarching logically consistent framework for the phenomena. This framework results in the development of a community or research-driven taxonomy of the phenomena and its subtypes.

- *Effects:* The goal is the evaluation of the effects that a manifestation of the phenomena within software artifacts has on the developed product. Such effects may direct, such as changes to the structure or behavior, or indirect, such as changes to quality attributes.

  - **In Isolation:** The goal, in isolation, is the evaluation of the causal relationship between the manifestation of the phenomena and the change this manifestation has on system properties such as software product quality attributes.

  - **In System:** The goal, in system, is to observe the changes explored in isolation, over time, and to relate these observations to the effects of observed phenomena.

- *Detection:* The goal of detection is to develop techniques by which one may identify instances of the phenomena in an automated or semi-automated way.

  - **In Isolation:** The goal, in isolation, is to define initial detection capabilities based upon the theoretical framework, from the definitional aspect, in conjunction with knowledge of the effects of the phenomena.

  - **In System:** The goal, in system, is the improvement of the efficiency and accuracy of detection technique or the automated generation of improved detection strategies based on the evolving understanding of the phenomena under study.

- *Correction:* The goal is the identification of refactoring combinations or other techniques (from here on called *correction strategies*) which remediate instances of the phenomena and mitigate its effects on the overall software system.
  - **In Isolation:** The goal, in isolation, is the initial identification of correction strategies that will remediate instances of the phenomena at various levels of severity.
  - **In System:** The goal, in system, is the improvement of initial correction strategy efficiency and accuracy through automated generation/improvement techniques. Such techniques may utilize exiting correction strategies or generate new ones based on the evolving understanding of the phenomena under study.

- *Relations:* This aspect focuses on the exploration of relationships between subtypes of the phenomena and relationships to other types of software engineering phenomena. The goals, in system, are as follows: (i) to further develop the descriptive framework, (ii) to develop a notion of severity, and (iii) to understand relative priorities between phenomena.

- *Evolution:* This aspect focuses on how instances of the phenomena change over time. The goals, in system, are as follows: (i) To understand how software items are affected due to the accumulation of the phenomena over time and (ii) To use knowledge accumulation to develop an understanding of the susceptibility of the types of artifacts affected and of the rarity of such phenomena occurring.

8.2 The Process

Having described the aspects, we are concerned with, the following describes a general process to study software engineering phenomena. This approach guides this dissertation and is based on the methods and techniques from Empirical Software Engineering, as described by Wohlin et al. [276], Juristo and Moreno [136], and Runeson et al. [230].

We have developed a six-phase process forming the basic philosophy to guide the development of experiments and case studies to further our understanding of design disharmonies. This process, depicted in Figure 8.2, is divided into four sections: "*in vitro* Experimentation", "Bridge", "*in vivo* Case Studies", and "Bridge". Although *in vitro* and *in vivo* approaches are not new in empirical software engineering research, the definition of an approach that prescribes a method to bridge these two concepts is novel. We consider this last phase to be the one with the most impact on both researchers and practitioners. Nevertheless, we have yet to complete all the necessary studies to gather the data needed to begin the last two phases. Thus, we have grayed out both Phase 5 and Phase 6 Figure 8.2, as we leave this to future work. The following describes this process in detail.

Central to the application of this process is the utilization of the Arc Framework, as detailed in Chapter 3. Arc provides, as depicted in Figure 8.2, an internal experience database surrounded by the hexagonal barrier with meter icons, we call this the core. The core represents the data collection process for each phase, in which the results of executed workflows are collected and stored in the database. Automated data collection exists for each phase, though the entire process itself is not automated, which is appropriate, as each phase is composed of multiple studies, requiring a period of refinement between phases. This refinement allows for the use of prior phase results and the evolution of the process.

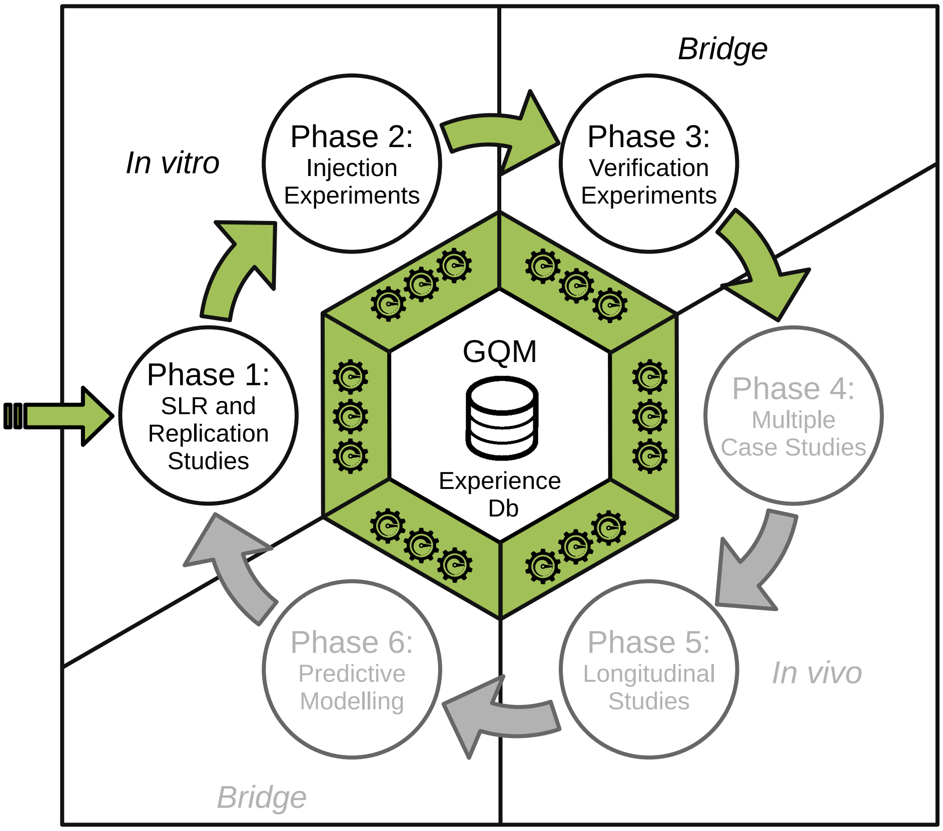


Figure 8.2: The methodological process for empirical research concerning software artifacts.

8.2.1 *In Vitro* Experimentation

The first section of this process is based on *in vitro* experimentation. Principles of experimental design (as described by Wohlin et al. [276] and Juristo and Moreno [136]) in concert with the results of prior studies and insight of the researchers guide the phases of this section. This section contains Phases 1 and 2, described in the following.

Phase 1: Meta-Studies and Experiential Studies. Phase 1 comprises the initial examination of the phenomena of concern, through a combination of several studies. The initial study focuses on a full literature review. Such reviews may be formal or informal meta-studies such as Systematic Literature Reviews (SLR) and mapping studies. These studies result in the development of an initial set of guiding questions and research goals and an initial theoretical framework.

The initial goals and questions, extracted from identified gaps in current knowledge, are refined using the GQM approach, central to the entire method. Following the GQM approach, researchers use the questions to develop a set of metrics (as indicated by the meter icons in Figure 8.2). Finally, the researchers store the results (literature review results and refined GQM) in the experience database (via the Arc Framework) for later analysis and consultation.

The development of theoretical frameworks, e.g., taxonomies or ontologies of the phenomena, should be developed based on existing knowledge of the phenomena and their relationship to existing software engineering knowledge, through a coherent, logical structure. A consistent structure is necessary if the framework is to be the basis for the development/evaluation of tools used to detect instances of the phenomena. Additionally, these theoretical frameworks are subject to validation by the research and practitioner communities.

Replication or pilot studies provide the necessary validation and experience with the theoretical models. These studies have the following goals: (i) to provide experience with current processes, tools, and methods, (ii) to identify issues such tools, processes, and methods, (iii) to validate prior research in new contexts, and (iv) to validate the theoretical frameworks derived from prior meta-studies. The following provides a detailed overview of the inner workings of this phase.

Figure 8.3 depicts the high-level overview of Phase 1. This phase has three basic steps, as indicated by the numbers encircled in green: 1.) Initially conduct a meta-study such as an SLR or Mapping Study to gather an understanding of the current research. The goal is the development of both a theoretical framework defining the phenomena and the development of an initial GQM hierarchy based on the current gaps in research. 2.) Refine the GQM

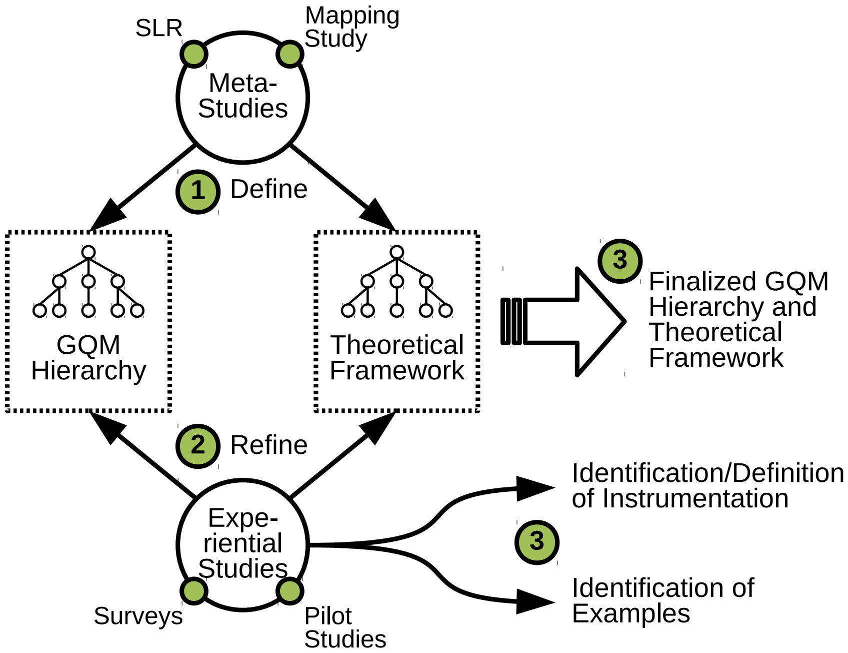

SLR
Mapping Study
Meta-Studies
1
Define
GQM Hierarchy
Theoretical Framework
3
Finalized GQM Hierarchy and Theoretical Framework
2
Refine
Experiential Studies
Surveys
Pilot Studies
Identification/Definition of Instrumentation
3
Identification of Examples

Figure 8.3: Phase 1 overview.

hierarchy through initial studies, such as community surveys (within both the industry and research communities) and pilot studies. 3.) The study results should lead to a finalized theoretical framework and GQM hierarchy. Furthermore, this phase results in an initial instrumentation and setup of methods used in subsequent phases. If the researchers elected to complete pilot studies, then exemplary instances of the phenomena should be collected for further examination. The remainder of this section details the two subsections of this phase.

Figure 8.4 depicts the meta-study process. This process follows the sequence indicated by the numbers encircled in green, as follows. 1.) There are two possible paths (though there are other study types and less formal approaches): (i) SLR and (ii) Mapping Studies. 2.) Both study types initiate a GQM process to formulate the goals, questions, and metrics, and defining the study GQM hierarchy. 3.) The constructed GQM hierarchy guides the studies per the guidelines of Wohlin et al. [276]. 4.) These guidelines inform the query protocols

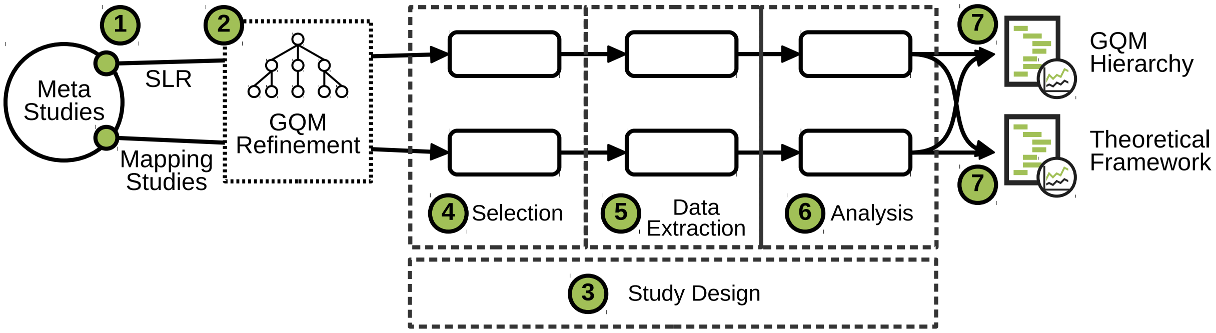


Figure 8.4: Phase 1 Meta-Studies details.

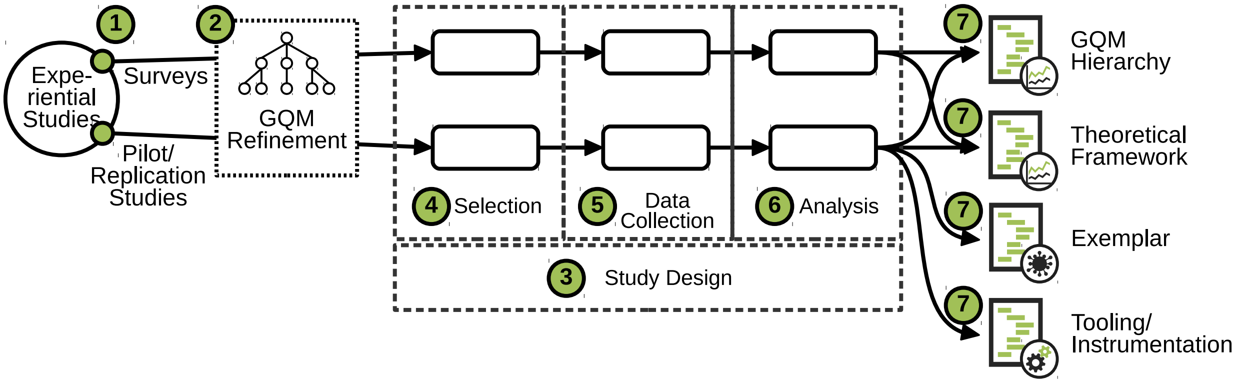


Figure 8.5: Phase 1 Experiencial Studies details.

defined for the collection of primary studies and the inclusion and exclusion criteria defined to filter the primary studies. Secondary studies are collected using the snowballing technique (tracing back from initial studies to identify other pertinent studies through references). The combined process of collecting primary and secondary studies continues until the set of new studies found is empty. 5.) Researchers refine the collected studies to a final set used during data analysis. 6.) The data analysis serves to answer the questions defined in the study's GQM hierarchy. 7.) The results of this analysis refine the synthesized theoretical framework and the initial GQM hierarchy for use in subsequent phases.

Figure 8.5 depicts the details of conducting the experiential studies. This process follows the sequence indicated by the numbers encircled in green, as shown: 1.) are the two paths previously mentioned: (i) Surveys and (ii) Pilot or Replication Studies. 2.) In both cases, the GQM process formulates the goals of the study into a GQM hierarchy. 3.) This hierarchy guides the study following the guidelines of Wohlin et al. [276]. 4.) For either study type, a randomization method selects the experimental subjects, and the experiment conducted. 5.) During execution, the instruments (such as the survey itself or metrics tools) collect data. 6.) The results of the data collection process are analyzed to answer the questions initially posed. 7.) The results are used in the following ways. First, the results refine the overarching GQM hierarchy and theoretical frameworks. Second, the data collected provides example instances of the phenomena for further study. Finally, the process of completing these studies yields the initial tooling/instrumentation needed to conduct further studies.

Phase 1 studies culminated in the development of both the background and literature review (found in Chapter 2) and in the development of the extended grime taxonomy (found in Chapter 9). Furthermore, we conducted a pilot study based on this foundational knowledge [101] to validate our initial Class Grime taxonomy. The combined results of these initial works are documented in the current grime taxonomy found in Chapter 9 and the GQM hierarchy found in Chapter 1 and further developed in Chapters 9–11.

Phase 2: Injection Experiments. This phase begins with the review and refinement of the GQM hierarchy developed as a result of Phase 1. This phase's goal is to develop a set of experiments, guided by the GQM hierarchy and based on the injection process, to study the effects of these phenomena. The experiments evaluate the effects on instances of the phenomena injected via Injection Strategies. These Injection Strategies derive from the Phase 1 framework describing the phenomena and are central to the experiments. The combination of these injection strategies and the operationalization of metrics defined in the GQM hierarchy form the method and instrumentation needed to support the experiments performed in this phase.

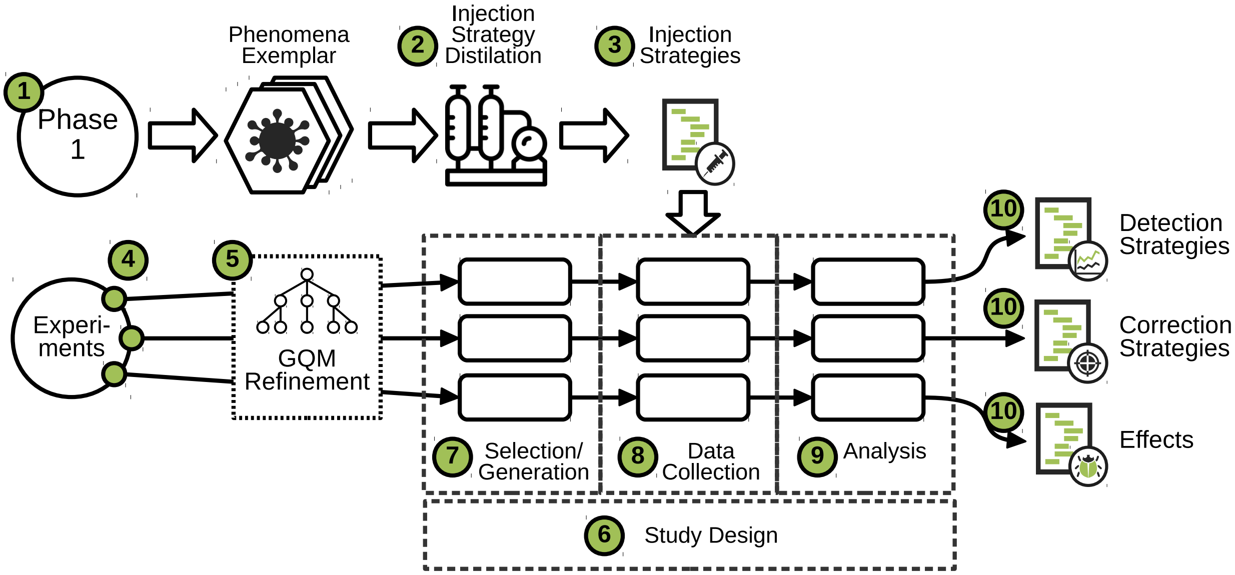


Figure 8.6: Phase 2 details.

The experiments execute after the refinement of the experimental methods, and the initial setup of instrumentation. The goal of these experiments is to examine causal relationships between selected measures and that the phenomena under study. Examining these relationships allows us to develop a theoretical basis upon which later studies can build. Regardless of whether the results are positive or negative, the system stores the results within the experience database for later use.

Figure 8.6 depicts the details of conducting the Phase 2 experiments. This process follows the sequence indicated by the numbers encircled in green, as follows: 1.) Resulting from Phase 1 experiential studies is a collection of example instances of the phenomena. 2.) These examples, combined with the theoretical framework, allow researchers to distill Injection Strategies describing the phenomena. These injection strategies generate instances of the phenomena as part of a process akin to the growing of cultures in pathological studies. This process allows for the investigation into the effects the investigated phenomena have on software systems. 3.) The result of this is a collection of Injection Strategies used during the data collection process of the experiments. 4.) Phase 2 experiments subdivide into three specific types: (i) Effects experiments, (ii) Detection experiments, and (iii) Correction experiments. The following further describes these experiments.

Each of the following experiments explores a specific facet of the phenomena in order to gain a complete understanding. Effects experiments establish the causal relationships between the manifestation of the phenomena and the effect it has on measures such as software quality. The Detection experiments establish the accuracy and effectiveness of proposed Detection Strategies on the detection of injected instances of the phenomena. These experiments also calibrate Detection Strategies, ensuring that the broadest variation of instances can be detected. Correction experiments define and establish the effectiveness of correction strategies concerning the mitigation of injected instances of the phenomena. This phase continues with the execution of the experiments selected.

5.) Each of these experiment types starts with the refinement of their respective sections of Phase 1 derived GQM hierarchy. 6.) Following this refinement, the study designs follow the guidance of Wohlin et al. [276]. 7.) The first process, within the experiment, is the selection of experimental subjects. Here, entities that are typically affected by the phenomena are either randomly selected from existing systems or generated via a randomized algorithmic process. 8.) Following selection, a refinement of the instrumentation used in Phase 1 experiments commences, followed by the execution of the data collection process. 9.) The data collected is then analyzed to answer the questions. 10.) These results, along with the generated detection strategies and correction, are recorded.

Chapter 10 exemplifies Phase 2. These studies evaluate the effects of design pattern grime on the measured software product maintainability (c.f. 2.4) and technical debt (c.f. 2.3) of generated design pattern instances (c.f. 4.3). These studies use the software injection process detailed in Chapter 6. The results of these studies form the basis for understanding the relationship between grime, quality and technical debt, and the basis for the verification experiments, case studies, and evolutionary studies detailed in the following subsections.

8.2.2 Bridge: *In Vitro* to *In Vivo*

The knowledge gained from experimentation provides a capability for causal reasoning, but it is not without limitations. Specifically, this experimental approach relies on the analysis of injected instances of software phenomena. Such an approach precludes the capability to observe the effects of "wild" instances found in live software systems. This capability, On the other hand, is inherent in case studies and longitudinal studies. Although, these studies typically preclude the ability for causal reasoning. Thus, a means by which we can bridge the gap between a strong foundation provided through experimentation and the insights gained via observational field studies are needed, leading to a combined form of study we call *Verification Experiments*.

Verification Experiments are experimental case studies conducted to validate the results of Effects, Inject, Correction, and Detection experiments within the context of real software systems. The key to this is the controlled use of software injection within real software systems allowing for the validation of the injection process itself. This approach provides an intermediate layer connecting the case studies of Phase 4 with the experiments of Phase 2. Phase 3, the single phase within this section, embodies the intermediate layer.

Phase 3: Verification Experiments. In this phase, we connect the approach used in the *in vitro* experiments with the observational nature of the *in vivo* case studies. There are two goals to this process. The first is to verify that the injection process works as expected (i.e., that it correctly injects the issues/items of concern). Secondly, this process verifies the accuracy of the injection process, such that injected instances are similar to those observed.

Figure 8.7 depicts the details of conducting Phase 3 verification experiments. This process follows the sequence indicated by the numbers encircled in green, as follows. 1.) There are two main verification experiments of concern: (i) Effects Verification and (ii)

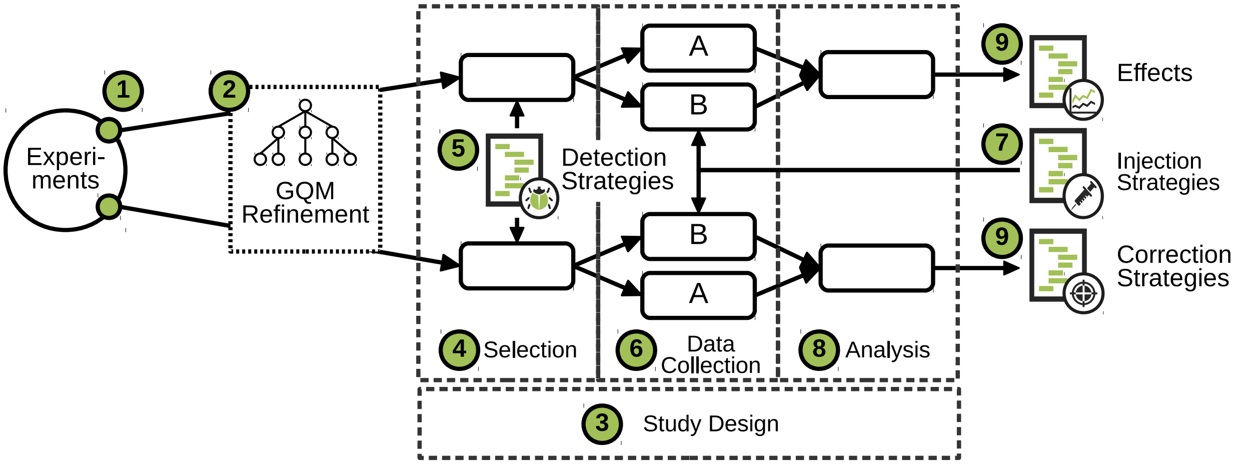


Figure 8.7: Phase 3 details.

Correction Strategy Verification. 2.) Both experiment types begin with a refinement of the respective section of the GQM hierarchy (from Phase 1). 3.) Following this refinement, the study is designed following the guidance of Wohlin et al. [276] and Runeson et al. [230]. 4.) Each study type begins with a project selection process, based on a rigorously defined case selection criteria. 5.) Detection strategies are executed across the selected projects to identify pairs of subsequent versions where the phenomena manifest. For the Effects Verification study, potential experimental subjects are those artifacts appearing in each version wherein the latter project contains the artifact afflicted. For the Correction Strategy Verification, potential experimental subjects are those artifacts appearing in each version wherein the former project contains the artifacts afflicted. The selection process then randomly selects from those identified pairs. 6.) When selection completed, the data collection process begins.

The data collection process begins with the creation of two data sets. The first data set marked (A), contains the natural evolution of the entities. That is, it contains the unmodified pairs of data. The second data set marked (B) contains the original starting version of the artifacts, and a version artificially evolved using either an injection or correction strategy, depending on whether it is an injection or correction verification experiment, respectively.

Upon data set construction, the instrumentation measures metrics (defined by the GQM) across the data sets, collecting data necessary to answer the GQM questions. 7.) After data collection completes, the results are analyzed to compare the differences in measurements, per pair, between the natural evolution and artificial evolution due to the application of injection or correction strategies. 8.) The system stores the results to the experience database. Research then use the analysis results to improve the injection and correction strategies.

Following this process, we conduct a verification study to validate that the observed effects of injected grime on software maintainability and technical debt (as documented in Chapter 10) match the effects we observed due to "wild" examples of grime found in open source Java™ software systems. This study is documented in Chapter 11.

8.2.3 *In Vivo* Field Studies

The goal of the *in vitro* experiments were to form the underlying foundational knowledge of the phenomena. This foundational knowledge expands, through *In vivo* studies, by observation of afflicted software components in live systems. Specifically, we wish to see how the phenomena and the knowledge gained in Phase 1 and 2 unfolds in actual software engineering contexts. This section consists of Phases 4 and 5.

Phase 4: Case Studies. In Phase 4, case studies provide a further understanding of the nature of the selected phenomenon. These studies provide the capability to observe the phenomena in multiple contexts, domains, and environments. Contexts to consider are the software type, i.e., open-source, industry, or governmental (or military), or the implementation language. Software domains may include games, business applications, operating systems, etc. Operating environments of the software may include, but are not limited to, cloud-based, desktop, and mobile environments. The more comprehensive a study is, the vaster the knowledge gained from it while reducing threats to external and conclusion

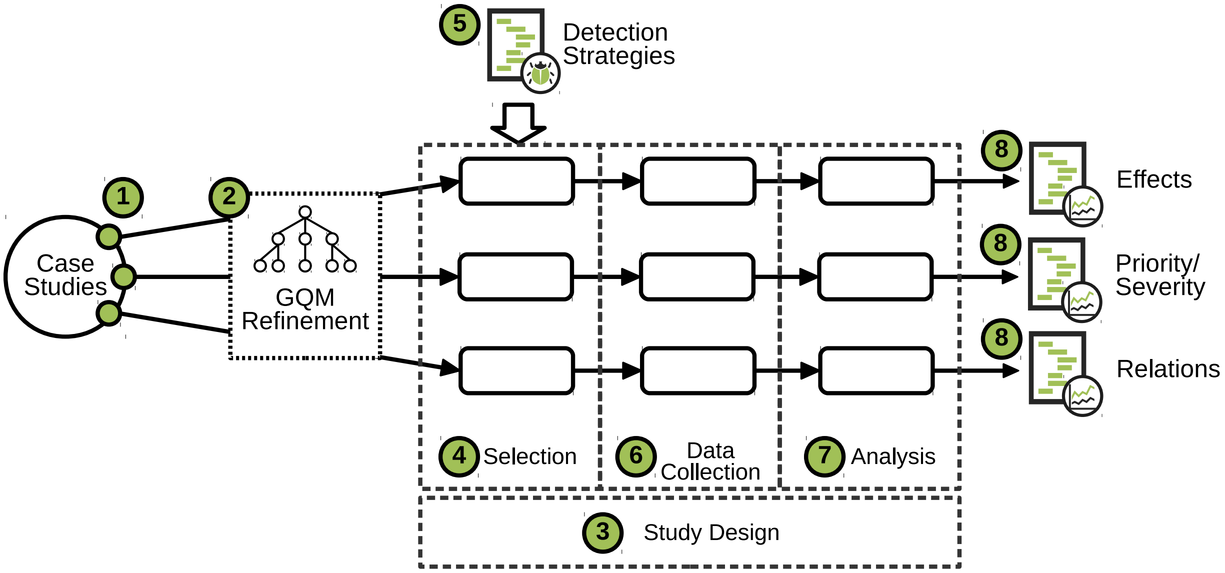


Figure 8.8: Phase 4 details.

validity. Additionally, case study methods provide the capability to gain an understanding of the phenomena, which is impossible to gain through experimentation.

Figure 8.8 depicts the details of conducting Phase 4 case studies. This process follows the sequence indicated by the numbers encircled in green, as follows. 1.) There are three types of case studies to consider, as follows. *Effects Case Studies* explore the effects of the phenomena on measures of interest (i.e. software quality aspects). *Priority/Severity Case Studies* develop and evaluate phenomena subtype prioritization techniques as a means to assign severity to instances of each subtype. *Relations Case Studies* explore the relationships between phenomena subtypes and relationships between forms of software engineering phenomena. 2.) Each of these study types begins with a refinement of the appropriate section of the GQM hierarchy extracted in Phase 1.

3.) Following this refinement, the studies design commences using the guidance of Runeson et al. [230]. 4.) Once designed, execution begins starting with a random project (case) selection process, based on a rigorously defined case selection criteria. 5.) Once projects are selected, detection strategies execute within the selected projects to identify artifacts afflicted by instances of the phenomena. For Effects Case Studies, the selection process considers changes occurring between versions to identify artifacts that become afflicted between version changes to evaluate the effects due to that change. In Priority/Severity and Relations Case Studies, the selection process only selects the current version of individual projects. 6.) The data collection executes the metrics (from the GQM hierarchy) measurement instruments. 7.) Researchers analyze the collected data to gain insight concerning the effects, priority/severity, or relations of the phenomena under study. The system stores this data in the experience database for later use.

Phase 5: Evolutionary Studies Evolutionary studies aim to expand upon the knowledge gained from the previous phase by examining instances of the phenomena as they change over time. Similar to case studies, this study includes instances across multiple contexts, including time. In the case of software systems, the units of time are versions, commits to a repository, release dates, or more typical time measurements such as days, weeks, months, or years. The aspect of time provides the capability to evaluate both changes in the phenomena and the system containing it. A desire to evaluate these changes leads to studies concerning the following: the accumulation of the phenomena, the susceptibility of artifacts to such accumulation, the priority of subtypes of the phenomena (when concerned with the need to address these issues), the relative severity of individual phenomena instances, and the ripple effects caused by the accumulation within the system.

Figure 8.9 depicts the details of conducting Phase 5 Evolutionary Studies. This process follows the sequence indicated by the numbers encircled in green, as shown. 1.) The three types of longitudinal studies to be considered. First, *Buildup/Susceptibility Studies* explore the accumulation of the phenomena within a software system and component susceptibility to accumulation. Secondly, *Priority/Severity Studies* evaluate the changes in instances of the phenomena and how this affects their priority and severity. Finally, Ripple Effect Studies

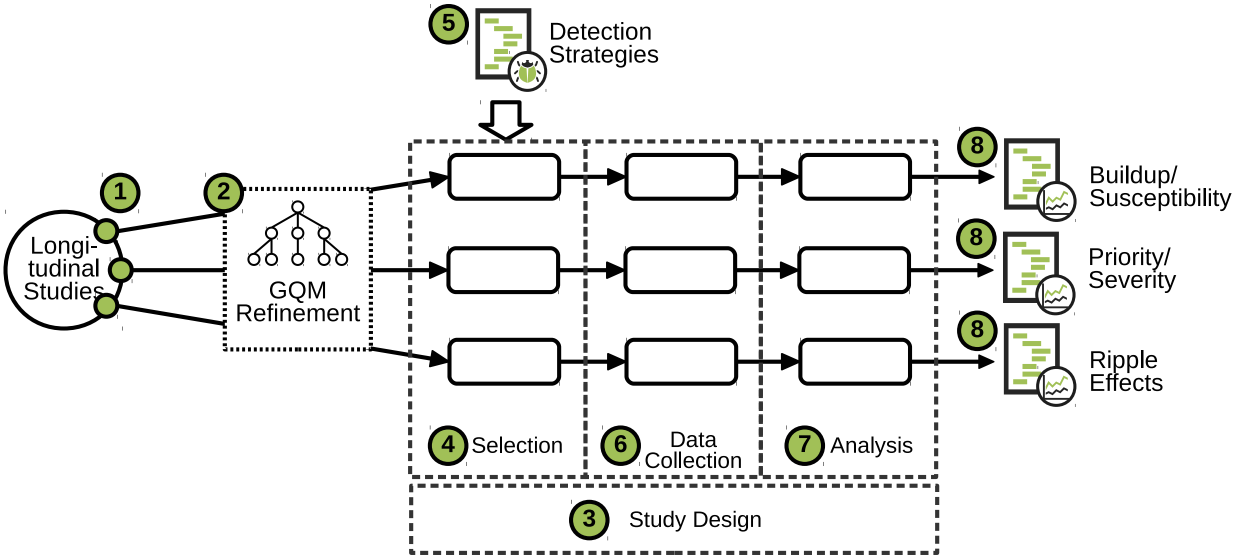


Figure 8.9: Phase 5 details.

evaluate the effects on connected components overtime after the point in time that the phenomena manifest. 2.) Each of these studies starts with the refinement of the appropriate section of the GQM hierarchy developed in Phase 1. 3.) Following this refinement, the study design relies on the guidance of Runeson et al. [230] for longitudinal case studies. 4.) Each study type begins with a random project selection process, based on a rigorously defined case selection criteria. This selection criteria defines the range of system/project versions and identifies whether each version contains artifacts afflicted by the phenomena under study. 5.) For each phenomenon subtype, detection strategies identify afflicted artifacts to study. 6.) Identified artifacts are then subject to the data collection process. This process collects the metrics identified as part of the GQM hierarchy, using selected instrumentation. 7.) This collected data is then analyzed to answer the questions and provide further insight concerning buildup/susceptibility, priority/severity, or ripple effects of the phenomena under study. Finally, the system stores the results of the analysis in the experience database for later use.

8.2.4 Bridge: *In Vivo* Results Informing *In Vitro* Experiments

The results of both the *in vitro* experiments and the *in vivo* studies bring together a whole picture of the phenomena under study. The knowledge gathered to this point, via a combination of studies, does not provide a clear method to operationalize this knowledge towards efficient means of detecting or correcting the phenomena studied. Furthermore, it does not provide a clear means as to provide the capability to predict the effects of an instance in the software, nor does it provide a means to decide if and when such an instance should be corrected. To address these limitations and bridge the gap between the prior studies and the initial knowledge gathered, we turn towards the application of model building studies. Here, the goal shifts from evaluating the effects of the phenomena under study and towards prediction and model building. This approach will improve our understanding of the effects of this phenomenon while developing new models and techniques for efficiently detecting and correcting instances of these phenomena. This section consists of a single phase, Phase 6.

Phase 6: Model Building. In this phase, our goal is to utilize machine learning and data science techniques to connect the results of the case studies and evolutionary studies to the results of the initial experiments. The reason this phase is the last in the process is three-fold. First, we must understand the phenomena before developing predictive models for the phenomena's effects or its priority/severity. Second, we need confirmed examples of the phenomena before generating effective detection or correction strategies. Lastly, we require the results of all the prior studies as a foundation to develop approaches that determine the future effects of an instance and whether such an instance should be corrected.

That is, Phases 1–5 provide the necessary understanding of the phenomena to begin the development of predictive models. These models can indicate the likely effects of an instance's continued presence within a software system, and decide which instance to remediate. The same information should provide empirical results in improving detection strategies and

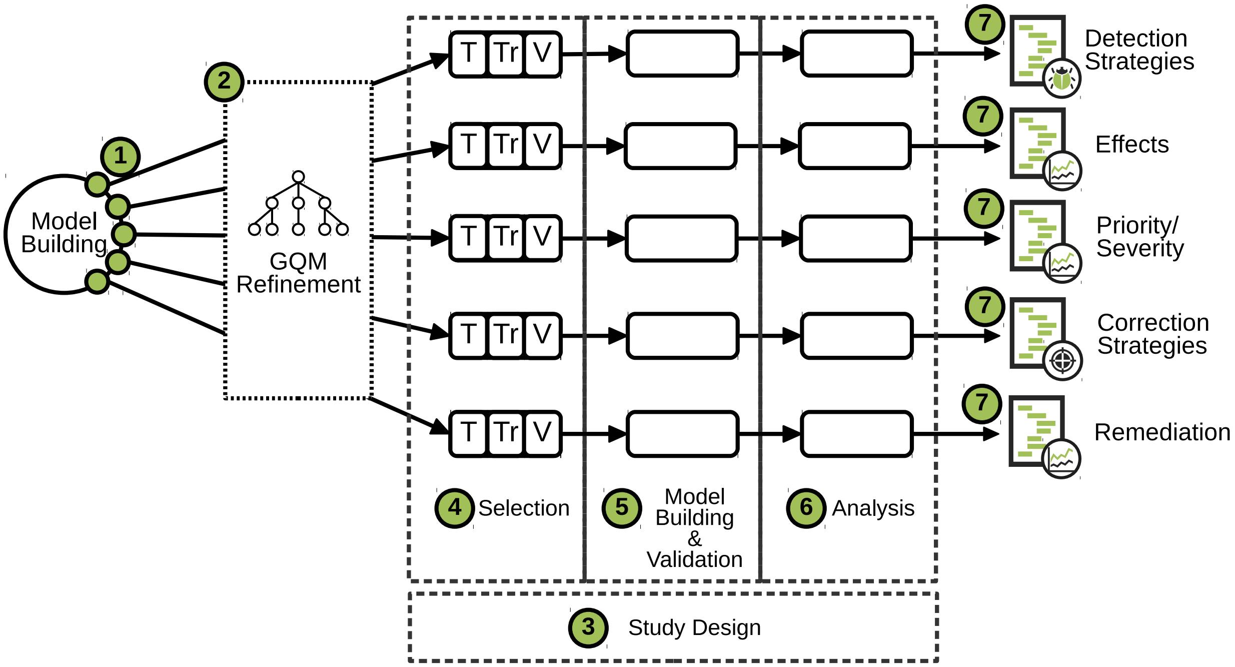


Figure 8.10: Phase 6 details.

correction strategies using automated generation processes (e.g. evolutionary programming). Furthermore, the studies in Phase 1–5 will produce the necessary datasets used to train these techniques.

Figure 8.10 depicts the details of conducting Phase 6 Model Building Studies. This process follows the sequence indicated by the numbers encircled in green, as follows. 1.) There are five types of model building studies to be considered. Firstly, *Detection Strategy Studies* develop and evaluate approaches to automated generation of detection strategies. Next, *Correction Strategy Studies* develop and evaluate approaches to automated generation and evaluation of correction strategies. Next, *Effects Studies* develop predictive models of the effects of phenomena instances. Next, *Priority/Severity Studies* develop and evaluate techniques for automated assignment of phenomena priority/severity. Finally, *Remediation Studies* explore the development of predictive models based on prior knowledge concerning the evolution, priority assessment, severity assessment, and other properties of the instance and the surrounding system, and explore the creation of decision models for instance remediation. 2.) Each of these studies begins with the refinement of the appropriate section of the GQM hierarchy extracted in Phase 1. 3.) Next, Researchers develop study designs following the guidance of Alpaydin [5] for machine learning experiments.

4.) Using this guidance, Researchers divide the data sets gathered in prior phases randomly into three subsets, as follows: (T) Test for testing the models, (Tr) Training for building/training the models and (V) Validation for validating the trained models. 5.) Using these datasets, the model building and validation step are conducted. As the models are built, a *k-fold cross-validation* process is used to evaluate each of the models. Models may be constructed using several different model-building techniques that are used as part of this process, with the goal of selecting the best one/combination. 6.) Performance data is collected by applying the model to the (V) dataset for each of the techniques used. The resulting data for each technique are compared and statistically analyzed. 7.) The results are stored in the experience database for later use. The generated detection and correction strategies replace prior versions for use in future studies. The results of the Effects and Priority/Severity Studies update the definitions of the phenomena.

8.3 Future Implications and Conclusions

This chapter illustrated the guiding process embodied within this work and the experimental Arc Framework. Through the application of this process and the Arc Framework, the following chapters show the power of this approach to explore software engineering phenomena with a focus on design pattern grime. Though shown within this research is a specific form of design disharmony, this process applies to any form of software engineering phenomena manifesting within the confines of software artifacts. Examples of such phenomena include (but are not limited to) the issues composing the technical debt landscape: antipatterns, code smells, design pattern grime, and static analysis issues [131].

Furthermore, by extending the Arc Framework, behavioral and dynamic analysis issues are also within the grasp of the study.

Thus, this process is not limited to the source code level. Instead, this approach (and its underlying framework, Arc) applies to the study and evaluation of other forms of software artifacts, including requirements, documentation, build scripts, and repository information. These additional artifacts require extensions to the Arc framework to construct a sound basis for experimentation. However, such a foundation allows for the extension of knowledge through the execution of case studies and evolutionary studies. Thus, allowing for the interrelation of multiple studies facilitating a deeper understanding of the nature of software engineering phenomena. The following chapters demonstrate this approach and provide evidence in support of these claims.

CHAPTER NINE

DESIGN PATTERN GRIME TAXONOMY

Good judgment comes from experience, and experience comes from bad judgment.

–Frederick P. Brooks

9.1 Introduction

This chapter describes the extensions to grime taxonomy by expanding the Class and Organizational leaves of the original grime taxonomy defined by Izurieta [132]. We first elaborate on the approach used to define the taxonomies. We then define the extended taxonomies for Class and Organizational Grime. We note that this taxonomy is only a refinement rather than a final taxonomy of all grime.

RG1: Analyze design patterns to elaborate on the complete taxonomy of Class and Organizational Grime.

RQ1.1: What are the types of Class Grime?

Rationale: This is a fundamental question of this research, inquiring as to the nature of Class Grime.

RQ1.2: What are the types of Organizational Grime?

Rationale: This is a fundamental question of this research, inquiring as to the nature of Organizational Grime.

Organization This chapter is organized as follows. Section 9.2 defines our process for the definition and refinement of a taxonomy. Subsection 9.3 provides a formal definition of the concepts underlying the taxonomy and provides for the predicates and set definitions necessary to describe the concepts defining grime categories. Section 9.4 describes the basis, conceptual components, and formal definitions of the Modular Grime taxonomy developed by Schanz and Izurieta [234]. Section 9.5 describes the basis, conceptual components, and formal definitions of the Class Grime taxonomy. Section 9.6 describes the basis, conceptual components, and formal definitions of the Organizational Grime taxonomy. Finally, Section 9.7 concludes this chapter.

9.2 Taxonomy Definition Process

The goal of the taxonomy definition process is to elaborate possible grime subtypes, further refining existing categories as was done by Schanz and Izurieta [234]. To extend this taxonomy, we developed a process to elaborate on the Class and Organizational Grime types further. The steps of this process are, as follows:

1. Develop an underlying formal framework necessary to develop the taxonomy.
2. Identify the software entities of concern, such as classes, packages, or relationships.
3. Identify the design principles or practices that affect these entities, which have not already been elaborated upon by existing disharmonies.
4. Identify the measurable properties of these principles or their components to develop the levels of the taxonomy.
5. Select metrics to measure the identified properties.
6. Define, formally, each type of grime identified as part of the newly extended taxonomy.

The following sections utilize this process to develop the Class Grime and Organizational Grime taxonomies.

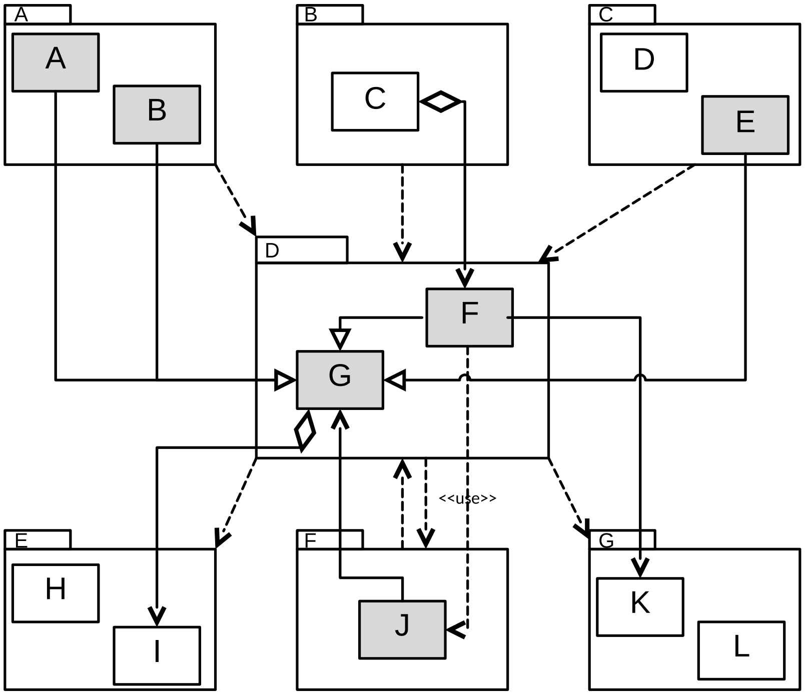


Figure 9.1: Example Package Graph.

9.3 Formal Framework

Design pattern grime categories, as defined by the following taxonomies, require an underlying formal framework. Initially, we assume there is a system under analysis, $\mathcal{S}$, and that this system is defined using an object-oriented language such as UML or Java™. Furthermore, a system can be described as $\mathcal{S} = \langle \mathcal{T}, \mathcal{Z} \rangle$, which is a tuple containing the set of types, $\mathcal{T}$, and relationships, $\mathcal{Z}$, of the system. We also define $\mathfrak{R}$ as the set of formal specifications of design patterns using a design pattern specification language such as RBML. To further define this framework, we consider two different perspectives: structural and relational.

9.3.1 Structure

This subsection details the formal concepts related to the structural aspects of the system and pattern representations. Each type, $t \in \mathcal{T}_{\mathcal{S}}$, can be further defined as $t = \langle \mathcal{A}_t, \mathcal{M}_t \rangle$, which is a tuple containing the set of attributes for type t, $\mathcal{A}_t$, and the set of methods for type t, $\mathcal{M}_t$. is further composed of a set of attributes $\mathcal{A}(t)$. The set of attributes, $\mathcal{A}_t$ represents the data directly defined in the type. The set of methods $\mathcal{M}(t)$ which represent the behavior associated with the type and its contained data. For example, Figure 9.1 represents a system, $\mathcal{S}$, wherein $\mathcal{T}_{\mathcal{S}} = \{A\,B\,C\,D\,E\,F\,G\,H\,I\,J\,K\,L\}$ and $\mathcal{Z}_{\mathcal{S}}$ is the set of relationship between the types (excluding the dashed arrows between packages).

Types can be either internal or external to a package as determined by the predicates *internal* and *external*, which are defined as follows: $internal(t \in \mathcal{T}, p \in \mathcal{P}) \equiv t \in p$ which evaluates to true if t is a type within the package p. $external(t \in \mathcal{T}, p \in \mathcal{P}) \equiv \neg internal(t, p)$ which evaluates to true if t is a type not in the package p. For example, in Figure 9.1 $internal(G, D)$ evaluates to true, as type G is in package D, and $external(G, A)$ evaluates to true as well. On the other hand, $internal(F, B)$ evaluates to false and similarly $external(K, G)$ evaluates to false. A system's types can also be partitioned into a set of packages (also known as namespaces), $\mathcal{P}_{\mathcal{S}}$. The set of packages can be defined using a set partition as follows: $\bigcup_{i \in \mathcal{P}_{\mathcal{S}}} = \mathcal{T}_{\mathcal{S}} \wedge \mathcal{P}_{\mathcal{S}_i} \cap \mathcal{P}_{\mathcal{S}_j} = \varnothing \text{ for } i, j \in \mathcal{P}_{\mathcal{S}} \wedge i \neq j$. We assume that a single type, t, may only be in a single package at one time. Thus, the set of types contained in a package $p \in \mathcal{P}_{\mathcal{S}}$ is a proper subset of the types of the system itself: $\mathcal{T}_p \subseteq \mathcal{T}_{\mathcal{S}}$. For example, in Figure 9.1 the set $\mathcal{P}_{\mathcal{S}} = \{\{A\,B\}\,\{C\}\,\{D\,E\}\,\{G\,F\}\,\{H\,I\}\,\{J\}\,\{K\,L\}\}$.

The set of methods, $\mathcal{M}_t$, for a given type, t, can be further partitioned into *operations*, *accessors*, *mutators*. The set of accessors is extracted using the function $accessors(t)$, which evaluates to the set of accessor methods defined in type t. A method, $m \in \mathcal{M}_t$, can be determined if it is an accessor for attribute, $a \in \mathcal{A}_t$, in type t using the predicate $accessor(m, a)$. The predicate $accessor(m, a)$ evaluates to true if method m simply returns

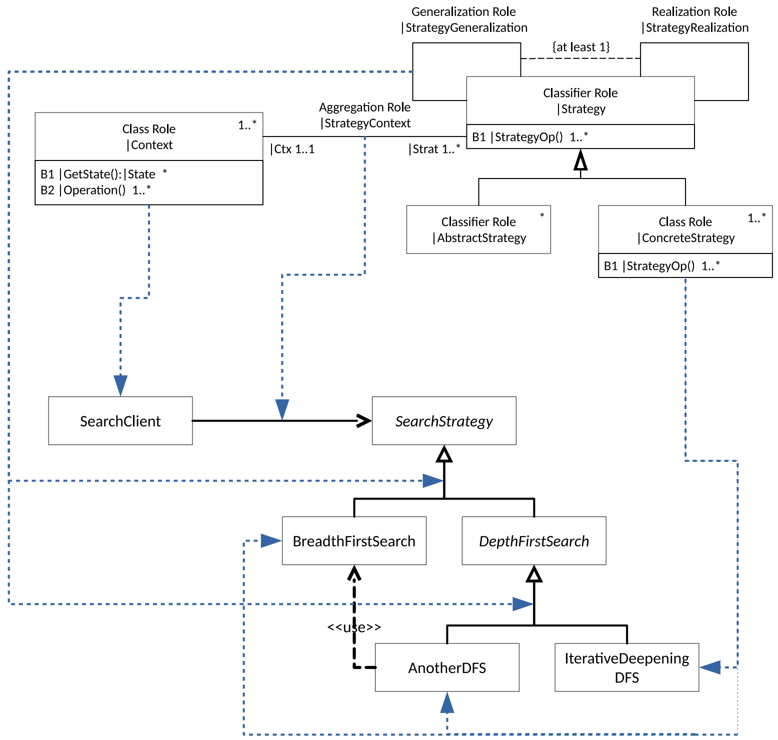


Figure 9.2: Example Pattern Graph.

the value of attribute a and does nothing else. Similarly, the set of mutators is extracted using the function $mutators(t)$, which evaluates to the set of mutator methods defined in type t. A method, $m \in \mathcal{M}_t$, can be determined if it is a mutator for attribute, $a \in \mathcal{A}_t$, in type t using predicate $mutator(m, a)$. The predicate $mutator(m, a)$ evaluates to true if method m simply modifies the value of attribute a and does nothing else. Finally, the set of methods calling a method m can be extracted using the $calls(m)$ function.

A design pattern realization $\mathcal{L} = \langle r \in \mathfrak{R}, \mathcal{B}_r \rangle$ is a tuple consisting of a design pattern specification (as defined using RBML [90]), $r \in \mathfrak{R}$, and a mapping of the roles of r to the components of the system, $\mathcal{B}_r$. This binding is defined as, $\mathcal{B}_r \equiv bind : \mathcal{C}_\mathcal{S} \rightarrow \mathcal{R}$. The set $\mathcal{C}_\mathcal{S} = \mathcal{T}_\mathcal{S} \cup \mathcal{A}_\mathcal{S} \cup \mathcal{M}_\mathcal{S} \cup \mathcal{P}_\mathcal{S}$ which is the union of types, attributes, methods, and packages defined across the system.

A pair of methods (m_i, m_j) can be either internal or external to a pattern realization.

This is determined by the *internal* and *external* predicates for method pairs. The *internal* predicate is defined as $internal((m_i, m_j, l \in \mathcal{L}) \in \mathcal{M}_t) \equiv internal(m_i, l) \wedge internal(m_j, l)$ which evaluates to true if both methods m_i and m_j are are both internal to the pattern representation l. The *external* predicate is defined as $external((m_i, m_j, l \in \mathcal{L}) \in \mathcal{M}_t) \equiv external(m_i, l) \vee external(m_j, l)$ which evaluates to true if either method m_i or m_j is determined to be external to the pattern realization. Both of the method pair predicates are based on the singular method predicates. The singular method *internal* predicate is defined as $internal(m \in \mathcal{M}_t, l \in \mathcal{L}) \equiv \mathcal{B}_l(m) \neq \varnothing$ which evaluates to true if there is a binding from method m to a role in a pattern specification of pattern realization l as defined by $\mathcal{B}_l$. Similarly, the singular method *external* predicate is defined as $external(m \in \mathcal{M}_t, l \in \mathcal{L}) \equiv \neg internal(m, l)$

9.3.2 Relationships

In the context of design pattern grime and object-oriented systems, we define three specific types of relationships: (i) usage relationships between methods and attributes of the same Class, (ii) connection relationships between classes, and (iii) dependency relationships between packages. The remainder of this subsection further defines these relationships.

A relationship simply defines a directed connection from one component of a system to another, as in: $r \equiv (c_1, c_2)$. Here, c_1 and c_2 are two components of the system. A relationship can be determined to be either persistent or temporary based on the type of relationship it is. This determination can be evaluated using the *persistent* and *temporary* predicates. The *persistent* predicate is defined as $persistent(r \in \mathcal{Z}) \equiv r.type \in \{Generalization, Realization, Association\}$ evaluates true if the relationship is of type Generalization, Realization, or Association (including Composition or Aggregation). The *temporary* predicate is defined as $temporary(r \in \mathcal{Z}) \equiv \neg persistent(r)$ evaluates true if not persistent. For example, in Figure 9.1 $persistent((A, B))$ and $persistent(C, F)$ both evaluate

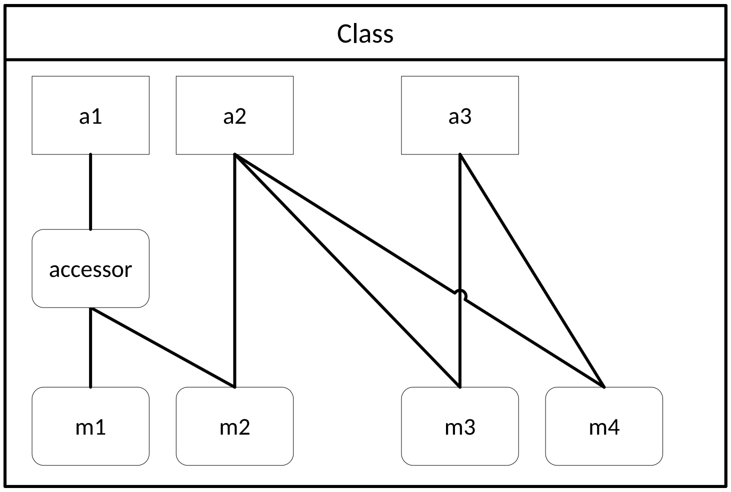


Figure 9.3: Example Composite Graph.

to true, while $persistent(F, J)$ evaluates to false. On the other hand, $temporary(F, J)$ evaluates to true, while $temporary(J, G)$ evaluates to false.

Usage relationships describe the connection between the methods of a class and the attributes within that Class. These connections form when the statements of a method's body contain a read or write to a field of the method's containing Class. Thus the predicate $usage(m, a)$ evaluates to true if for some type, t, there is a method $m \in \mathcal{M}_t$ and an attribute $a \in \mathcal{A}_t$ where m contains a statement which reads or writes the value of a. The set of usages of a type, t, can be defined as $\mathcal{U}_t \equiv \{\langle m, a\rangle \mid usage(m, a) \wedge m \in \mathcal{M}_t \wedge a \in \mathcal{A}_t\}$.

A pair of methods can either directly or indirectly access an attribute. This can be evaluated using the *direct* or *indirect* predicates for method pairs. The *direct* predicate is defined as $direct((m_i, m_j) \in \mathcal{M}_t, a) \in \mathcal{M}_t) \equiv direct(m_i, a) \wedge direct(m_j, a)$ and evaluates to true if both the usage from method m_i to attribute a is a direct access (rather than through an accessor or mutator) *and* the usage from m_j to a is also direct access. On the other hand, the *indirect* predicate for method pairs is defined as $indirect((m_i, m_j) \in \mathcal{M}_t, a) \equiv indirect(m_i, a) \vee indirect(m_j, a)$ which evaluates to true if either the usages from method m_i or from method m_j to attribute a is via an accessor or mutator. Both the method pair *direct* and *indirect* predicates are derived from the singular method *direct* and *indirect* predicates. The singluar method *direct* predicate is defined as $direct(m \in \mathcal{M}_t, a \in \mathcal{A}_t) \equiv (m, a) \in \mathcal{U}_t$ which evaluates to true if there is a direct usage between method m and attribute a and m and a are defined in the same type t. Similarly the singluar method *indirect* predicate is defined as $indirect(m \in \mathcal{M}_t, a) \equiv (calls(m, m_i \in accessors(t)) \wedge accessor(m_i, a)) \vee (calls(m, m_i \in mutators(t)) \wedge mutator(m_i, a))$ which evaluates to true if there is call from m to some method m_i which is either an accessor or mutator for attribute a and all are defined in the same type t. Figure 9.3 depicts several of the type properties as well as the aspects evaluated by the previously defined predicates. For example, in Figure 9.3 we can see that for the type, $Class$, depicted; the attributes set is defined as $\mathcal{A}_{Class} = \{a1\, a2\, a3\}$ and the methods set is defined as $\mathcal{M}_{Class} = \{accessor\, m1\, m2\, m3\, m4\}$. Additionally, we can see that $direct((m1, m2), a1)$ evaluates to false due to the fact that both $direct(m1, a1)$ and $direct(m2, a1)$ evaluate to false. On the other hand, $direct((m3, m4), a3)$ evaluates to true as both $direct(m3, a3)$ and $direct(m4, a3)$ evaluate to true. Additionally, $indirect(m2, a1)$ evaluates to true, but $indirect(m3, a2)$ evaluates to false.

Connections represent relationships between classes. Relationships that form connections are Associations (including Aggregations and Compositions), Generalizations, Realizations, and Dependencies (such as Usage). Connections are between classes and are used to define the relationships between packages (known as dependencies). Note, multiple connections could realize a single dependency. The set of connections in the system can be defined as $\mathcal{K} \equiv \{\langle a, b\rangle \mid conn(x, y)\}$. For example, in Figure 9.1 $conn(F, K)$ evaluates to true, but $conn(K, L)$ evaluates to false.

A connection (t_1, t_2) can be determined as invalid in the context of a design pattern realization using the following predicate: $invalid(r = (t_1, t_2) \in \mathcal{K}, l \in \mathcal{L}) \equiv (r \notin \mathcal{B}_l \wedge internal(r)) \vee (internal(t1, l) \wedge external(t2, l))$ For example, in Figure 9.2 $invalid((SearchClient, SearchStrategy))$ evaluates false as there is a binding to that relationship, but $invalid(AnotherDFS, BreadthFirstSearch)$ evaluates to true as there is no binding to that relationship.

A connection can be either internal or external to a pattern realization. This can be evaluated using the predicates *internal* and *external*. The *internal* predicate is defined as $internal((t_i, t_j) \in \mathcal{K}, l \in \mathcal{L}) \equiv t_i, t_j \in \mathcal{T}_l$ and evaluates true if the provided connection is between two types defined within the pattern realization, l. The *external* predicate is defined as $external((t_i, t_j) \in \mathcal{K}, l \in \mathcal{L}) \equiv t_i \notin \mathcal{T}_l \vee t_j \notin \mathcal{T}_l$ and evaluates to true if the provided connection has at least one type which is external to the pattern specification. For example, in Figure 9.2 $internal(SearchClient, SearchStrategy)$ evaluates to true, as both SearchClient and SearchStrategy are members of the pattern realization and fulfill roles according to the pattern specification.

Dependencies represent relationships between packages. These relationships (as noted previously) are formed by the connections between classes that cross package boundaries. Thus the predicate $depend(x, y) \equiv \exists_{c \in \mathcal{T}_x, d \in \mathcal{T}_y}\, conn(x, y)$ evaluates to true if there exists a class c in package x and a class d in package y such that there is a connection between classes c and d. The set of dependencies in the system can be defined as $\mathcal{D} \equiv \{\langle x, y\rangle \,|\, depend(x, y) \wedge x, y \in \mathcal{P} \wedge x \neq y\}$. For example, in Figure 9.1 $depend(A, G)$ evaluates to true, but $depend(A, B)$ evaluates to false. For example, in Figure 9.1 the set $\mathcal{D} = \{(A, D)\,(B, D)\,(C, D)\,(D, E)\,(D, F)\,(F, D)\,(D, G)\}$, while the set of connections between classes $\mathcal{K} = \{(A, G)\,(B, G)\,(C, F)\,(E, G)\,(F, G)\,(F, K)\,(G, I)\,(J, G)\}$.

A sequence of dependencies where the target of a preceeding dependency is the source of a succeeding depedency is called a path. Determining if such a sequence exists from one package p_i to another p_j is determined by the predicate $path(p_i, p_j)$. For example, in Figure 9.1 $path(A, E)$ evaluates to true for packages A and E. Paths can then be used to form the set of packages reachable from a given package p, which is defined as $reachable(p) \equiv \{x | path(p, x) \wedge p, x \in \mathcal{P}\}$ represents the set of packages to which a path (excluding cycles).

For example, in Figure 9.1 $reachable(A) = D\,E\,F\,G$. Additionally, a path can form a cycle if the source of the first dependency in the sequence is the target of the last dependency in the sequence. Formally, we can define a predicate *cycle* as $cycle(p, q) \equiv path(p, q) \wedge depends(q, p)$ which evaluates to true when there is a path from package p to package q and a dependency from q to p. Often there packages define simple cycles where a package p depends on package q and q also depends on p. This is formalized by the predicate $directCycle(p, q) \equiv depends(p, q) \wedge depends(q, p)$. For example, in Figure 9.1 $cycle(D, F)$ and $directCycle(F, D)$ both evaluate to true.

Each type of relationship defined above allows for the representation of the entities as a graph structure. Usage relationships between the methods and attributes of a Type construct a bipartite directed graph. On the other hand, Connections represent the structural relationships between types forming a directed graph similar to a UML class diagram. Dependencies represent the structural relationships between packages, which form a directed graph similar to a UML Package Diagram.

9.4 Modular Grime

Modular Grime is the build-up of unnecessary and invalid relationships between classes of a pattern realization and between classes of a pattern and classes external to the pattern. Schanz and Izurieta [234] defined the Modular Grime taxonomy, c.f. Figure 9.4, included here for completeness. We have re-envisioned the definitions of Modular Grime to be consistent with the above formal framework while being equivalent to the original definitions.

9.4.1 Class Coupling

Coupling describes the interconnectedness of a set of classes. Based on this and the principle of Low Coupling, Schanz and Izurieta decomposed coupling into three categories: (i) Strength, (ii) Scope, and (iii) Direction, which form the levels of the Modular Grime

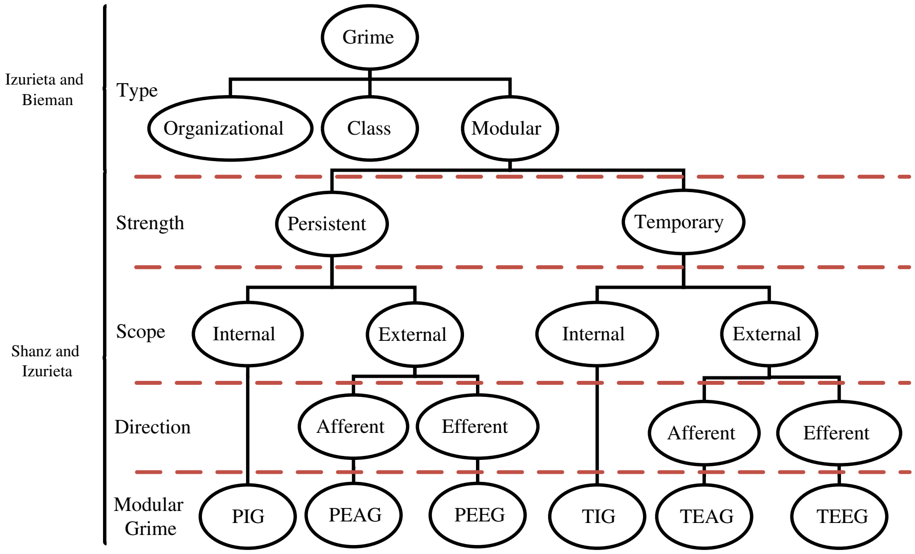


Figure 9.4: The extended Modular Grime taxonomy.

taxonomy.

Strength Strength indicates the strength of the coupling between classes. The strength may be either persistent (couplings based on associations, generalizations, or realizations between classes) or temporary (couplings based on local variable types, method return types, or method parameter types (aka Use Dependencies)). Persistent couplings are more challenging to remediate via techniques such as refactoring, whereas temporary relationships tend to be more amenable to remediation via techniques such as refactoring.

Scope Scope indicates that a relationship contained within the context of the pattern or not. Thus, the scope may be either internal (a relationship defined between two types fulfilling roles within the pattern realization) or external (a relationship defined between a type fulfilling a role in the pattern realization and one outside the definition of the pattern realization).

Direction Direction indicates the direction of the coupling between types, in the context of the pattern realization. The direction is only pertinent to those relationships that are determined to be external. Relationships that are outgoing from the pattern realization are called efferent couplings, and those that are incoming to the pattern realization are called afferent couplings. These are measured using the Ce and Ca metrics [183], respectively. Where Ce is the count of outgoing dependencies from a package to other packages, or in the context of a pattern realization, it is the count of the outgoing couplings from pattern internal classes to pattern, external classes. Similarly, Ca is the count of incoming dependencies from other packages into a package, or in the context of a pattern realization, it is the count of the incoming couplings from pattern external classes to pattern internal classes.

9.4.2 Modular Grime Examples

Figure 9.5 depicts an example of Persistent Internal Grime (PIG). The figure depicts a Strategy Pattern realization in which all depicted classes are internal to the pattern. Except for the green relationship from “AnotherDFS” to “BreadthFirstSearch,” all relationships are valid based on the Strategy Pattern RBML definition. The green relationship indicates an example of PIG.

Figure 9.6 depicts an example of Persistent External Efferent Grime. The dashed red line partitions pattern realization connected classes into either internal classes (above and to the right) or external classes (below and to the left). Furthermore, we see that there are two incoming (efferent) associations (persistent) relationships crossing this border, which are invalid to the pattern definition. These two lines (marked green) are two examples of PEEG.

In Figure 9.9 we can see an example of Direct Internal Single Grime (DISG). The figure is the representation of a pattern class. The dashed red line indicates those methods internal (specified by the pattern) or external (not specified by the pattern) to the pattern. The figure depicts a case of DISG as there is a method, $m1$, allowed by the pattern specification that

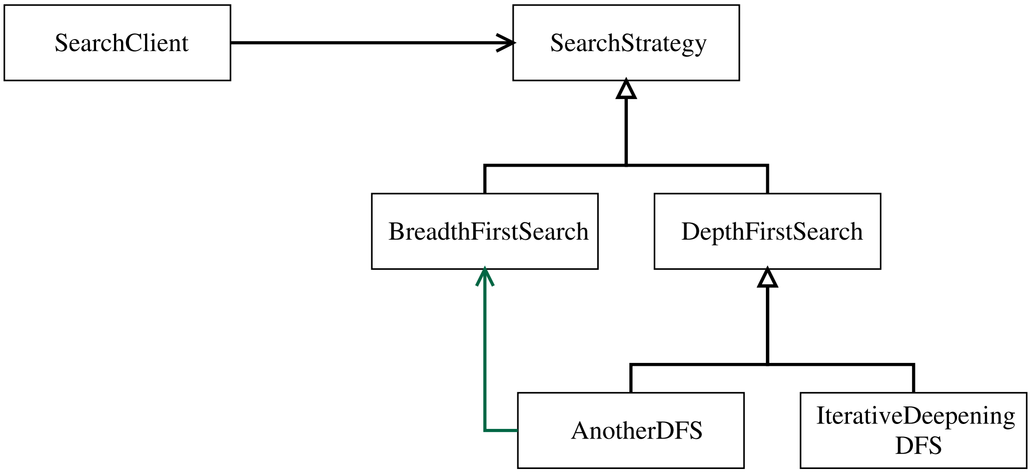


Figure 9.5: An example of the PIG type of Modular Grime.

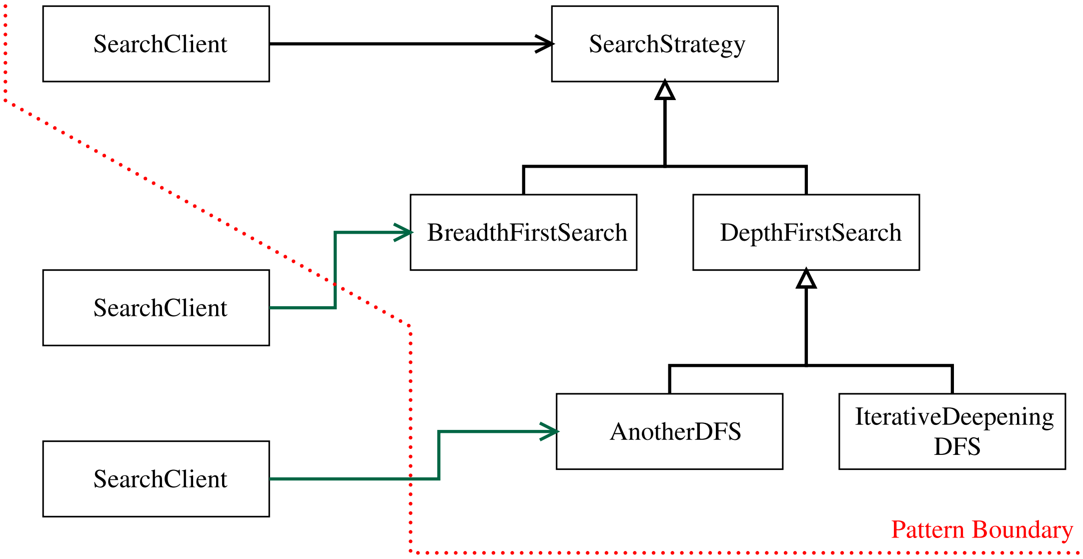


Figure 9.6: An example of the PEEG type of Modular Grime.

directly uses an attribute, $a1$, but no other method uses that attribute. This instance of grime indicates that there could be an unintentional secondary responsibility associated with this Class or a misunderstanding on the part of the developer in the pattern's implementation.

9.4.3 Modular Grime Categories

Using the notions of Strength, Scope, and Direction, Schanz and Izurieta defined six specific categories of Modular Grime. In the following, we describe these categories and provide a formal set definition for each based on the framework from Section 9.3.

Persistent Internal Grime (PIG) The set of relationships which are persistent, internal to the pattern realization l, are invalid according to the pattern specification of l and where the number of connections between classes of l increases. Formally, this defines the set $\{r \,|\, l \in \mathcal{L_S} \wedge persistent(r) \wedge internal(r,l) \wedge invalid(r, l.r) \wedge |\mathcal{K}_l| \uparrow\}$.

Temporary Internal Grime (TIG) The set of relationships which are temporary, internal to the pattern realization l, are invalid according to the pattern specification of l and where the number of connections between classes of l increases. Formally, this defines the set $\{r \,|\, l \in \mathcal{L_S} \wedge temporary(r) \wedge internal(r,l) \wedge invalid(r, l.r) \wedge |\mathcal{K}_l| \uparrow\}$.

Persistent External Efferent Grime (PEEG) The set of relationships which are persistent, external to the pattern realization l, are invalid according to the pattern specification of l, and which increase the efferent coupling of pattern instance l. Formally, this defines the set $\{r \,|\, l \in \mathcal{L_S} \wedge persistent(r) \wedge external(r,l) \wedge invalid(r, l.r) \wedge Ce(l) \uparrow\}$.

Temporary External Efferent Grime (TEEG) The set of relationships which are temporary, external to the pattern realization l, are invalid according to the pattern specification of l, and which increase the efferent coupling of pattern instance l. Formally, this defines the set $\{r \,|\, l \in \mathcal{L_S} \wedge temporary(r) \wedge external(r,l) \wedge invalid(r, l.r) \wedge Ce(l) \uparrow\}$.

Persistent External Afferent Grime (PEAG) The set of relationships which are persistent, external to the pattern realization l, are invalid according to the pattern specification of l, and which increase the afferent coupling of pattern instance l. Formally, this defines the set $\{r \mid l \in \mathcal{L}_{\mathcal{S}} \land persistent(r) \land external(r, l) \land invalid(r, l.r) \land Ca(l) \uparrow\}$.

Temporary External Afferent Grime (TEAG) The set of relationships which are temporary, external to the pattern realization l, are invalid according to the pattern specification of l, and which increase the afferent coupling of pattern instance l. Formally, this defines the set $\{r \mid l \in \mathcal{L}_{\mathcal{S}} \land temporary(r) \land external(r, l) \land invalid(r, l.r) \land Ca(l) \uparrow\}$.

9.5 Class Grime

Class Grime is the build-up of unnecessary (given the specification of a pattern) methods and fields within the classes of a pattern instance. Such a build-up implies a violation of one or more of the following design principles:

- The YAGNI (You Ain't Gonna Need It) principle – you should not add functionality until you are going to need it [88].
- The Single Responsibility Principle (SRP) – a class should only have responsibility for a single part of the functionality of the software, and the Class should fully encapsulate this responsibility [183].
- The Interface Segregation Principle (ISP) – no client should depend on those methods it does not use [183].
- High class cohesion – the responsibilities of the methods within a class should be highly related and support the responsibility of that Class [286].

Each of these principles speaks to the cohesion of a class. High cohesion is a fundamental object-oriented principle, in which a highly cohesive class is a class in which its member

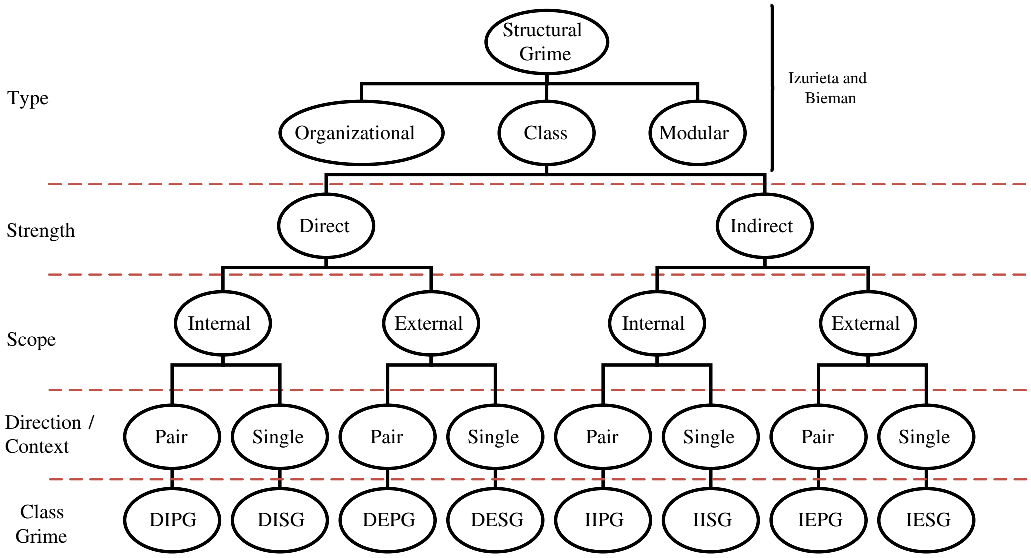


Figure 9.7: The extended Class Grime taxonomy.

fields and methods work together to address a single primary responsibility of the Class. Using cohesion as the fundamental property, we have divided Class Grime into eight specific subtypes, as depicted in Figure 9.7. The following sections further explain this division.

9.5.1 Class Cohesion

Cohesion describes how well constructed a class is [39]. The higher the cohesion of a class, the closer aligned its internal components are towards a common goal. In design pattern realizations, the classes should represent individual responsibilities of the pattern, and each Class should have high cohesion. Thus cohesion provides a basis to determine whether a design pattern realization's classes contain Class Grime.

Strength Strength indicates how a class' methods access its local attributes. The method of access can be either direct (methods directly access attributes) or indirect (attribute accessed through the use of accessor/mutator methods). Figure 9.8 depicts each method of access. In this figure, the unbroken lines between attributes (rectangles) and methods

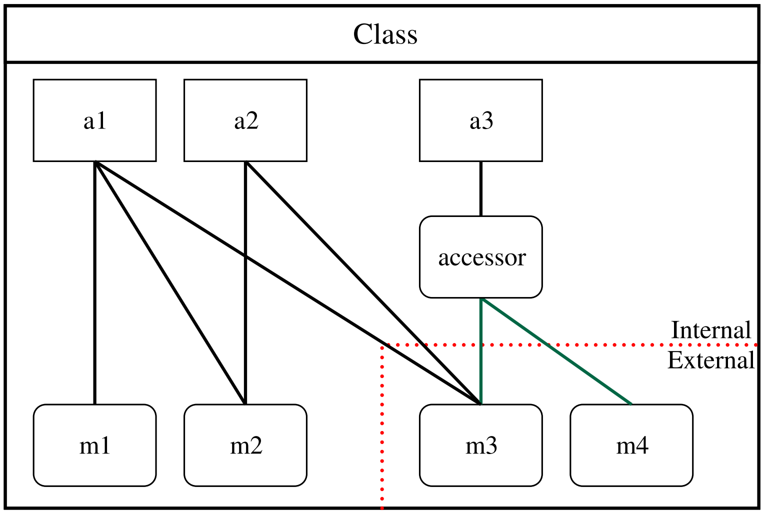


Figure 9.8: An example of the IESG type of Class Grime.

(rounded rectangles) are direct relationships, and the lines broken by a smaller rounded rectangle are indirect relationships. Direct attribute use provides a stronger but more brittle relationship between the method and attribute, causing issues when attempting to refactor by moving the attribute. Whereas indirect attribute use implies a flexible and weaker relationship between the method and attribute, but one which is more amenable to refactoring.

Scope In the context of pattern classes, the scope can either be internal or external. Internal refers to attributes access by a local method (or local method pair, depending on context) defined by the pattern specification. External refers to attributes accessed by at least one local method (or local method pair) not defined by the pattern specification. In Figure 9.8, the internal/external division is shown by the dashed red line dividing the Class into methods/attributes associated with the pattern specification of that Class and those methods/attributes not specified by the pattern specification. Thus providing a means to distinguish between identification of attributes or methods that obscure the pattern implementation by reducing overall class cohesion.

Context The context refers to the types of relationships taken into account by surrogate metrics used to measure cohesion. The majority of cohesion metrics take one of two perspectives: single-method use or method pair use of attributes [39]. In order to satisfy the strength, scope, and context aspects of the taxonomy, we have selected two metrics. The first is Tight Class Cohesion (TCC) [30], which measures the cohesion of a class by looking at pairs of methods with attributes in common, and it can handle both indirect and direct attribute use. The second is the Ratio of Cohesive Interactions (RCI) [40] metric, which measures the cohesion of a class by looking at how particular methods use attributes, and it can handle both indirect and direct attribute use.

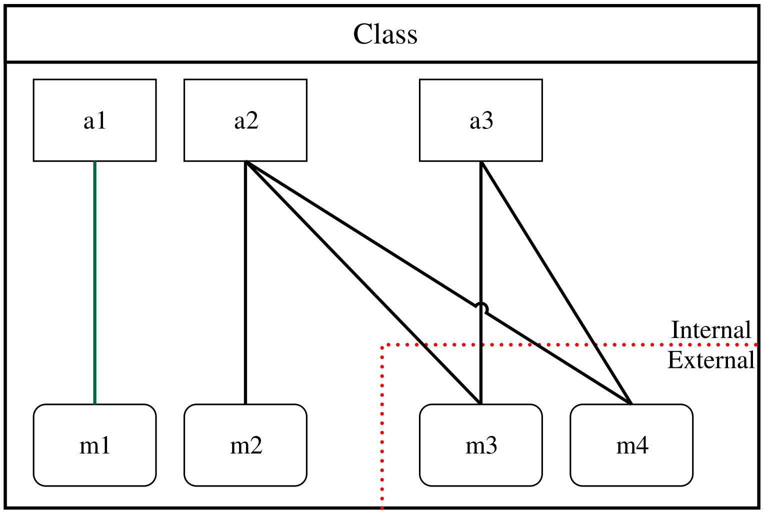


Figure 9.9: Example of DISG.

9.5.2 Class Grime Example

In Figure 9.9 we can see an example of Direct Internal Single Grime (DISG). The figure is the representation of a pattern class. Where the dashed red line indicates those methods that are internal (specified by the pattern) and those external (not specified by the pattern).

The figure depicts a case of DISG, as there exists a method, $m1$, allowed by the pattern specification that directly uses an attribute, $a1$, but no other method uses that attribute. This instance of DISG indicates that there could be an unintentional secondary responsibility associated with this Class or a misunderstanding on the part of the developer in the pattern's implementation.

9.5.3 Class Grime Categories

Using the notions of Strength, Scope, and Context, we define eight specific categories of Class Grime. In the following, we describe these categories and provide a formal set definition for each based on the framework from Section 9.3.

Direct Internal Pair Grime (DIPG) The set of method pairs from the type t which are internal to pattern realization l, form a direct relationship to the same attribute a, the set of methods calling m_1 is empty, and for which the relationships decrease the TCC of t. Formally, this defines the set $\{(m_1, m_2) \,|\, t \in \mathcal{T}_\mathcal{S} \wedge m_1, m_2 \in \mathcal{M}_t \wedge r_1 = (m_1, a) \wedge r_2 = (m_2, a) \wedge a \in \mathcal{A}_t \wedge direct(r_1, r_2) \wedge internal(m_1, m_2) \wedge calls(m_1) = \varnothing \wedge TCC(t) \downarrow\}$.

methods and attributes within the classes of a pattern. DIPG can be observed when $(m_i,\, m_j) \in Internal$, $(r_i,\, r_j) \in Direct$, $r_i.attribute = r_j.attribute$, and TCC decreases.

Direct Internal Single Grime (DISG) The set of methods from the type t which are internal to pattern realization l, forms a direct relationship to attribute a, the set of methods calling m is empty, and for which the relationships decrease the RCI of t. Formally, this defines the set $\{m \,|\, t \in \mathcal{T}_\mathcal{S} \wedge m \in \mathcal{M}_t \wedge r = (m, a) \wedge a \in \mathcal{A}_t \wedge direct(r) \wedge internal(m) \wedge calls(m) = \varnothing \wedge RCI(t) \downarrow\}$.

Direct External Pair Grime (DEPG) The set of method pairs from the type t which are external to pattern realization l, form a direct relationship to the same attribute a, the set of methods calling m_1 is empty, and for which the relationships decrease the TCC of t. Formally, this defines the set $\{(m_1, m_2) \,|\, t \in \mathcal{T}_\mathcal{S} \wedge m_1, m_2 \in \mathcal{M}_t \wedge r_1 = (m_1, a) \wedge r_2 = (m_2, a) \wedge a \in \mathcal{A}_t \wedge \mathit{direct}(r_1, r_2) \wedge \mathit{external}(m_1, m_2) \wedge \mathit{calls}(m_1) = \varnothing \wedge \mathit{TCC}(t) \downarrow\}$.

Direct External Single Grime (DESG) The set of methods from the type t which are external to pattern realization l, forms a direct relationship to attribute a, the set of methods calling m is empty, and for which the relationships decrease the RCI of t. Formally, this defines the set $\{m \,|\, t \in \mathcal{T}_\mathcal{S} \wedge m \in \mathcal{M}_t \wedge r = (m, a) \wedge a \in \mathcal{A}_t \wedge \mathit{direct}(r) \wedge \mathit{external}(m) \wedge \mathit{calls}(m) = \varnothing \wedge \mathit{RCI}(t) \downarrow\}$.

Indirect Internal Pair Grime (IIPG) The set of method pairs from the type t which are internal to pattern realization l, form an indirect relationship to the same attribute a, the set of methods calling m_1 is empty, and for which the relationships decrease the TCC of t. Formally, this defines the set $\{(m_1, m_2) \,|\, t \in \mathcal{T}_\mathcal{S} \wedge m_1, m_2 \in \mathcal{M}_t \wedge r_1 = (m_1, a) \wedge r_2 = (m_2, a) \wedge a \in \mathcal{A}_t \wedge \mathit{indirect}(r_1, r_2) \wedge \mathit{internal}(m_1, m_2) \wedge \mathit{calls}(m_1) = \varnothing \wedge \mathit{TCC}(t) \downarrow\}$.

Indirect Internal Single Grime (IISG) The set of methods from the type t which are internal to pattern realization l, forms an indirect relationship to attribute a, the set of methods calling m is empty, and for which the relationships decrease the RCI of t. Formally, this defines the set $\{m \,|\, t \in \mathcal{T}_\mathcal{S} \wedge m \in \mathcal{M}_t \wedge r = (m, a) \wedge a \in \mathcal{A}_t \wedge \mathit{indirect}(r) \wedge \mathit{internal}(m) \wedge \mathit{calls}(m) = \varnothing \wedge \mathit{RCI}(t) \downarrow\}$.

Indirect External Pair Grime (IEPG) The set of method pairs from the type t which are external to pattern realization l, form a indirect relationship to the same attribute a, the set of methods calling m_1 is empty, and for which the relationships decrease the TCC of t. Formally, this defines the set $\{(m_1, m_2) \,|\, t \in \mathcal{T}_\mathcal{S} \wedge m_1, m_2 \in \mathcal{M}_t \wedge r_1 = (m_1, a) \wedge r_2 = (m_2, a) \wedge a \in \mathcal{A}_t \wedge \mathit{indirect}(r_1, r_2) \wedge \mathit{external}(m_1, m_2) \wedge \mathit{calls}(m_1) = \varnothing \wedge \mathit{TCC}(t) \downarrow\}$.

Indirect External Single Grime (IESG) The set of methods from the type t which are external to pattern realization l, forms an indirect relationship to attribute a, the set of methods calling m is empty, and for which the relationships decrease the RCI of t. Formally, this defines the set $\{m \,|\, t \in \mathcal{T_S} \wedge m \in \mathcal{M}_t \wedge r = (m, a) \wedge a \in \mathcal{A}_t \wedge indirect(r) \wedge external(m) \wedge calls(m) = \varnothing \wedge RCI(t) \downarrow\}$.

9.6 Organizational Grime

Organizational Grime is the accumulation of design pattern grime due to the allocation of pattern classes to packages, namespaces, or modules within a software system. The development of the Organizational Grime hierarchy comes from the following design principles:

- The Acyclic Dependencies Principle (ADP) – Dependencies between packages should not form cycles [183].
- The Stable Dependencies Principle (SDP) – Depend in the direction of stability [183].
- The Stable Abstractions Principle (SAP) – Abstractness should increase with stability [183].
- The Common Closure Principle (CCP) – Classes in a package should be closed to the same kinds of changes [183].
- The Common Reuse Principle (CRP) – Classes in the same package should be reused together [183].

These principles speak to both the coupling between packages and the cohesion within a package. Using the properties of package coupling and cohesion, we have divided package grime into twelve specific subtypes, as depicted in Figure 9.10. The following sections further explain this division.

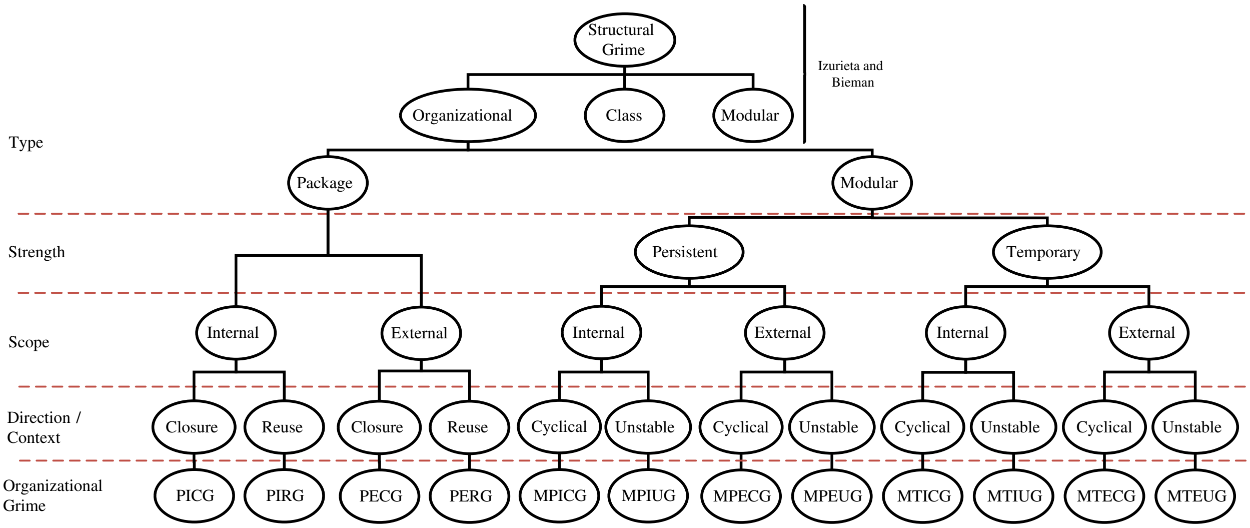


Figure 9.10: Organizational Grime taxonomy.

9.6.1 Package Cohesion

Package cohesion is used to develop the *Package* subtype of Organizational Grime, as seen in Figure 9.10. Here we consider only the scope and context properties. Together these concepts are used to form the Package branch of Organizational Grime.

Scope The scope can be either internal or external, both referring to the addition of a new class or type to a package. If the new class or type is also a member of the pattern under consideration, then its scope is internal; otherwise, it is external.

Context/Direction The context property takes the form of either closure or reuse. Closure here indicates that the new class or type fits within the package by being closed to similar changes as the other classes. Reuse indicates that we are concerned with how well a class integrates into its containing package based on how tightly it couples with the remaining classes.

The closure quality of a package is measured using the *CohesionQ* metric defined by Abdeen et al. [3]. The reuse quality of a package is measured using the *CouplingQ* metric defined by Abdeen et al. [3]. These metrics are calculated using the following formulas:

$$CohesionQ(p) = \frac{|p_{Int.D}|}{|p_D|} \tag{9.1}$$

$$CouplingQ(p) = 1 - \frac{|p_{Pro.P} \cup p_{Cli.P}|}{|p_D|} \tag{9.2}$$

Common to both $CohesionQ$ and $CouplingQ$, the set p_D is the set of all dependencies in and out of package p. Formally we define p_D as $p_D \equiv \forall_{p_i \in \mathcal{P}_S}((p, p_i) \in \mathcal{D} \vee (p_i, p) \in \mathcal{D}) \wedge p \neq p_i$. The set $p_{Int.D}$ is the set of internal dependencies (connections) between classes within package p. Formally we define $p_{Int.D}$ as $p_{Int.D} \equiv \forall_{\{t_i,t_j\} \in p}(t_i, t_j) \in \mathcal{K}$. Finally, $p_{Pro.P}$ is the set of packages which a package, p, depends on, and $p_{Cli.P}$ is the set of packages a package, p, is depended on by.

9.6.2 Package Coupling

Package coupling is the basis of the *Modular* subtype of Organizational Grime, c.f. Figure 9.10. Together these concepts are used to form the Modular branch of Organizational Grime.

Strength Here we consider three properties of coupling between packages. The first is the *strength*, which can be either *persistent* or *temporary.* Persistent couplings are those created by inheritance, realization, associations (including aggregation and composition), temporary are the remaining dependencies such as use dependencies.

Scope The next property is *scope*, which can be either *internal* or *external.* Internal couplings are those that are caused by classes within the same pattern but spread across packages. External couplings are relationships between packages that are caused by external classes interacting with pattern classes across packages.

Context/Direction The final property is *direction/context*. Here we are looking at how the coupling affects cyclic dependencies between packages, *cyclical* value, and the flow of stability between packages, *unstable* value. When we are considering whether the new dependency will cause cycles between packages we are in the cyclical context, and when we are considering the flow of dependencies towards stability, then we are in the unstable context. Cycles in the package dependency graph can be evaluated using the algorithm for enumerating cycles in a directed graph developed by Liu and Wang [120]. Instability of packages is measured using Martin's *Normalized Distance (D')* metric [183], which is calculated as follows:

$$D' = |A + I - 1| \tag{9.3}$$

$$A = \frac{N_a}{N_c} \tag{9.4}$$

$$I = \frac{C_e}{C_a + C_e} \tag{9.5}$$

Equation 9.1 represents the distance from the main sequence and has values in the range [0,1] with 0, indicating that the package is on the main sequence. Packages with values away from zero tend to be less maintainable and more sensitive to change. To complete the computation of D', we need to calculate several other values. The first is the *Abstractness (A)* metric, which is a measure of the level of abstraction in a given package. The A metric is calculated as the ratio of the *number of abstract classes* (N_a) to the *number of classes* (N_c) in a package. The second value necessary for calculating D' is the *Instability (I)* metric, which measures the level of instability of a given package. The I metric is the ratio of the number of efferent couplings (C_e) of a package to the sum of the number of efferent (C_e) and afferent couplings (C_a) of a package.

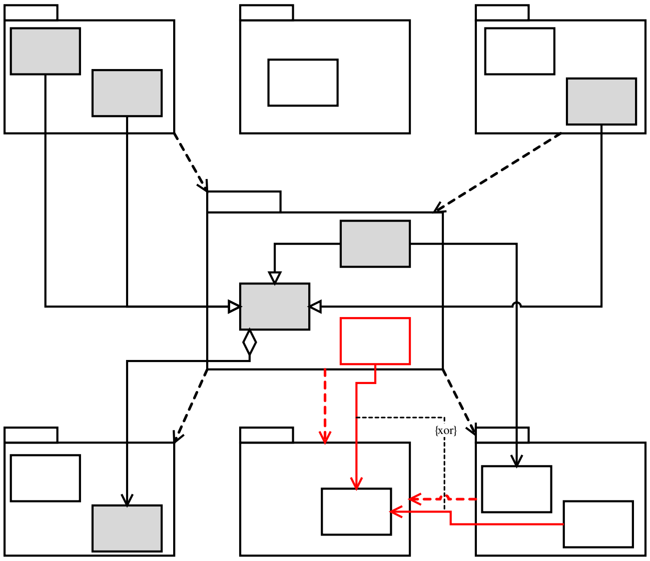


Figure 9.11: Example of PECG.

9.6.3 Organizational Grime Example

In Figure 9.11 we can see an example of Package External Closure Grime (PECG). In this diagram, those classes which are a part of the pattern as grey rectangles, and classes not part of the pattern as white rectangles. The figure depicts dependencies between packages using a dashed line with an open arrowhead pointing in the direction of the dependency, while other relationships follow the usual UML syntax. The red items mark the causes of grime. Here there is an XOR relation between either an existing class or new class (both external to the pattern) interacting with a non-pattern class but increasing the number of packages reachable from pattern packages.

9.6.4 Organizational Grime Categories

Using the notions of Strength, Scope, and Context, we define twelve specific categories of Organizational Grime. In the following, we describe these categories and provide a formal set definition for each based on the framework from Section 9.3.

Package External Closure Grime (PECG) The set of classes which are internal to a pattern bearing package, external to a pattern realization, and whose connections decrease the cohesion quality (and thus the common closure) of the package. Formally, this defines the set $\{c \,|\, l \in \mathcal{L} \land p \in \mathcal{P}_l \land \mathit{external}(c, l) \land \mathit{internal}(c, p) \land \mathit{CohesionQ}(q) \downarrow\}$.

Package External Reuse Grime (PERG) The set of classes which are internal to a pattern bearing package, external to a pattern realization, and whose connections decrease the coupling quality (and thus the common reuse) of the package. Formally, this defines the set $\{c \,|\, l \in \mathcal{L} \land p \in \mathcal{P}_l \land \mathit{external}(c, l) \land \mathit{internal}(c, p) \land \mathit{CouplingQ}(q) \downarrow\})$.

Package Internal Closure Grime (PICG) The set of classes which are internal to a pattern bearing package, internal to a pattern realization, and whose connections decrease the cohesion quality (and thus the common closure) of the package. Formally, this defines the set $\{c \,|\, l \in \mathcal{L} \land p \in \mathcal{P}_l \land \mathit{internal}(c, l) \land \mathit{internal}(c, p) \land \mathit{CohesionQ}(q) \downarrow\}$.

Package Internal Reuse Grime (PIRG) The set of classes which are internal to a pattern bearing package, internal to a pattern realization, and whose connections decrease the coupling quality (and thus the common reuse) of the package. Formally, this defines the set $\{c \,|\, l \in \mathcal{L} \land p \in \mathcal{P}_l \land \mathit{internal}(c, l) \land \mathit{internal}(c, p) \land \mathit{CouplingQ}(q) \downarrow\})$.

Modular Persistent External Cyclical Grime (MPECG) The set of dependencies between a pattern-bearing package and a non-pattern-bearing package, created due to a persistent connection between classes, and cause a cyclic dependency between the connected packages. Formally, this defines the set $\{(d, e) \,|\, \mathit{persistent}((d, e)) \land (\mathit{external}(d, \mathcal{P}_r) \lor \mathit{external}(e, \mathcal{P}_r)) \land r \in \mathcal{R} \land \mathit{cycle}(d, e)\}$.

Modular Temporary External Cyclical Grime (MTECG) The set of dependencies between a pattern-bearing package and a non-pattern-bearing package, created due to a temporary connection between classes, and cause a cyclic dependency between the connected packages. Formally, this defines the set $\{(d,e)\,|\,\mathit{temporary}((d,e)) \wedge (\mathit{external}(d,\mathcal{P}_r) \vee \mathit{external}(e,\mathcal{P}_r)) \wedge r \in \mathcal{R} \wedge \mathit{cycle}(d,e)\}$.

Modular Persistent Internal Cyclical Grime (MPICG) The set of dependencies between packages which contain types that are internal to a pattern instance, created due to a persistent connection between classes, and cause a cyclic dependency between the connected packages. Formally, this defines the set $\{(d,e)\,|\,\mathit{persistent}((d,e)) \wedge \mathit{internal}(d,e,\mathcal{P}_r) \wedge r \in \mathcal{R} \wedge \mathit{cycle}(d,e)\}$.

Modular Temporary Internal Cyclical Grime (MTICG) The set of dependencies between packages which contain types that are internal to a pattern instance, created due to a temporary connection between classes, and cause a cyclic dependency between the connected packages. Formally, this defines the set $\{(d,e)\,|\,\mathit{temporary}((d,e)) \wedge \mathit{internal}(d,e,\mathcal{P}_r) \wedge r \in \mathcal{R} \wedge \mathit{cycle}(d,e)\}$.

Modular Persistent External Unstable Grime (MPEUG) The set of dependencies between pattern-bearing package and a non-pattern-bearing package, created due to a persistent connection between classes, and the source side of the dependency is more stable than the target side of the dependency. Formally, this defines the set $\{(d,e)\,|\,\mathit{persistent}((d,e)) \wedge ((\mathit{external}(d,\mathcal{P}_r) \wedge \mathit{internal}(e,\mathcal{P}_r) \wedge I(e) \leq I(d)) \vee (\mathit{internal}(d,\mathcal{P}_r) \wedge \mathit{external}(e,\mathcal{P}_r) \wedge I(d) \leq I(e)))\}$.

Modular Temporary External Unstable Grime (MTEUG) The set of dependencies between pattern-bearing package and a non-pattern-bearing package, created due to a temporary connection between classes, and the source side of the dependency is more stable than the target side of the dependency. Formally, this defines the set $\{(d,e)\,|\,\mathit{temporary}((d,e)) \wedge$

$((external(d, \mathcal{P}_r) \wedge internal(e, \mathcal{P}_r) \wedge I(e) \leq I(d)) \vee (internal(d, \mathcal{P}_r) \wedge external(e, \mathcal{P}_r) \wedge I(d) \leq I(e)))\}$.

Modular Persistent Internal Unstable Grime (MPIUG) The set of dependencies between packages which contain types that are internal to a pattern instance, created due to a persistent connection between classes, and the source side of the dependency is more stable than the target side of the dependency. Formally, this defines the set $\{(d, e) \,|\, persistent((d, e)) \wedge internal(d, e, \mathcal{P}_r) \wedge I(d) \leq I(e)\}$.

Modular Temporary Internal Unstable Grime (MTIUG) The set of dependencies between packages which contain types that are internal to a pattern instance, created due to a temporary connection between classes, and the source side of the dependency is more stable than the target side of the dependency. Formally, this defines the set $\{(d, e) \,|\, temporary((d, e)) \wedge internal(d, e, \mathcal{P}_r) \wedge I(d) \leq I(e)\}$.

9.7 Conclusion

This chapter presented the enhanced design pattern grime taxonomy. We detailed the exact methodology used to develop enhanced Class and Organizational Grime taxonomies. Furthermore, we connected these taxonomies to the underlying software engineering principles they are codifying as well as the metrics that lead to the definition of their detection strategies. The development of these taxonomies, along with the existing modular grime taxonomy, leads directly into the design of both injection and detection strategies for different grime types.

CHAPTER TEN

EXPERIMENTATION: THE EFFECTS OF GRIME ON MAINTAINABILITY AND TECHNICAL DEBT

It's hard enough to find an error in your code when you're looking for it; its even harder when you've assumed your code is error-free.

–Steve McConnell

10.1 Introduction

The ability to predict the effects on software quality that a design or implementation decision will have on the underlying software is of great concern. Also of great concern is the effects that design and management decisions will of on a systems incurred technical debt. Predicting the effects that design pattern grime would have for a given design or implementation is no different. The first step in this direction is to understand the relationship between grime and quality and grime and technical debt before predicting its effects. Thus, this chapter explores the effects of design pattern grime on software systems' maintainability and technical debt. The goal of these experiments, stated in Section 1.1.1, are restated here for the reader's convenience:

RG2: Analyze design pattern instances afflicted with design pattern grime for the purpose of evaluation with respect to the ISO/IEC 25010 Maintainability subcharactersitics [126], from the perspective of researchers, in the context of generated Java™ design pattern instances.

RG3: Analyze design pattern instances afflicted with grime for the purpose of evaluation with respect to the Technical Debt Principal and Interest, from the perspective of researchers, in the context of generated Java™ design pattern instances.

This goal leads to our main questions of interest and their corresponding rationales:

RQ2: How does each type of grime affect software product maintainability?

Rationale: Evaluate the assertion that as grime builds up in a pattern instance or software system, it will negatively affect the software or pattern instance's maintainability.

RQ3: How does each type of grime affect a software product's technical debt estimate?

Rationale: Evaluate the assertion that as grime builds up in a pattern instance or software system, it will increase the technical debt principal and interest.

This chapter is organized as follows. Section 10.2 describes the experimental methods used and the corresponding experimentation plan, including the data collection and analysis procedures. Section 10.3 describes the outcome of the execution of the experiment, including sample characteristics, data preparation steps, data collection performed, and any deviations from the experiment plan. Section 10.4 describes the results and analysis conducted following the analysis procedures. Section 10.5 discusses the analysis results and their interpretation within this study's context and prior work. Section 10.6 concludes this study.

10.2 Methods

This section describes the experimental methods used to answer **RQ2** and **RQ3**. Towards these goals this section contains subsections that further refine these research questions and identify their necessary metrics. Additionally, we describe the experimental designs, data collection procedures, and analysis procedures used when answering the research questions. Finally, we end this section with a discussion concerning the validity of the overall approach.

10.2.1 Refined Research Questions and Metrics

Following the GQM approach, we begin the experimental design with a refinement of the questions **RQ2** and **RQ3** into a series of directly answerable questions, their underlying rationale, and a set of metrics defined to facilitate answering these questions. The refined questions are as follows:

RQ2.1: How does each type of Grime affect design pattern quality for each of the selected Maintainability sub-characteristics?

Rationale: Evaluate the assertion that as grime builds up in a pattern instance or software system, it will negatively affect Maintainability.

RQ2.2: What level of injection severity affects a change in design pattern quality for each of the Maintainability sub-characteristics?

Rationale: Evaluate the assertion that grime affects Maintainability at all severity levels.

RQ2.3: What is the difference between the effects of the grime types and their subtypes on maintainability sub-characteristics?

Rationale: Evaluate the assertion that each grime type or each grime subtype affects Maintainability similarly.

RQ3.1: How does each type of grime affect design pattern technical debt principal and interest?

Rationale: Evaluate the assertion that as grime builds up in a pattern instance or a software system, it will affect the accumulation of the software or pattern technical debt principal.

RQ3.2: What level of grime severity affects a change in design pattern technical debt principal and interest?

Rationale: Evaluate the assertion that as grime builds up in a pattern instance or a software system, it will affect the accumulation of the software or pattern technical debt principal.

RQ3.3: What is the difference between the effects of the grime types and their subtypes on technical debt principal and interest?

Rationale: Evaluate the assertion that grime affects technical debt interest and principal at all severity levels.

Maintainability, as defined in the ISO/IEC 25010 Quality Model, is composed of following five sub-characteristics: Analyzability, Testability, Modifiability, Modularity, and Reusability. These are the sub-characteristics referenced in **RQ2.1 – 2.3**. The definition of these and other pertinent metrics necessary to answer questions **RQ2.1 – 2.3** and **RQ3.1 – 3.3** are defined as follows:

M2.1: *Analyzability* – "degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified" [126]. Analyzability will be measured using a modified implementation of the SIG Maintainability [115] quality model, which considers Analyzability to be a continuous value falling in the range $[0, 5]$.

M2.2: *Testability* – "degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met" [126]. This will be measured using a modified implementation of the SIG Maintainability Model [115] quality model, which considers Testability to be a continuous value falling in the range $[0, 5]$.

M2.3: *Modifiability* – “degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality” [126]. This will be measured using a modified implementation of the SIG Maintainability Model [115] quality model, which considers Modifiability to be a continuous value falling in the range $[0, 5]$.

M2.4: *Modularity* – “degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components” [126]. This will be measured using a modified implementation of the SIG Maintainability Model [115] quality model, which considers Modularity to be a continuous value falling in the range $[0, 5]$.

M2.5: *Reusability* – “degree to which an asset can be used in more than one system, or in building other assets” [126]. This will be measured using a modified implementation of the SIG Maintainability Model [115] quality model, which considers Reusability to be a continuous value falling in the range $[0, 5]$.

M2.6: *Injection Severity* (IS) – An indicator of the severity of grime affliction for a given design pattern instance. This metric is measured using the mapping defined as follows:

$$GS(p) = \begin{cases} GP(p) = 0\% & 0\,\text{(Control)} \\ GP(p) \leq 15\% & 1 \\ GP(p) \leq 30\% & 2 \\ GP(p) \leq 45\% & 3 \\ GP(p) \leq 60\% & 4 \\ GP(p) \leq 75\% & 5 \end{cases}$$

Where $GP(p)$ is the percentage of grime affecting a pattern instance. $GP(p)$ is calculated as the ratio of pattern instance members bound to a role defined by the associated pattern RBML and affected by grime to the total number of pattern instance members bound to a role. For a measured value of GS, v, where $v \in \mathbb{N}^+$ and $v \in [0, 5]$, and is measured on an ordinal scale.

M2.7: *Pattern Type* (PT) – the pattern type name for a given pattern instance. This metric is measured on a *nominal* scale, with each measured value being one of the following: (Object) Adapter, Bridge, Chain of Responsibility, Command, Composite, Decorator, Flyweight, Factory Method, Observer, Prototype, Proxy, Singleton, State, Strategy, Template Method, or Visitor. These values are limited to those reported by the Pattern4 tool.

M2.8: *Injection Type* (IT) – the grime type for the specific type of grime affecting a given pattern instance. This metric is measured on a *nominal* scale, with each measured value being one of the 26 grime type acronyms identified in Chapter 9.

M3.1: *Technical Debt Principal* – A measure of the man-months or monetary value of the effort required to remediate (via refactoring) the issues identified as technical debt within a software system. This metric is measured using Nugroho et al.'s method as described in Section 2.2.3. This method measures TD Principal as a continuous positive value with units in man-months.

M3.2: *Technical Debt Interest* – A measure of the effort to remediate the compounding effect of unremediated technical debt on the maintenance of a software system. This metric is measured using Nugroho et al.'s approach as described in Section 2.2.3. This method measures TD Interest as a continuous positive value with units in man-months.

Using these basic metrics we next describe the experimental designs.

10.2.2 Experimental Design

This study is further decomposed into seven separate experiments. Each experiment considers one of the corresponding quality attributes: five for the maintainability subcharacteristics (Analyzability, testability, Modifiability, modularity, and Reusability) and two for the technical debt components (principal and interest). Each experiment uses a three-factor factorial design. This design was selected to accommodate for the potential interactions between the independent variables. These variables include Pattern Type, Injection Severity, and Injection Type. The dependent variable in each experiment is the corresponding quality attributes. This design will require 2496 experimental subjects per replication to account for each combination of pattern type, grime type, and injection severity level.

We generate each replication's pattern instances using the design pattern generation technique described in Section 4.4.3. This approach frees us from any restrictions that a lack of experimental subjects would impose. The following subsection describes the method used to collect this data.

10.2.3 Data Collection

The following describes the data to be collected, the collection process, and how this data is stored. For each instance under study, we extract the instance identifier, the grime type injected, the grime severity level for the grime injected, the pattern type, and the change in the quality attribute of concern (between pattern generation and grime injection). The ReportGenerator extracts this information from the PatternInstance, Findings, and Measures tables of the ArcDb. Once extracted, the ReportGenerator generates a table, similar to the example shown in Table 10.1, with the following specifications:

Table 10.1: Example data collection table for grime and quality experiment.

ID	PT	IT	IS	ΔQA
0	Singleton	DIPG	2	0.95
⋮	⋮	⋮	⋮	⋮

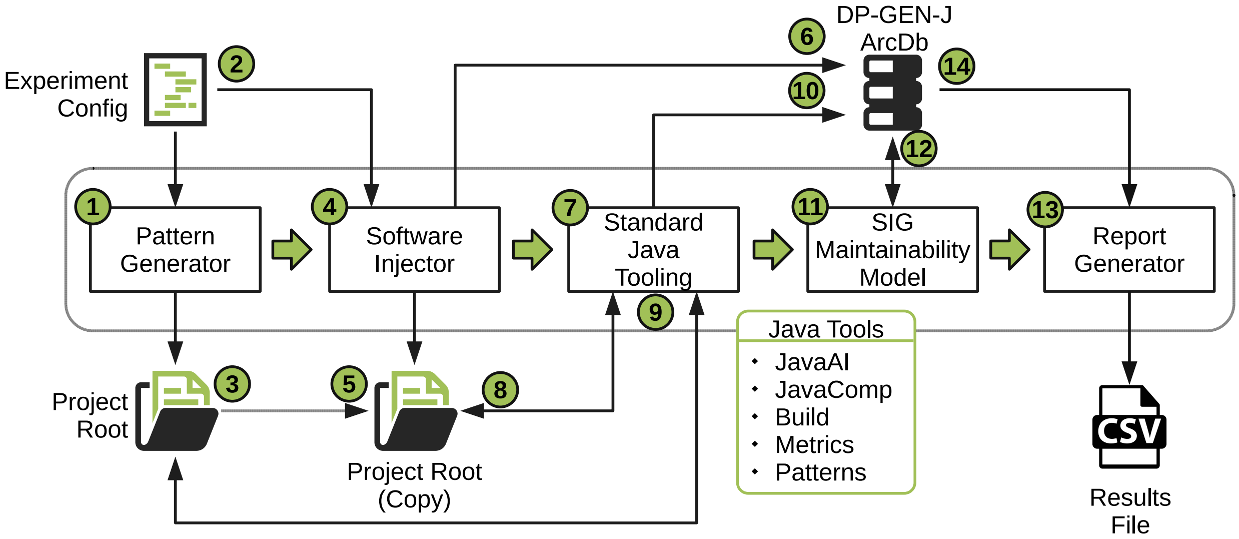


Figure 10.1: Grime effect on Quality data collection process.

- Each row of the table represents a single design pattern instance.
- The first column of the table is the identifier representing a specific pattern instance.
- The second column of the table represents the design pattern type.
- The third column of the table represents the type of grime injected.
- The fourth column of the table represents the severity rating for the injected grime.
- The fifth column of the table represents the change in value for the quality attribute which is the subject of the experiment in question.

Figure 10.1 depicts an overview of the data collection process. This process follows the path indicated by the numbers encircled in green, as follows. 1.) The Arc system executes the *Pattern Generator*, 2.) which utilizes the *ExperimentalConfig* to define which pattern types to generate and the number of instances needed. 3.) This process results in generating a project per instance as a physical project folder. 4.) The workflow shifts to the *SoftwareInjector*, which uses the *ExperimentalConfig* to determine the type and severity of grime to inject. 5.) The *SoftwareInjector* copies the physical project and injects the specified grime into this copy. 6.) Once injected, both copies of the project are then analyzed using the standard java tooling. 7.) The standard java tooling (which includes the items indicated by the "Java Tools" key on the diagram) stores its results in the ArcDb. 8.) The SIG Maintainability Model quality analysis executes across all systems in the database and stores its resulting measures in the ArcDb. 9.) The report generator uses the stored measures to construct the data table according to the specification previously described. These results, once gathered, are then analyzed (for each experiment separately) using the following analysis procedures.

10.2.4 Analysis Procedures

This section describes the analysis models and procedures used for these experiments. The analysis of these experiments will proceed from the data collection forward, as follows:

1. A size analysis conducted to determine the number of replications needed to achieve the statistical power required.

2. Data collection occurs according to the data collection procedure.

3. Descriptive statistics are gathered and recorded based on the collected data.

4. Evaluate the ANOVA assumptions, as follows:

   (a) Evaluate the Homogeneity of Variance assumption visually using a Residuals vs. Fitted values plot and analytically using Levene's Test [162].

   (b) Evaluate the Normality of the sample population visually using a Normal Q-Q plot and analytically using the Anderson-Darling normality test [14].

   (c) For each experiment, following the data collection procedure, the data meets the assumption that the samples are drawn independently of one another.

   (d) For each experiment, following the data collection procedure, the observations are sampled randomly and independently of one another.

5. If any violations of the assumptions are detected, attempts will be made to address or explain these issues. If there are no violations, or they have been addressed, then the ANOVA analysis will continue. If attempts to address violations are unsuccessful, then a permutation F-test will be conducted instead.

6. If a significant difference is detected (F-test with $p < 0.05$), then the following will occur:

   (a) Evaluation of the interaction effects in the model. If present, the significant interactions will be noted and the interactions plotted for further evaluation. Additionally, if such interaction effects are detected, then we will not explore the main effects further.

   (b) In the case of a lack of significant interaction effects, the main effects will be considered. Additionally, we will conduct a multiple comparison procedure between the means for all treatment levels and the results recorded. Finally, the execution of the pre-planned contrasts will be conducted and the results recorded.

We will conduct this analysis using the R Project for statistical computing version 4.1.1 and various R modules.

10.2.4.1 Size Analysis A design size analysis determines the number of replications required to achieve the analytical power necessary to detect a difference. There are four required values to conduct this analysis: (i) an estimate of the smallest relative distance between means, (ii) an estimate of the standard deviation, (iii) the alpha level, and (iv) the power level desired. The latter two are known, but we must estimate the former two based on either prior knowledge or pilot study results.

Given the lack of prior knowledge, we have selected to conduct a small pilot study. This study follows the same approach as the primary study. However, with the following restrictions: (i) we will only generate instances for two patterns: Singleton and State, and (ii) we will generate 156 instances per pattern (each injection type for each level of severity). The data collected will be enough to estimate both the smallest relative distance between means and the standard deviation. We will conduct the design size analysis and generate the pattern instances for each experiment using this information.

10.2.4.2 ANOVA/Permutation F-test For our design and subsequent analysis we have elected to utilize an ANOVA model (and if not possible a permutation F-test). The model for analysis is as follows:

$$y_{ijkl(m)} = \mu + \tau_i + \beta_j + \gamma_k + (\tau\beta)_{ij} + (\tau\gamma)_{ik} + (\beta\gamma)_{jk} + (\tau\beta\gamma)_{ijk} + \epsilon_{ijkl}$$

In this model:

- y_{ijkl} represents quality attribute (QA) of concern
- τ_i represents the effect of the i^{th} pattern type (PT)
- β_j the effect of the jth injection type (IT)
- γ_k the effect of the kth injection severity (IS)

- $(\tau\beta)_{ij}$, represents the effects of the two-factor interaction of PT and IT
- $(\tau\gamma)_{ik}$, represents the effects of the two-factor interaction of PT and IS
- $(\beta\gamma)_{jk}$, represents the effects of the two-factor interaction of IT and IS
- $(\tau\beta\gamma)_{ijk}$, represents the effects of the three-factor interaction of PT, IT, and IS
- ϵ_{ijkl} represents the random error of the l^{th} observation of the $(i, j, k)^{th}$ treatment

Using this model, the ANOVA/permutation F-test analysis determines only if there is any difference in mean change in the quality attribute of concern due to any treatment.

10.2.4.3 Interaction Effect The evaluation of the interaction effects will first determine whether there is any evidence of a three-way interaction. If this is the case, then the remaining interaction effects, main effects, multiple comparisons, and pre-planned contrasts will not be considered. If there is weak or no evidence of such an interaction, we will review each two-factor interaction. Next, suppose there is evidence suggesting the presence of two-factor interactions. In that case, we will only consider main effects, multiple comparisons, or pre-planned contrasts for those effects not contributing to interactions with supporting evidence. Finally, if weak or no evidence supports any interactions, we will move forward in considering the main effects, multiple comparisons, and pre-planned contrasts as described in the following subsections.

10.2.4.4 Main Effects, Multiple Comparisons and Pre-planned Contrasts The main effects, τ_i, β_j, and γ_k, representing the effect that pattern type, injection type, and injection severity have on the mean change in quality attribute, will be evaluated as part of the ANOVA/Permutation F-test. If there is strong evidence for the main effects, we will execute multiple comparison procedures and pre-planned contrasts. In such a case, we will execute multiple comparisons with a Bonferroni corrected p-value [35]. Except for injection severity, we will perform an all-pairs comparison. For injection severity we will utilize a comparison versus control approach (such as Dunnett's test [68] or Steel's test [249]). Evaluating these multiple comparisons will provide detailed insight into the specific differences in the effects of different grime types and grime severity levels. Thus, providing deeper insights into answering questions **RQ2.1** − **RQ2.2** and **RQ3.1** − **RQ3.2**.

To answer questions **RQ2.3** and **RQ3.3** we will conduct a series of contrast analyses to evaluate the following statistical hypotheses related to the research questions. We derive these contrasts from linear combinations of the mean difference of each specific grime type used as a treatment. Each combination provides insight into the relative effects grime subtypes and categories have on the mean change in the quality attribute. In addition, these combinations provide a relative ranking between subtypes within a category and then between categories.

10.2.5 Evaluation of Validity

In any empirical inquiry, we are concerned with the reliability and validity of our methods. In this study, we ensure the reliability and validity of the tools and methods used in the experiment: We selected third-party tools known to be of high quality and relatively bug-free. We have thoroughly tested the tools we developed at the unit, integration, and system level. We include assurance of the validity of the methods as part of the process. Specifically, for each analysis technique used, we validate the assumptions before executing the analysis. The pilot study used to evaluate design size serves to test the data collection method and identify reliability or validity issues in the process. We will correct any issues identified during the pilot study before commencing the complete study.

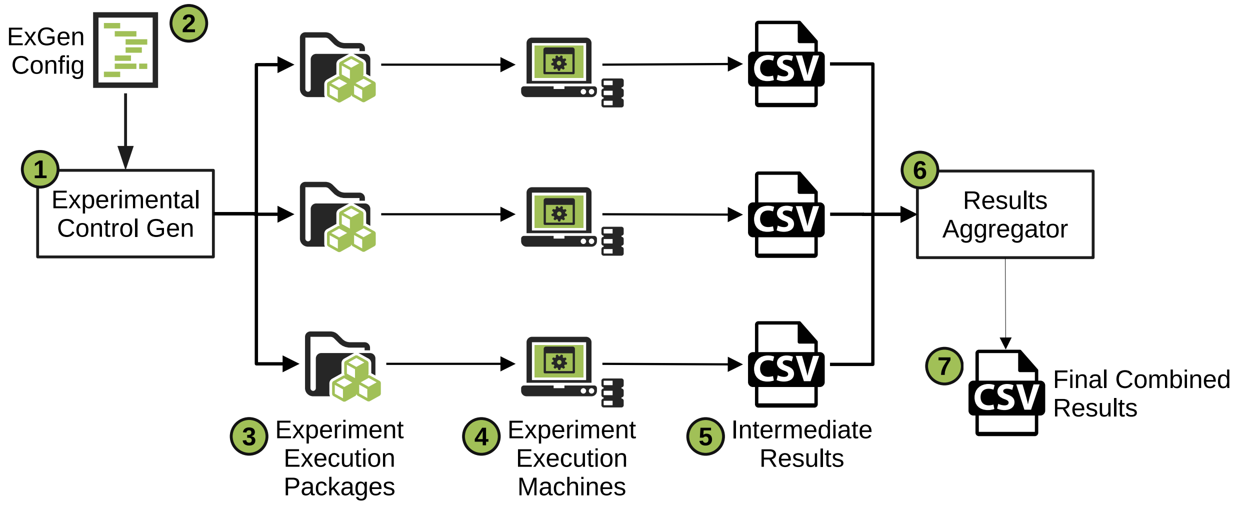


Figure 10.2: Data collection execution process.

10.3 Execution

The data was collected using the process described in Section 10.2.3. In addition to the process defined, we made the following modifications. First, to finish the experiments in a reasonable amount of time, we spread the execution across multiple machines. The exact process for this is depicted in Figure 10.2 and follows the process identified by the numbers encircled in green. 1.) We developed a small tool to generate and separate the experimental configurations for deployment across multiple machines. The *Experimental Control Generator* performs this function by first generating all experimental configurations and then randomizing this list. The *Experimental Control Generator* subdivides this list based on information from 2.) the provided *ExGen Config* which specifies the number of machines, cores per machine, and parts per machine for execution. 3.) The *Experimental Control Generator* combines this data with the experiment executor and several scripts (used to facilitate automated execution and database management) into an *Experiment Execution Package*. 4.) We distribute these packages to several execution machines. During the execution of these experiments, the number of available machines ranged from 6 to 30 desktop

Table 10.2: Size analysis results.

Characteristic	α	**Power**	**Effect Size**	**df**	**Rep Size**	**Size**	**Reps**
Analyzability	0.05	0.95	0.2255272	1875	2496	5425	**3**
Testability	0.05	0.95	0.0863064	1875	2496	29506	**12**
Modifiability	0.05	0.95	0.1695395	1875	2496	8536	**4**
Modularity	0.05	0.95	0.1679058	1875	2496	8661	**4**
TD Principle	0.05	0.95	0.2456884	1875	2496	4816	**2**
TD Interest	0.05	0.95	0.1980468	1875	2496	6602	**3**

computers within a single computer lab. 5.) As experimental packages completed execution, a script collected their results and stored them for later aggregation. 6.) The *Results Aggregator* performs the aggregation combining all results from multiple executions 7.) into a single results CSV file for analysis.

10.4 Analysis Results

This section summarizes the collected data and describes the results of our analysis. This section contains eight subsections: first, we describe the size analysis results, identifying the number of replications needed in the subsequent experiments, and the remaining seven sections detail the results of each experiment conducted.

10.4.1 Size Analysis

We conducted a pilot study to determine the number of replications necessary to achieve the required statistical power for the hypothesis tests in each of the experiments. This study used only two patterns (Singleton and State) for each of the six levels of Injection Severity and all 26 Injection Types. This combination led to the generation of 312 pattern instances.

Table 10.3: Summary of Analyzability data.

Characteristic	Min	Median	Mean	Max	SD
ΔAnalyzability	-1.0382265	-0.0038076	-0.1226307	0.0	0.2772369

These instances were analyzed using the same data collection approach defined for the actual experiments, and the results were collected. These results were analyzed to determine the number of replications needed for each experiment to achieve 95% power when considering the three-way interaction effects. To determine this, we used the G * Power 3.1 [75]. Table 10.2 depicts the results of this analysis.

10.4.2 Analyzability

This subsection describes the results of the Analyzability analysis. We subdivided the analysis into a subsection describing the data and descriptive statistics, and a subsection describing hypothesis testing.

10.4.2.1 Descriptive Statistics This section presents the results of the Analyzability experiment using descriptive statistics and plots. First, we show the summary of the Change in Analyzability (the dependent variable) in Table 10.3. The table shows the basic statistics across the 7,488 observations. This table shows that across all observations, the most significant change in Analyzability (the min value in the table) is negative and had a value of -1.0382265, the smallest change in Analyzability (the max value in the table) is 0.0, and the mean change in Analyzability is -0.1226307. However, given the distribution of the values being highly skewed to the right as depicted by the histogram in Figure 10.3, the median value of -0.0038076 provides a better measure of the centrality of the data. Combining all of this with the standard deviation of 0.2772369, we know the following about this data: i) the vast majority of the observations showed no change to Analyzability; ii) of those

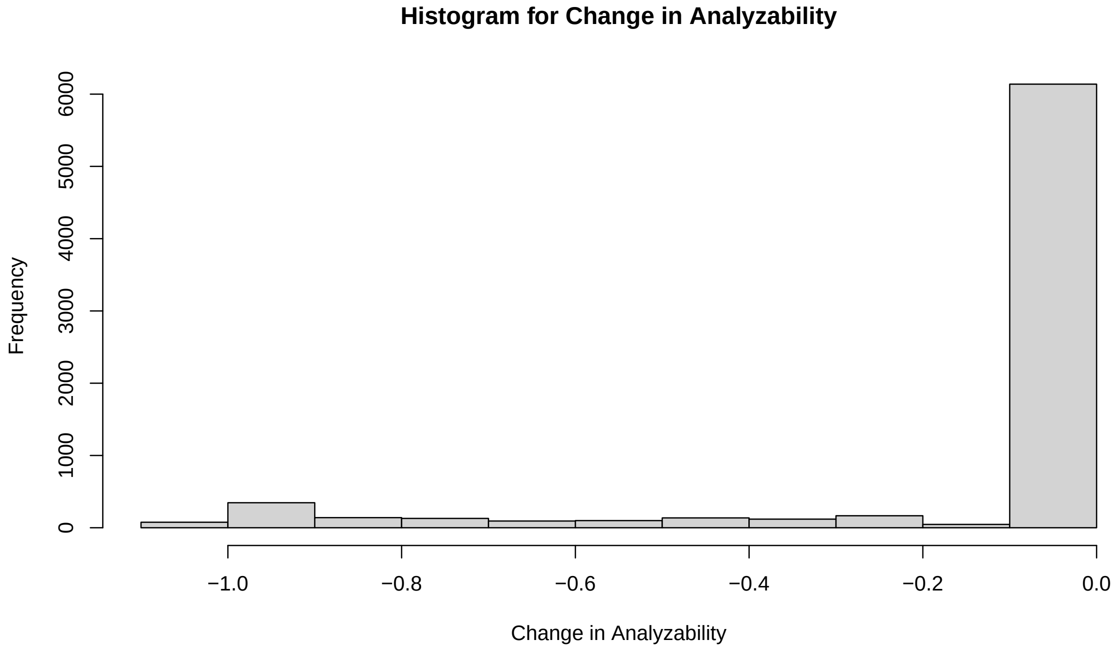


Figure 10.3: Histogram of the change in Analyzability.

observations that showed any change in Analyzability, it is negative and relatively small; and iii) there were some observations which show significant changes in Analyzability. To better understand how this data is distributed, in the context of the independent variables, we constructed two plots: the first is a table plot (see Figure 10.4), and the second is a scatterplot (see Figure 10.5).

Figure 10.4 depicts a table plot of the dependent variable and each of the independent variables. Each column of this plot represents a single variable, while each plot row represents a sample of the data. The first column presents a histogram of the Change in Analyzability (sigAnalyzability) separated into 100 bins containing 75 observations. The remaining columns show the values of Pattern Type (PTFactor), Injection Type (ITFactor), and Injection Severity (ISFactor) for the 75 values for each row of the Change in Analyzability. This data view allows us to see the distribution of the data and any interesting patterns that

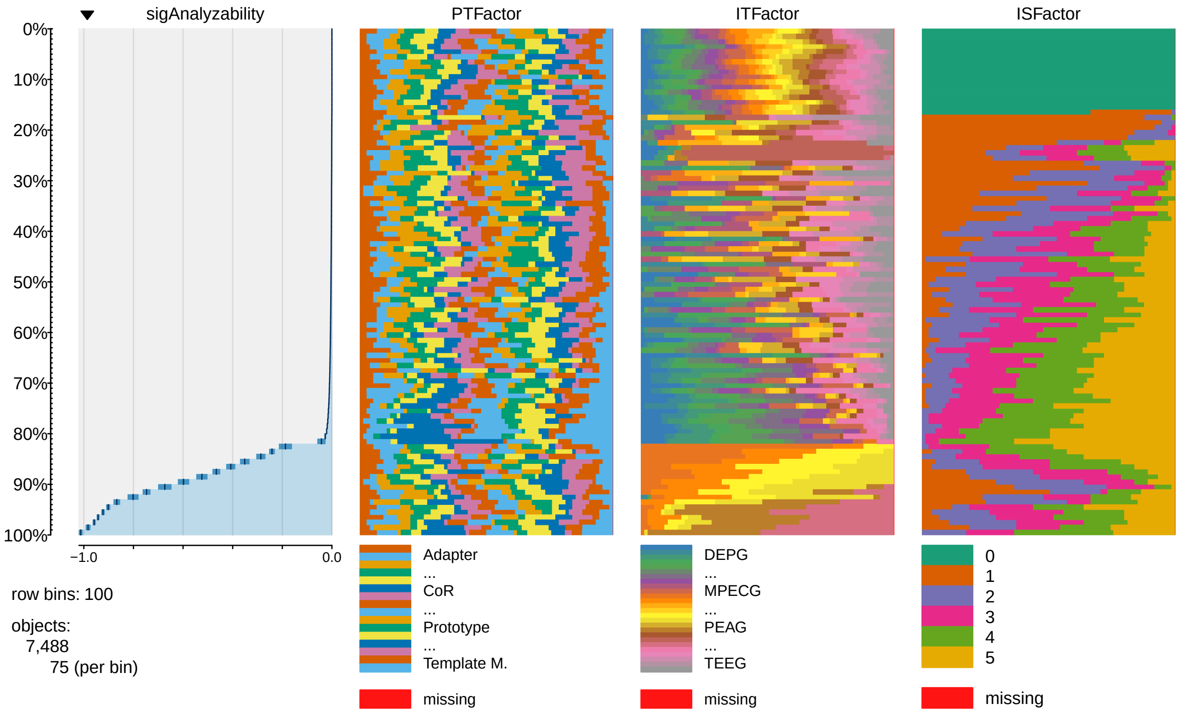


Figure 10.4: Table plot of Analyzability data.

may exist across the columns.

In this plot, we initially see that approximately 75% of the change in Analyzability is very close to zero. At approximately the 75% mark, the data begins visibly deviating from 0.0. This deviation markedly increases at approximately 85%, ending in a value of -1.0382265. Additionally, when considering the Injection Severity, we can see that the only time Injection Severity is 0, the corresponding change in Analyzability is zero. This finding makes sense as this is the only time no grime is injected. Therefore, we should expect to see zero change in Analyzability for this level of Injection Severity. The next exciting piece of information that stands out in this plot is that apparently, only a few Injection Types are responsible for the largest changes in Analyzability. As a final note about this plot, we can see that regardless of the value of the Change in Analyzability, there appears to be little change in the distribution of the Pattern Type data.

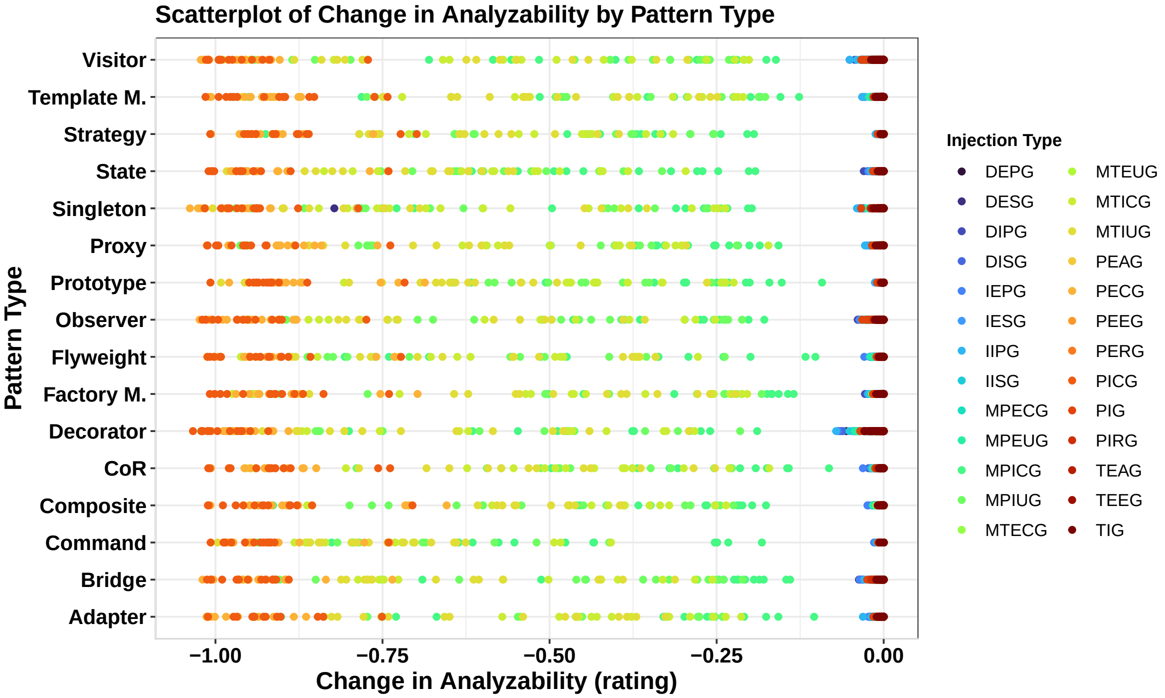


Figure 10.5: Scatterplot of the Change in Analyzability and Pattern Type.

Figure 10.5 shows the scatterplot of the Change in Analyzability by Pattern Type with each point colored according to the Injection Type. This plot shows several key things. First, negative changes occur across all Pattern Types and all Injection Types. However, we can see that the largest magnitude of change is the injection of primarily Organizational Grime. Furthermore, these changes are separated into two bands. First, Package Organizational Grime has the largest magnitude of change ranging from -0.75 to approximately -1.0. Next, the changes due to Modular Organization Grime range from -0.125 to -0.875. The remaining small negative changes are primarily due to Class and Modular Grime.

10.4.2.2 Hypothesis Testing Initially, we begin the analysis by determining if using the parametric ANOVA approach is appropriate. We determined this by validating the fundamental assumptions of ANOVA. As noted above, the two fundamental assumptions we are concerned with are the normality and homogeneity of variances assumptions.

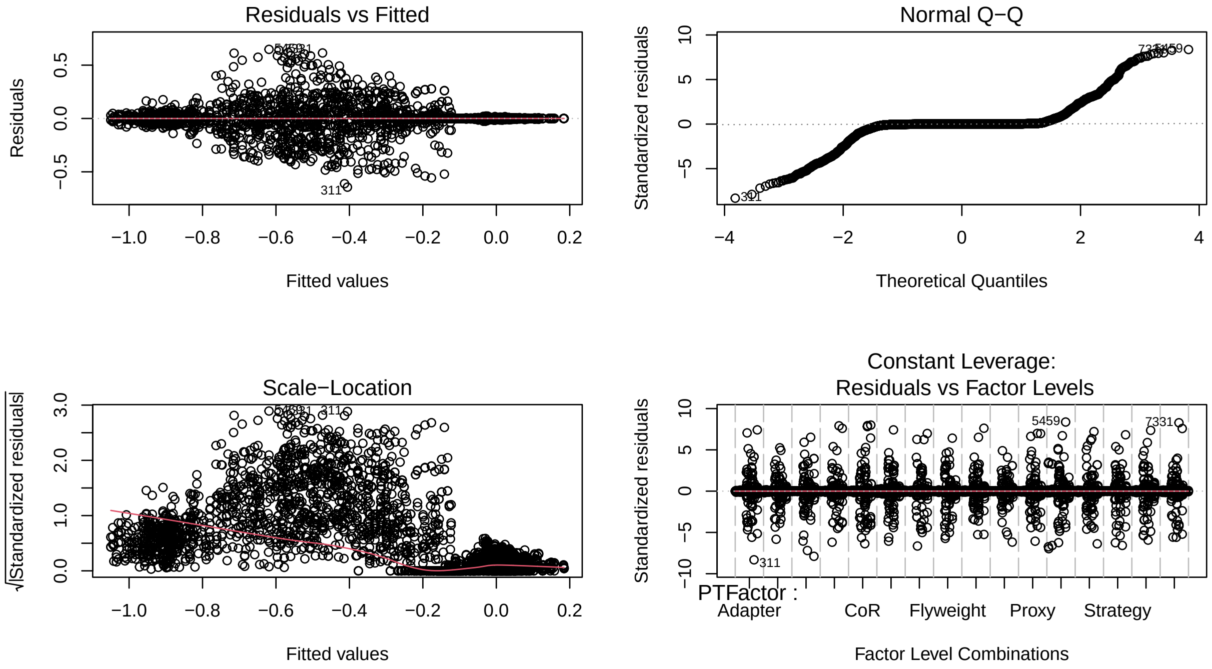


Figure 10.6: Analyzability diagnostic plots.

Normality Assumption To evaluate this assumption, we plotted the ANOVA model, as depicted in Figure 10.6. The pertinent plot here is the “Normal Q-Q” Plot in the upper right quadrant. Here we see deviations from Normal in the tails of the data, which is a strong indicator of a violation of the normality assumption. This evidence is further confirmed using the Anderson-Darling normality test [14] as provided by the `ad.test` function in the `nortest`[1] package for R. This test assumes that we have a set of observations sampled from some continuous distribution, $F(x)$, and that the measurement scale is at least ordinal. If these assumptions are met, then we can compare $F(x)$ to a hypothesized distribution, $F^*(x)$, such as the normal distribution in this case. The null and alternative hypotheses for such a two-sided test are as follows:

[1] `https://cran.r-project.org/package=nortest`

$$\begin{aligned} H_0 &: F(x) = F^*(X) \\ H_A &: F(x) \neq F^*(X) \end{aligned}$$

The results of this test ($A = 1812.5$, $p < 2.2\text{e}{-16}$) provides strong evidence to reject the null hypothesis and further confirming the violation of the normality assumption.

Homogeneity of Variances Assumption We validated this assumption using a similar process as the Normality assumption. We again look to Figure 10.6, focusing on the "Residual vs. Fitted" plot in the upper-left quadrant. This plot indicates that there is a violation of the assumption. To analytically confirm this, we executed Levene's Test for Homogeneity of Variance [162] (as provided by the `car` package in R [89]). The null and alternative hypotheses for this test are as follows:

$$\begin{aligned} H_0 &: \sigma_1^2 = \sigma_2^2 = \ldots = \sigma_n^2 \\ H_A &: \sigma_i^2 \neq \sigma_j^2 \text{ for some } i \neq j \end{aligned}$$

The results ($F(2495, 4992) = 1.6523\text{e}26$, $p < 2.2\text{e}{-16}$) of this test provide strong evidence to reject the null hypothesis that the variances are the same. These results further confirming this assumption has been violated.

Permutation F-Test Analysis The results of the assumption validation steps lead to the conclusion that either we transform the data or we use a permutation F-test approach (according to our analytical approach defined above in Section 10.2.4). After several attempts to adjust for the violations, we opted to conduct the permutation F-test approach. The permutation F-Test tests the following null and alternative hypotheses:

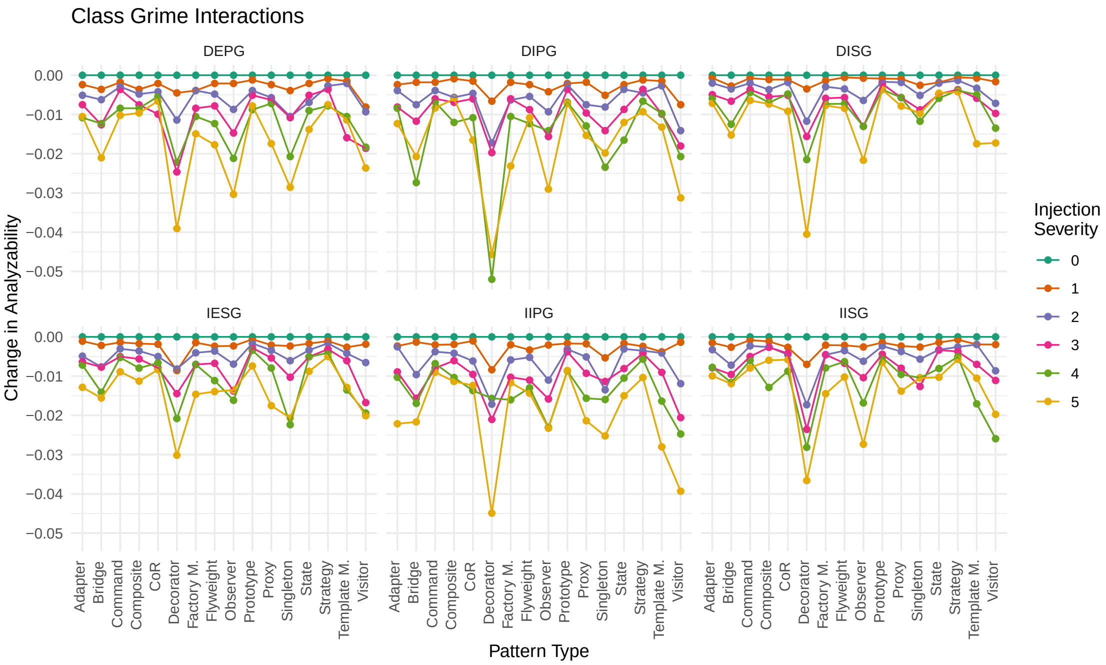


Figure 10.7: Analyzability Class Grime interactions part 1.

$$H_{1,0} \quad : \quad \mu_{111} = \mu_{112} = \ldots = \mu_{ijk}$$

$$H_{1,1} \quad : \quad \mu_{ijk} \neq \mu_{i'j'k'}$$

Where μ_{ijk} represents the mean change in Analyzability for the ith Pattern Type, jth Injection Type, and kth Injection Severity. If the evidence is strong enough to reject the null hypothesis, $H_{1,0}$, then we will evaluate whether we should consider the main or the interaction effects. To test this, we used the lmp function of the lmPerm[2] package for R. The overall results of this test ($F(2495, 4992) = 23.68$, $p < 2.2\text{e}{-16}$) indicate strong evidence to reject the null hypothesis that there is no difference in the mean change in Analyzability.

[2] https://cran.r-project.org/package=lmPerm

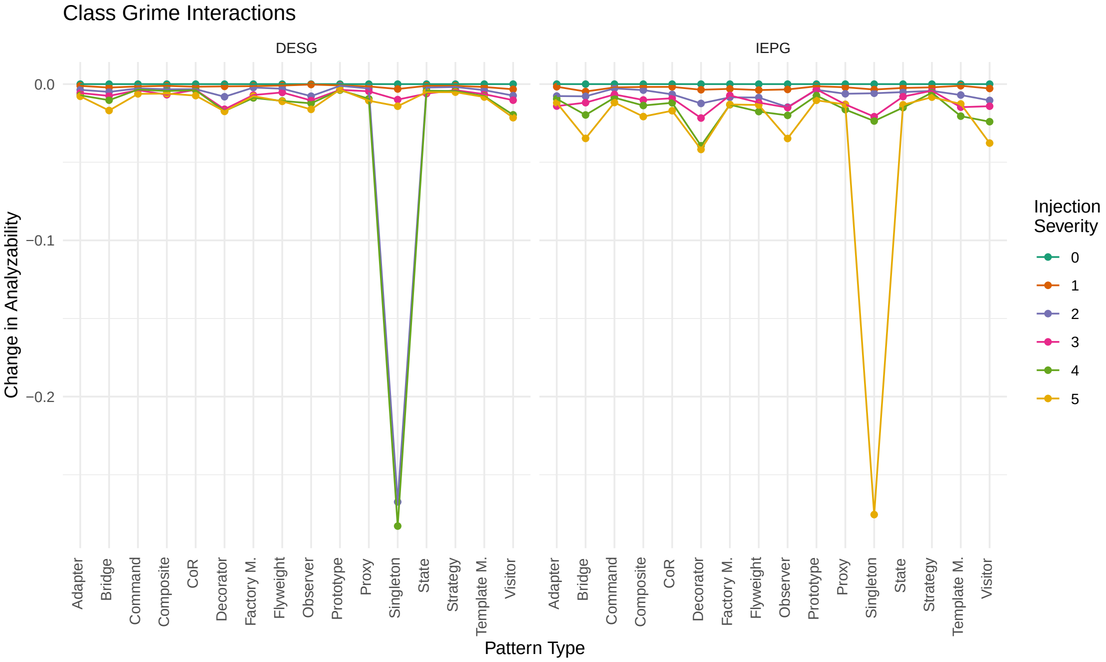


Figure 10.8: Analyzability Class Grime interactions part 2.

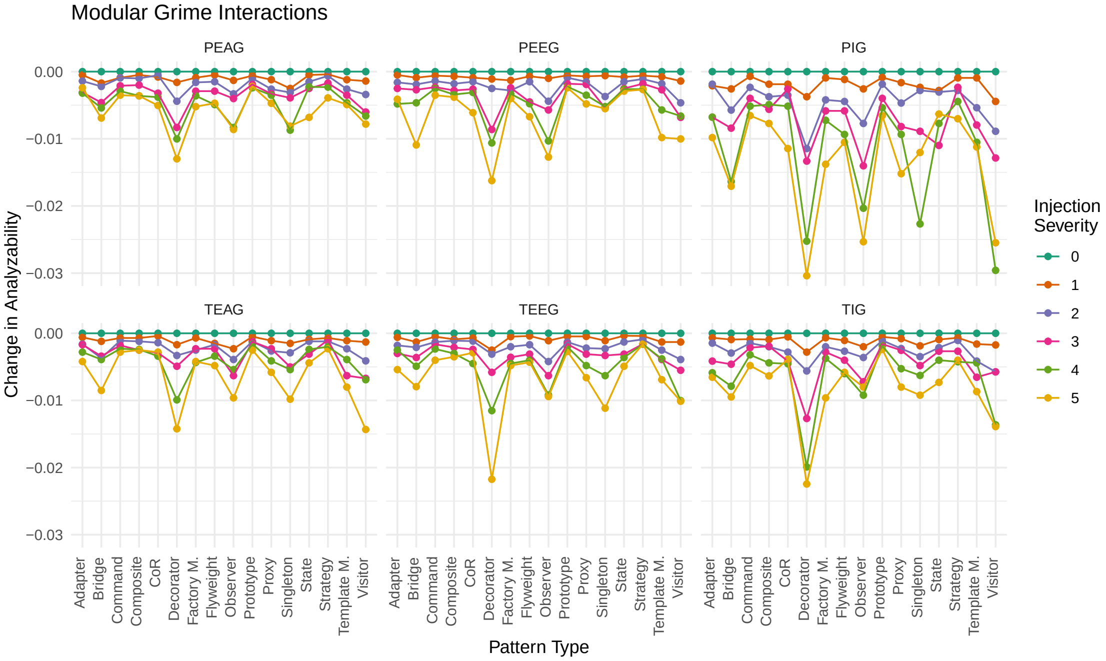


Figure 10.9: Analyzability Modular Grime interactions.

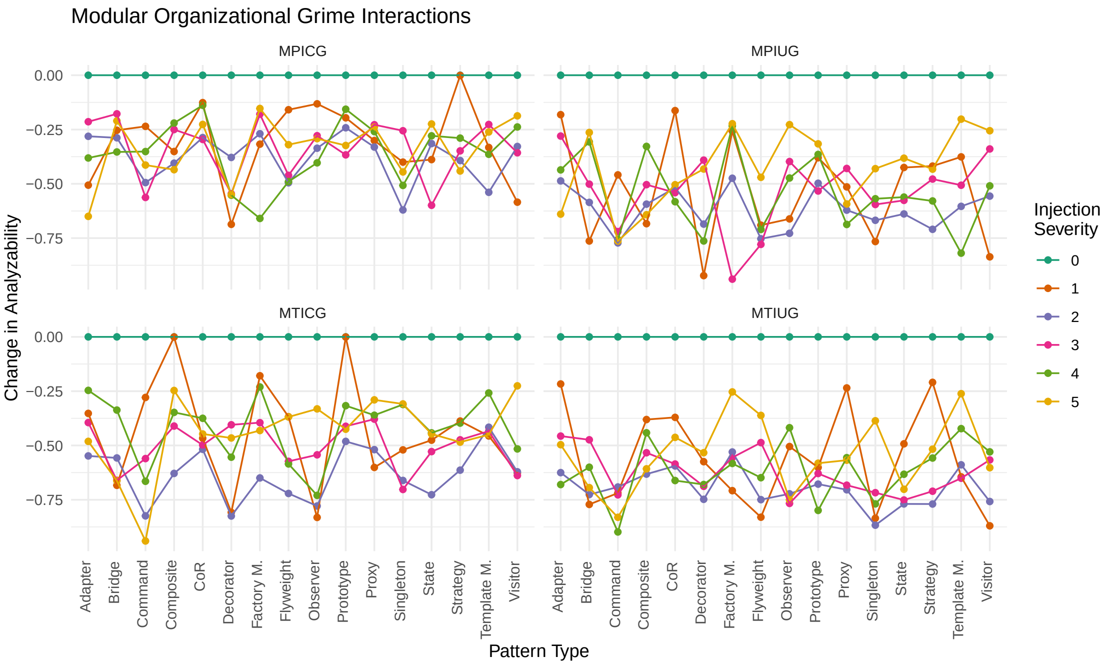


Figure 10.10: Analyzability Modular Organizational Grime interactions part 1.

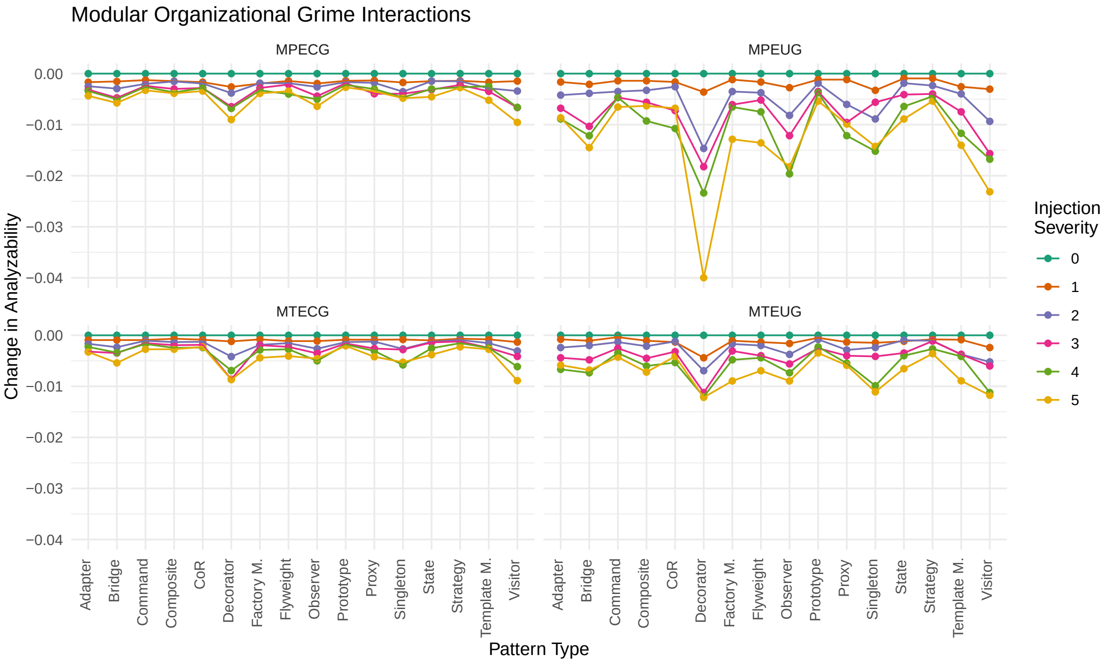


Figure 10.11: Analyzability Modular Organizational Grime interactions part 2.

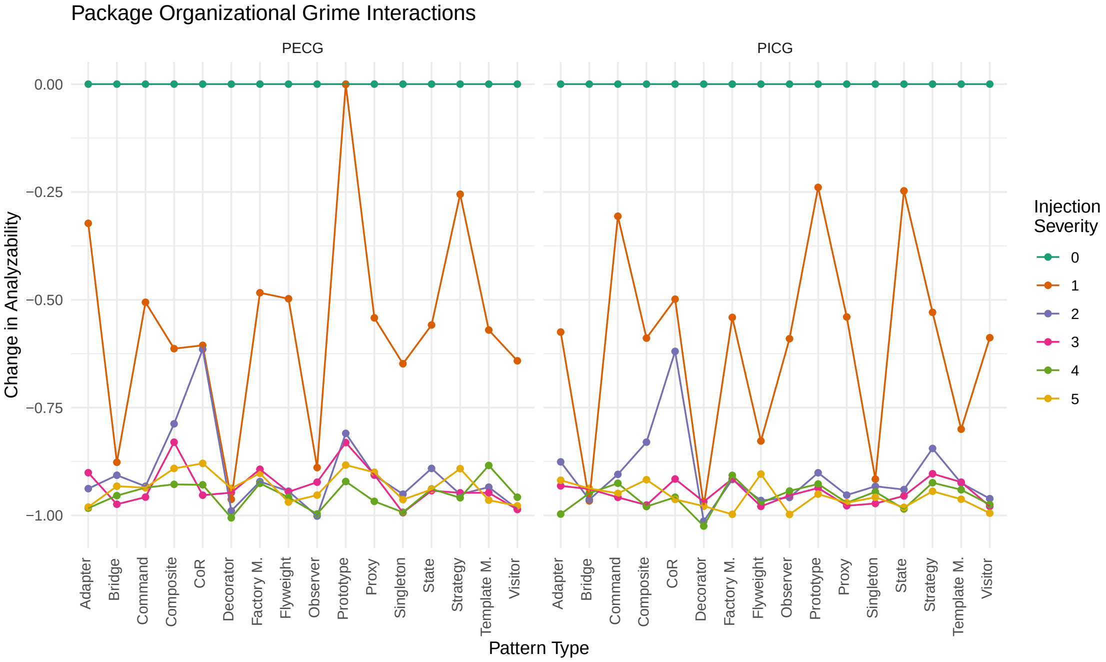


Figure 10.12: Analyzability Package Organizational Grime interactions.

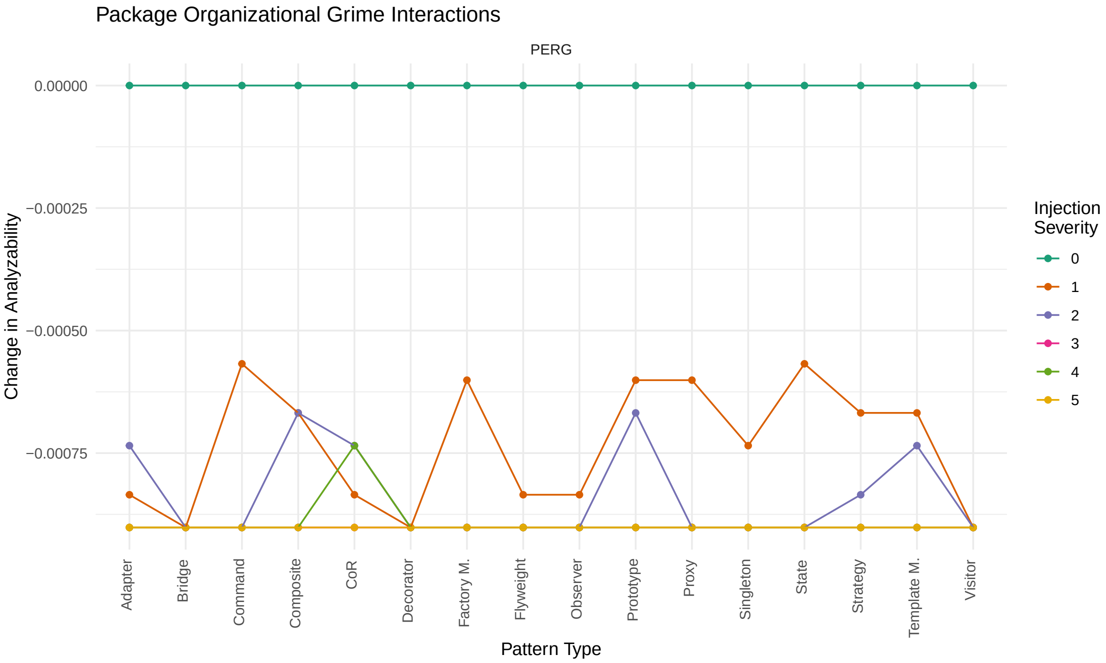


Figure 10.13: Analyzability interactions for the PERG subtype.

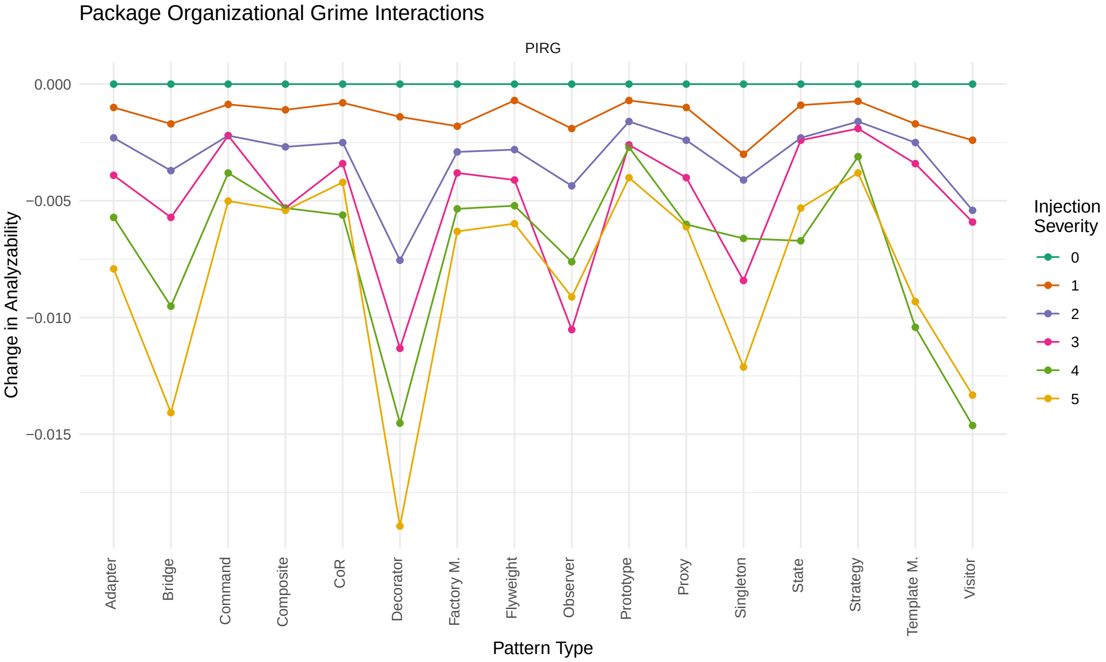


Figure 10.14: Analyzability interactions for the PIRG subtype.

Interaction Effects With the knowledge that a difference in the mean Change in Analyzability exists between two or more treatment combinations, we continue by considering any significant interactions. We begin by testing the following hypotheses:

$$H_{2,0} \ : \ (\tau\beta\gamma)_{111} = (\tau\beta\gamma)_{112} = \ldots = (\tau\beta\gamma)_{ijk}$$

$$H_{2,1} \ : \ \text{at least one } (\tau\beta\gamma)_{ijk} \neq (\tau\beta\gamma)_{i'j'k'}$$

In this case, there is strong evidence ($p < 2.2\text{e}{-16}$) to reject $H_{2,0}$ that there is no difference in the mean change in Analyzability for each level of the three-way interaction effect. With this in mind, we will consider a graphical analysis of these interactions. To plot these interactions, we subdivided into grime categories: Class Grime, Modular Grime, and Organizational Grime (which we further split into Package Organizational Grime and Modular Organizational Grime). Each grime category plot contains a matrix of subplots (one per grime type in the category). The y-axis is the change in Analyzability, the x-axis is the design pattern type, and the points plotted are the values for each injection severity. We begin with the plots for Class Grime.

Figures 10.7 and 10.8 depict the interaction effects associated with the class grime subtypes. Each subplot in this figure shows that as injection severity increases, there is a corresponding decrease in the change in Analyzability across each pattern type. We will note that for the Bridge, Decorator, Observer, Singleton, and Visitor patterns, there is a more pronounced dip in the change. Additionally, for DESG and IEPG, there are significant dips in the Change in Analyzability for the Singleton pattern. Specifically for DESG when Injection Severity is 3 or 4 and for IEPG when Injection Severity is 5. Given these results, we now shift to the interaction plots for the Modular Grime category.

Figure 10.9 depicts the interaction effects associated with the modular grime subtypes. Like the Class Grime subtypes, we can see an apparent corresponding decrease in the Change in Analyzability across all pattern types as injection severity increases. A similar relationship appears to be true for each subtype of modular grime, but there is a stark contrast between PIG and the other forms of Modular Grime. Additionally, we can see similar dips in the Change in Analyzability for Bridge, Decorator, Observer, Singleton, and Visitor patterns as we did in Class Grime.

Figures 10.10 and 10.11 depict the interaction effects associated with modular organizational grime subtypes. In Figure 10.11, for MPECG, MTECG, and MTEUG, we see the familiar trend of a decrease in the Change in Analyzability as Injection Severity increases across pattern types. Furthermore, we also can see that there are pronounced dips at the Bridge, Decorator, Observer, Singleton, and Visitor design patterns. One might question how a Singleton pattern is affected by Organizational Grime given that Singleton instances are typically housed in a single file. Because Organizational Grime concerns the relationships between packages, a Singleton instance can cause cycles between packages (MPECG and MTECG), induce instability between packages (MTEUG) via couplings to classes in other

Table 10.4: Summary of Testability data.

Characteristic	**Min**	**Median**	**Mean**	**Max**	**SD**
ΔTestability	-1.459066	-0.005952	-0.248710	0.0	0.5072027

packages and therefore reducing overall quality. However, in Figure 10.10 there is significant variability in the change between levels of Injection Severity depending on the pattern type for MPICG, MPIUG, MTICG, and MTIUG. It appears that for MPICG that regardless of the pattern type and injection severity level, the effect is nearly the same.

Finally, Figures 10.12, 10.13, and 10.14 depict the interaction effects associated with package organizational grime. For PECG and PICG subtypes, depicted in Figure 10.12, we note that there is only small variability across design pattern types for Injection Severity levels of 3 or more. The primary variability in these grime types occurs for Injection Severity levels 1 and 2. When considering PERG, as depicted in Figure 10.13, we note that for Injection Severity levels 3 and above, there is no discernible variability across patterns, but for levels 1 and 2 pattern type seems to make some difference, but the actual difference is extremely small (approximately -0.00055 to -0.00080 on average). Finally, when considering PIRG, as depicted in Figure 10.14, the familiar interaction pattern returns. This figure shows a corresponding decrease in Change in Analyzability across patterns as Injection Severity increases. Again, we also note that there are discernible dips at the Bridge, Decorator, Observer, Singleton, and Visitor design pattern types.

10.4.3 Testability

This subsection describes the results of the Testability analysis. We subdivided the analysis into a subsection describing the data and descriptive statistics, and a subsection describing hypothesis testing.

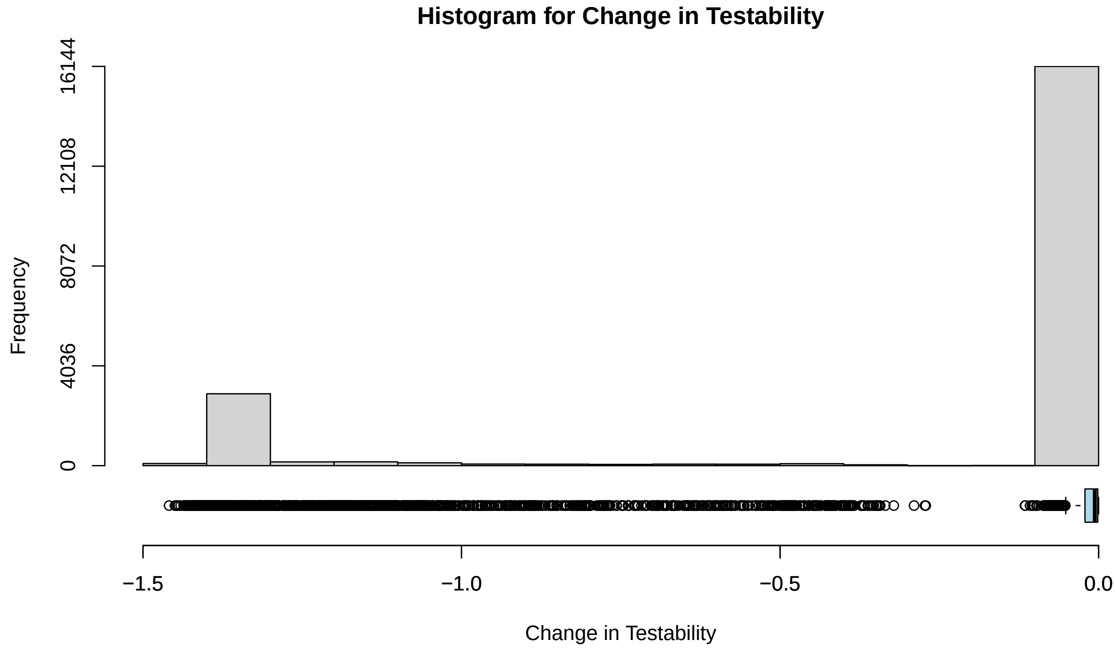


Figure 10.15: Histogram of the change in Testability.

10.4.3.1 Descriptive Statistics This section presents the results of the Testability experiment using descriptive statistics and plots. First, we show the summary of the Change in Testability (the dependent variable) in Table 10.4. The table shows the basic statistics across the 19,968 observations. This table shows that across all observations, the most significant change in Testability (the min value in the table) is negative and has a value of -1.459066, the smallest change in Testability (the max value in the table) is 0.0, and the mean change in Testability is -0.248710. However, given the distribution of the values being bimodal and highly skewed to the right as depicted by the histogram in Figure 10.15, the median value of -0.005952 provides a better measure of the centrality of the data. Combining all of this with the standard deviation of 0.5072027, we know the following about this data: i) the vast majority of the observations showed no change to Testability; ii) of those observations that showed any change in Testability, it is negative and relatively small; and iii) there were

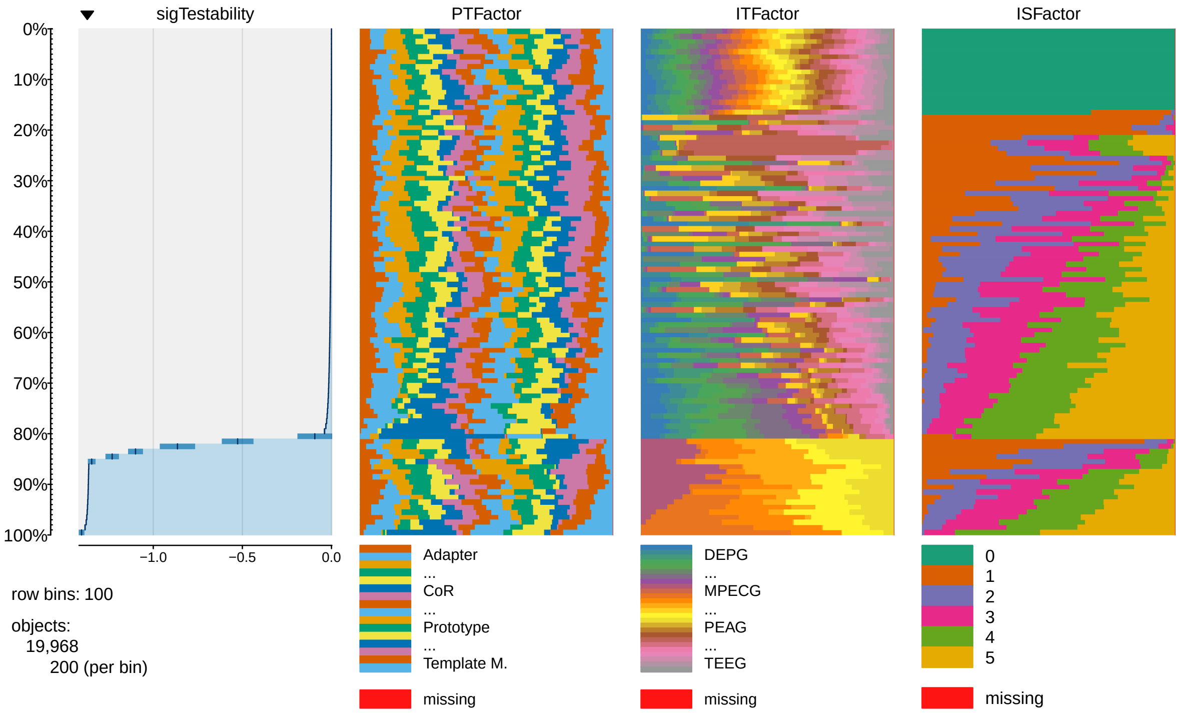


Figure 10.16: Table plot of Testability data.

some observations which show significant changes in Testability. To better understand how this data is distributed, in the context of the independent variables, we constructed two plots: the first is a table plot (see Figure 10.16), and the second is a scatterplot (see Figure 10.17).

Figure 10.16 depicts a table plot of the dependent and each of the independent variables. Each column of this plot represents a single variable, while each plot row represents a sample of the data. The first column presents a histogram of the Change in Testability (sigTestability) separated into 100 bins containing 200 observations. The remaining columns show the values of Pattern Type (PTFactor), Injection Type (ITFactor), and Injection Severity (ISFactor) for the 75 values for each row of the Change in Testability. This data view allows us to see the distribution of the data and any interesting patterns that may exist across the columns.

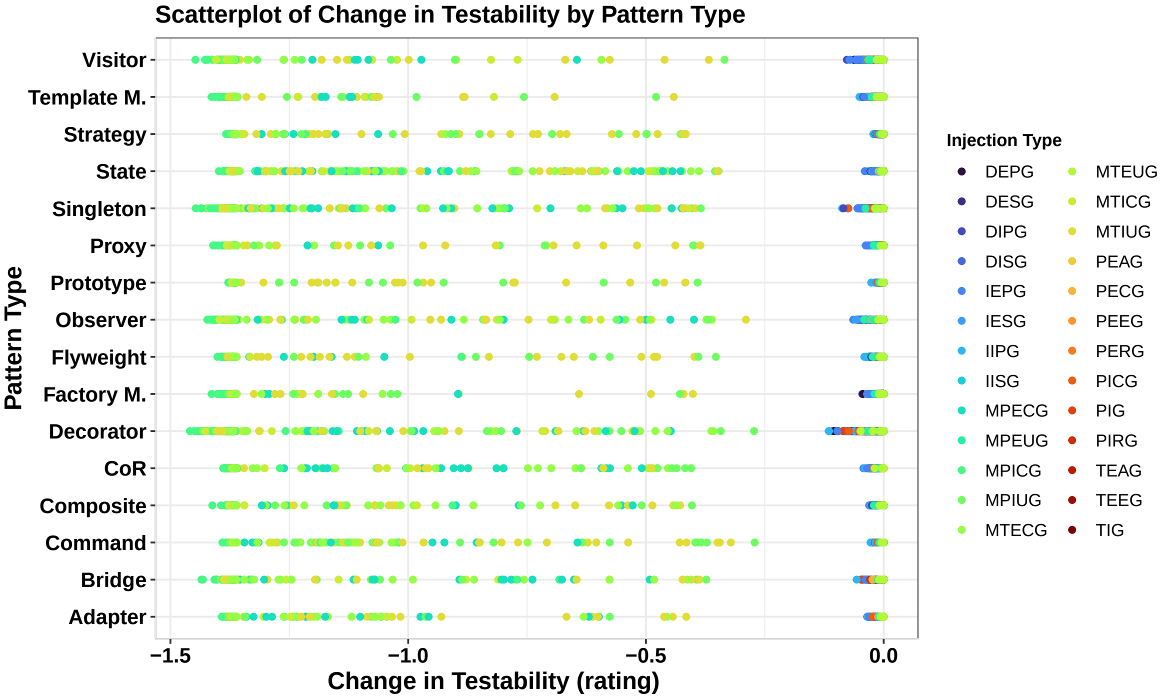


Figure 10.17: Scatterplot of the Change in Testability and Pattern Type.

In this plot, we initially see that approximately 75% of the change in Testability is very close to zero. At approximately the 75% mark, the data begins visibly deviating from 0.0. This deviation markedly increases at approximately 85%, ending in a value of -1.459066. Additionally, when considering the Injection Severity, we can see that the only time Injection Severity is 0, the corresponding change in Testability is zero. This finding makes sense as this is the only time no grime is injected. Therefore, we should expect to see zero change in Testability for this level of Injection Severity. The next exciting piece of information that stands out in this plot is that apparently, only a few Injection Types are responsible for the largest changes in Testability. Finally, we see that regardless of the value of the Change in Testability, there appears to be little change in the distribution of the Pattern Type data.

Figure 10.17 shows the scatterplot of the Change in Testability by Pattern Type colored according to the Injection Type. This plot shows several key things. First, negative changes occur across all Pattern Types and all Injection Types. However, we can see that the largest magnitude of change is the injection of primarily Modular Organizational Grime. The changes due to Modular Organization Grime ranges from approximately -0.125 to approximately -1.5. The remaining small negative changes are primarily due to Class, Modular, and Package Organizational Grime. With this basic understanding of the data, we next discuss data set reduction and hypothesis testing.

10.4.3.2 Data Set Reduction The initial data collection for this experiment collected 12 replicates consisting of a total of 29952 observations. However, due to the long processing time, this was reduced to 8 replicates of 19968 total observations.

10.4.3.3 Hypothesis Testing Initially, we begin the analysis by determining if using the parametric ANOVA approach is appropriate by validating its fundamental assumptions. As noted in Section 10.2.4, the two fundamental assumptions we are concerned with are the normality and homogeneity of variances assumptions.

Normality Assumption To evaluate this assumption, we plotted the ANOVA model, as depicted in Figure 10.18. The pertinent plot here is the "Normal Q-Q" Plot in the upper right quadrant. Here we see deviations from normal in the tails of the data, which is a strong indicator of a violation of the normality assumption. This evidence is further confirmed using the Anderson-Darling normality test. The results of this test ($A = 5069.2$, $p < 2.2\text{e}{-16}$) provides strong evidence to reject the null hypothesis and further confirming the violation of the normality assumption.

Homogeneity of Variances Assumption This assumption is evaluated using a similar process as the Normality assumption. We again look to Figure 10.18, focusing on the "Residual vs. Fitted" plot in the upper-left quadrant. This plot indicates that there is

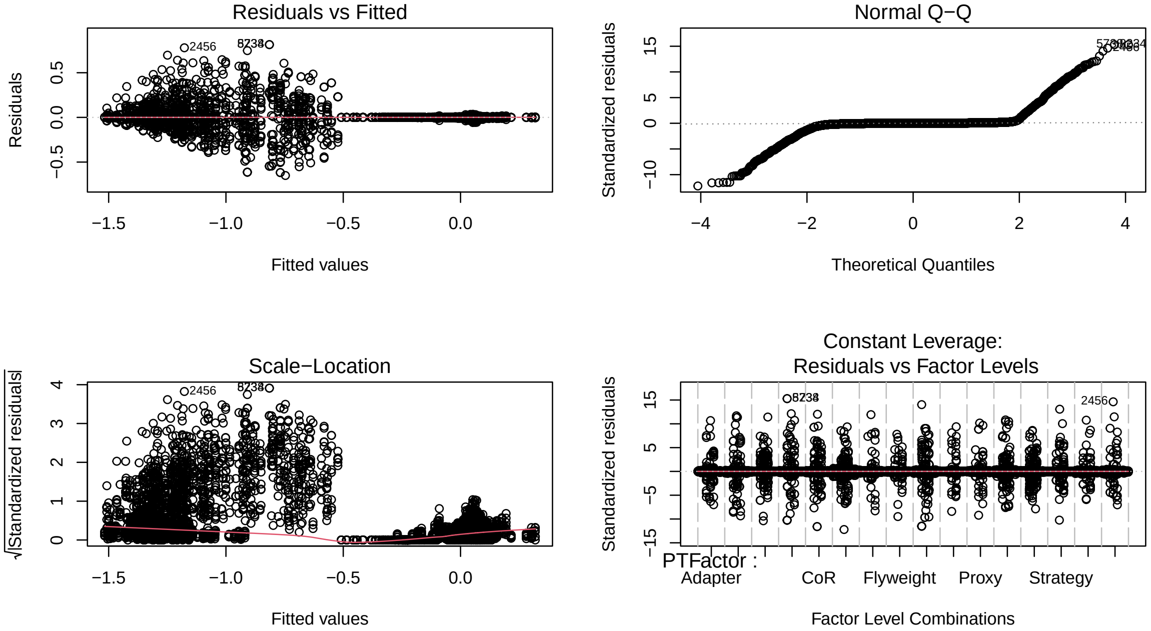


Figure 10.18: Testability diagnostic plots.

a violation of the assumption. To analytically confirm this, we executed Levene's Test for Homogeneity of Variance. The results ($F(2495, 17472) = 8.7427$, $p < 2.2\text{e}{-16}$) of this test provides strong evidence to reject the null hypothesis that the variances are the same. These results further confirming this assumption has been violated.

Permutation F-Test Analysis The results of the assumption validation steps lead to the conclusion that either we need to transform the data or use a permutation F-test approach (according to our analytical approach defined above in Section 10.2.4)). After several attempts to adjust for the violations, we opted to conduct the permutation F-test approach. The overall results of this test ($F(2495, 17472) = 627.8$, $p < 2.2\text{e}{-16}$) indicates strong evidence to reject the null hypothesis that there is no difference in the mean change in Testability.

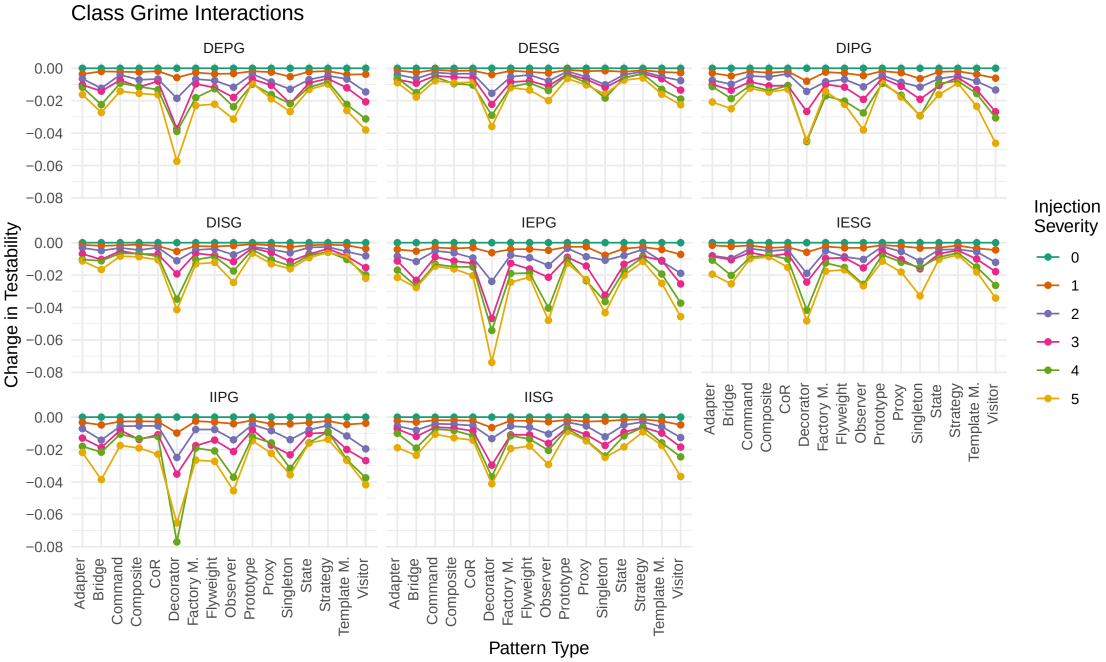


Figure 10.19: Testability interaction plots for class grime injection.

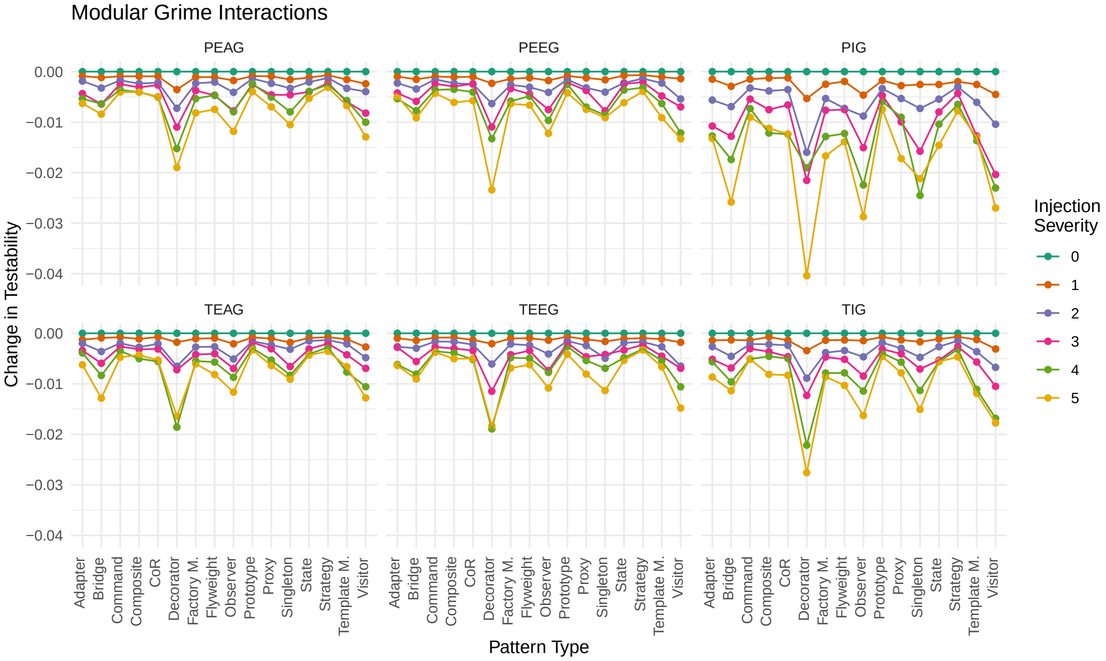


Figure 10.20: Testability interaction plots for modular grime injection.

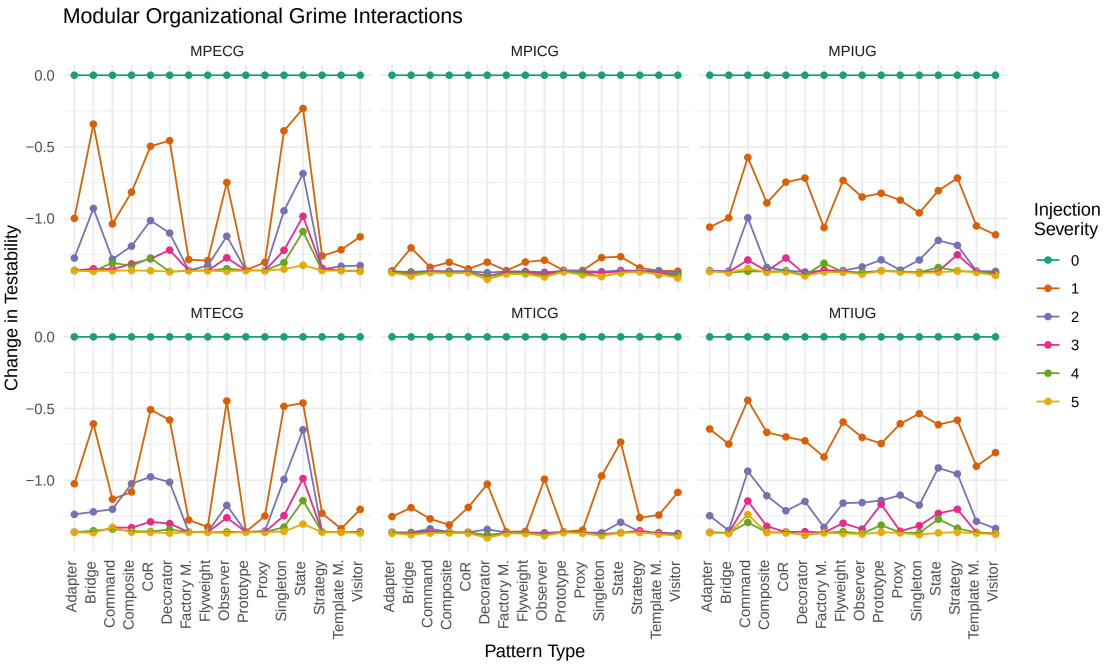


Figure 10.21: Testability interaction plots for modular organizational grime injection.

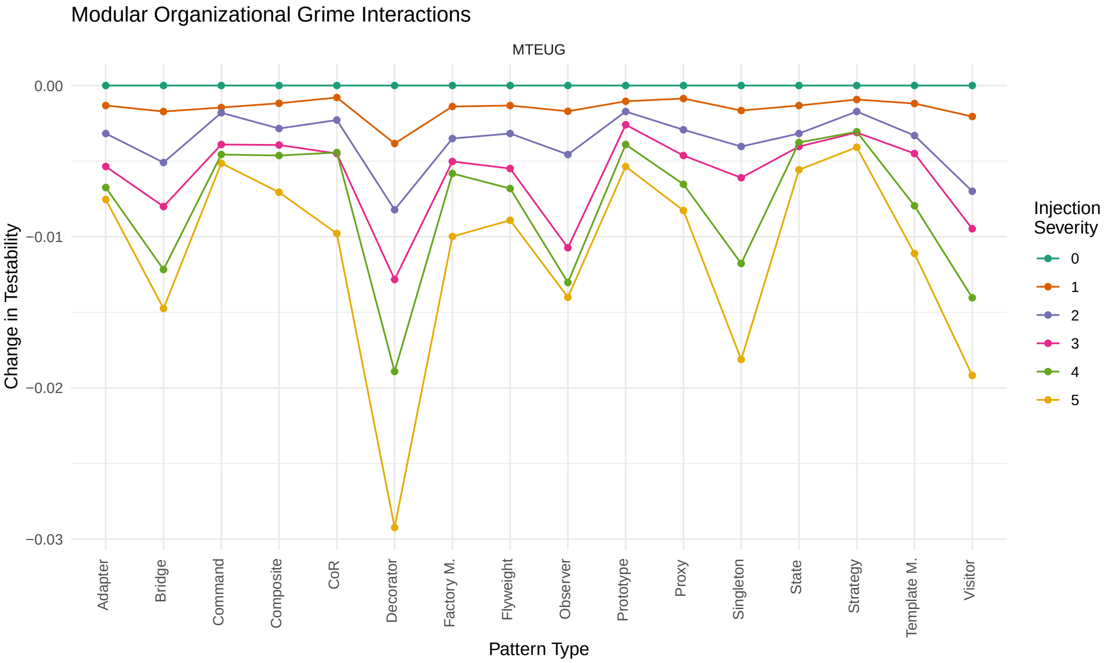


Figure 10.22: Testability interaction plots for MTEUG subtype.

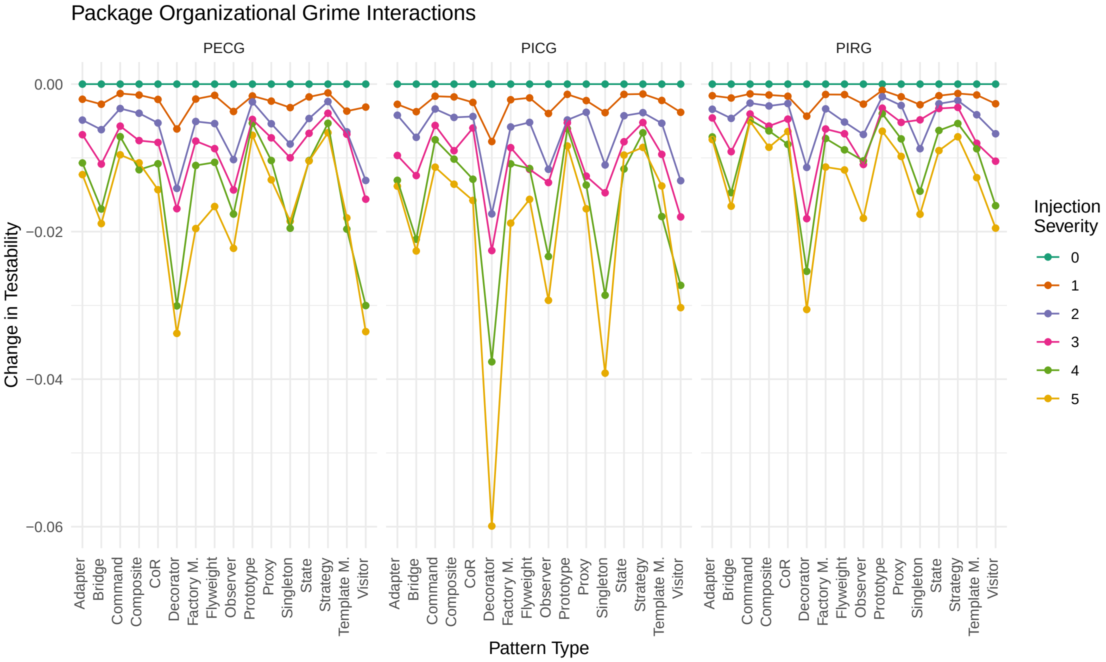


Figure 10.23: Testability interaction plots for package organizational grime injection.

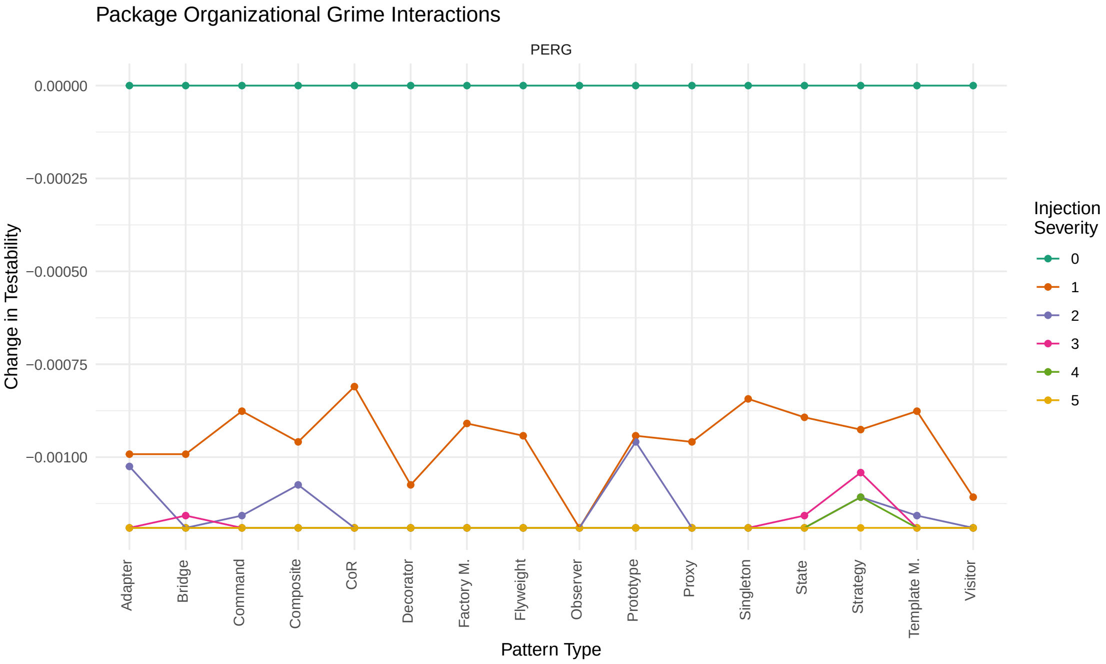


Figure 10.24: Testability interaction plots for PERG subtype.

Interaction Effects With the knowledge that a difference in the mean change in Testability exists between two or more treatment combinations, we continue by considering significant interactions. In this case, there is strong evidence ($p < 2.2\text{e}{-16}$) to reject $H_{2,0}$ that there is no difference in the mean change in Testability for each level of the three-way interaction effect. With this in mind, we consider a graphical analysis of these interactions. To plot these interactions, we subdivided them into grime categories: Class Grime, Modular Grime, and Organizational Grime. Each grime category plot contains a matrix of subplots (one per grime type in the category). The subplot matrix contains interaction plots. The y-axis is the change in Testability, the x-axis is the design pattern type, and the points plotted are the values for each injection severity. We begin with the plots for Class Grime.

Figure 10.19 depicts the interaction effects associated with the class grime subtypes. Each subplot in this figure shows that as injection severity increases, there is a corresponding decrease in the change in Testability across each pattern type. We will note that there is a more pronounced dip in the change for the Bridge, Decorator, Observer, Singleton, and Visitor patterns. Given these results, we now shift to the interaction plots for the Modular Grime category.

Figure 10.20 depicts the interaction effects associated with the modular grime subtypes. Similar to the Class Grime subtypes, we see an apparent corresponding decrease in the change in Testability across all pattern types as injection severity increases. This relationship appears to be true for each subtype of modular grime, but there is a stark contrast between PIG and the other forms of Modular Grime. Additionally, we can see similar dips in the change in Testability for Bridge, Decorator, Observer, Singleton, and Visitor patterns as we did in Class Grime.

Figures 10.21 and 10.22 depict the interaction effects associated with modular organizational grime subtypes. As depicted in Figure 10.21, we can see an apparent corresponding decrease in the change in Testability as Inject Severity increases across all pattern types.

Table 10.5: Summary of Modifiability data.

Characteristic	Min	Median	Mean	Max	SD
ΔModifiability	-1.36	0.0	-0.01526	0.49121	0.1230359

The difference here is that there is significant variability in the change between levels of Injection Severity depending on the pattern type for MPECG, MPIUG, MTECG, MTICG, and MTIUG. It appears that for MPICG, as depicted in Figure 10.22 the familiar pattern of decreasing Change in Testability as Injection Severity increases across Pattern Types. We can also see dips for the Bridge, Decorator, Observer, Singleton, and Visitor patterns.

Finally, Figures 10.23 and 10.24 depict the interaction effects associated with package organizational grime. As shown in Figure 10.23, there is a corresponding decrease in the Change in Testability as the Injection Severity level increases across each Pattern Type. Again, we also note that there are dips in the effect level for the Bridge, Decorator, Observer, Singleton, and Visitor patterns. PERG, on the other hand, as shown in Figure 10.24 shows minimal variability in the Change in Testability for Injection Severity levels of 3 or higher, but does show some for levels 1 and 2.

10.4.4 Modifiability

This subsection describes the results of the Modifiability analysis. We subdivided the analysis into a subsection describing the data and descriptive statistics, and a subsection describing hypothesis testing.

10.4.4.1 Descriptive Statistics This section presents the results of the Modifiability experiment using descriptive statistics and plots. First, we show the summary of the Change in Modifiability (the dependent variable) in Table 10.5. The table shows the basic statistics across the 9,984 observations. This table suggests that across all observations, the change

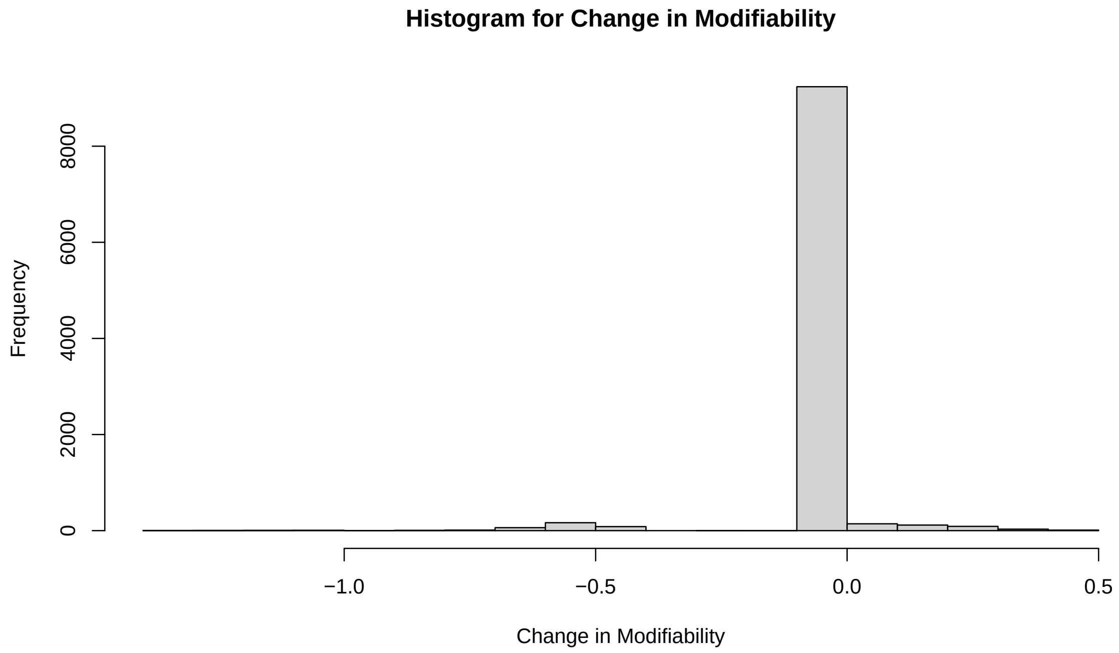


Figure 10.25: Histogram of the change in Modifiability.

in Modifiability ranges between -1.36 and 0.49121, and the mean change in Modifiability is -0.01526. However, given the distribution of the values being skewed to the right as depicted by the histogram in Figure 10.25, the median value of -0.005952 provides a better measure of the centrality of the data. Combining all of this with the standard deviation of 0.1230359, we know the following about this data: i) the vast majority of the observations showed no change to Modifiability; ii) of those observations that showed any change in Modifiability, it can be either negative or positive and that the magnitude is greater in the negative direction; and iii) There were some observations which show significant changes in Modifiability. To better understand how this data is distributed, in the context of the independent variables, we constructed two plots: the first is a table plot (see Figure 10.26), and the second is a scatterplot (see Figure 10.27).

Figure 10.26 depicts a table plot of the dependent and each of the independent variables.

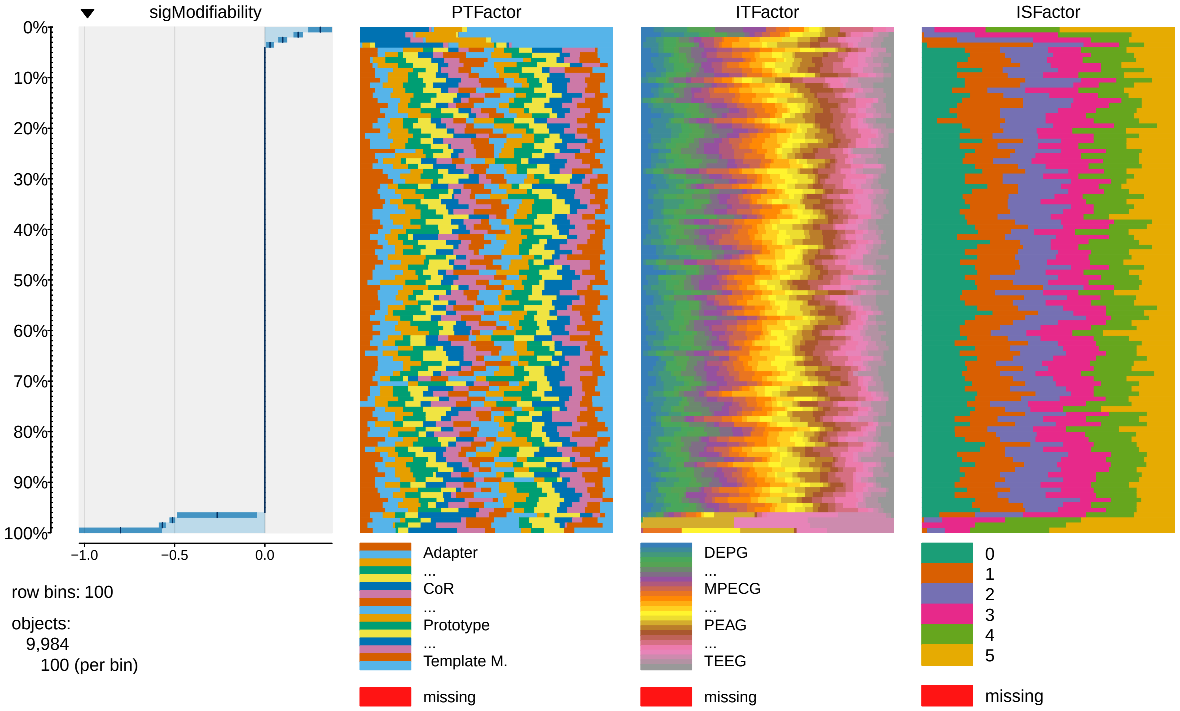


Figure 10.26: Table plot of Modifiability data.

Each column of this plot represents a single variable, while each plot row represents a sample of the data. The first column presents a histogram of the Change in Modifiability (sigModifiability) separated into 100 bins each containing 100 observations. The remaining columns show the values of Pattern Type (PTFactor), Injection Type (ITFactor), and Injection Severity (ISFactor) for the 100 values for each row of the Change in Modifiability. This data view allows us to see the distribution of the data and any interesting patterns that may exist across the columns.

In this plot, we initially see that approximately 5% of the change in Modifiability is positive, 5% is negative, and the remaining is zero. Between the 0% and approximately 5% marks the change in Modifiability is positive and appears to related to only a subset of the patterns, but there does not appear to be any relation to any of the other independent variables. Additionally, the last approximately 5% of the data indicates a negative change

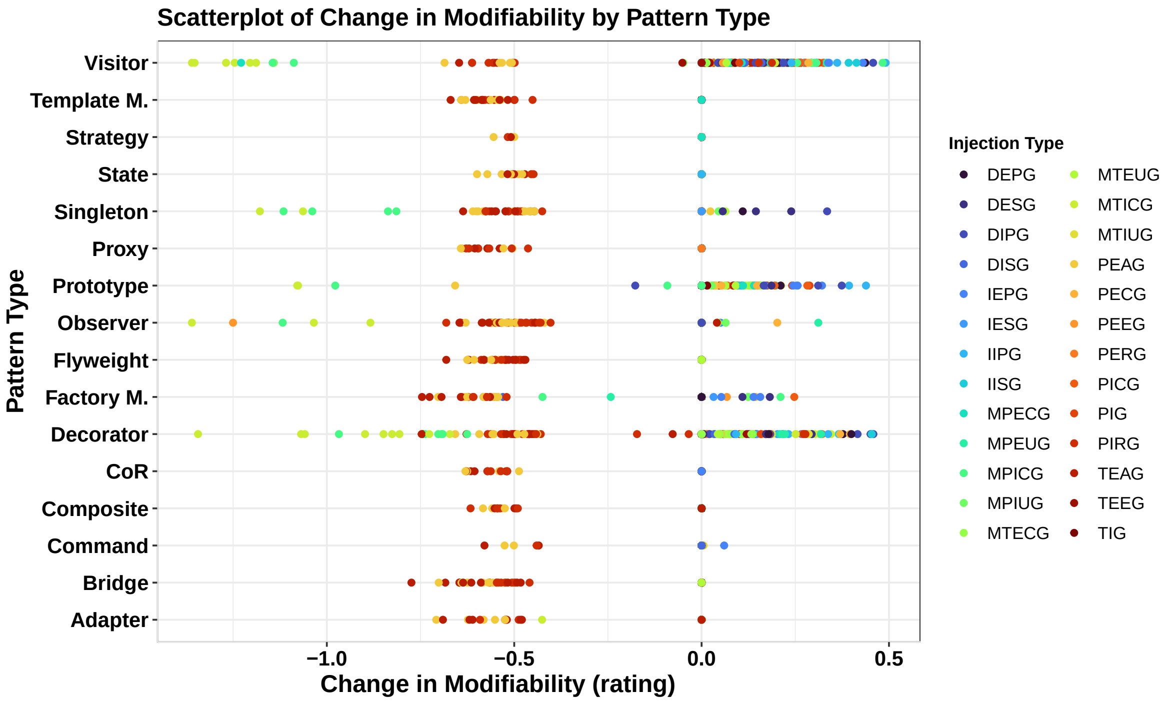


Figure 10.27: Scatterplot of the Change in Modifiability and Pattern Type.

in Modifiability and appears to be related to a subset of Injection Types but not to either the Pattern Type nor the Injection Severity. The remaining 90% of the data has a value of zero. This data is the only data where the Injection Severity level of 0 occurs, but at the same time all other Injection Severity levels, Pattern Types, and Injection Types also affect a change of zero.

Figure 10.27 shows the scatterplot of the Change in Modifiability by Pattern Type, with each point colored according to the Injection Type. This plot shows several key things. First, negative changes occur across all Pattern Types and all Injection Types. Furthermore, we can see that the negative changes are due to primarily Organizational Grime injection. The largest magnitude of negative change focuses on the Visitor, Singleton, Prototype, Observer, and Decorator patterns and is due to Modular Organization Grime. However, the negative changes are due to Package Organizational Grime (ranging from approximately -0.4375 to

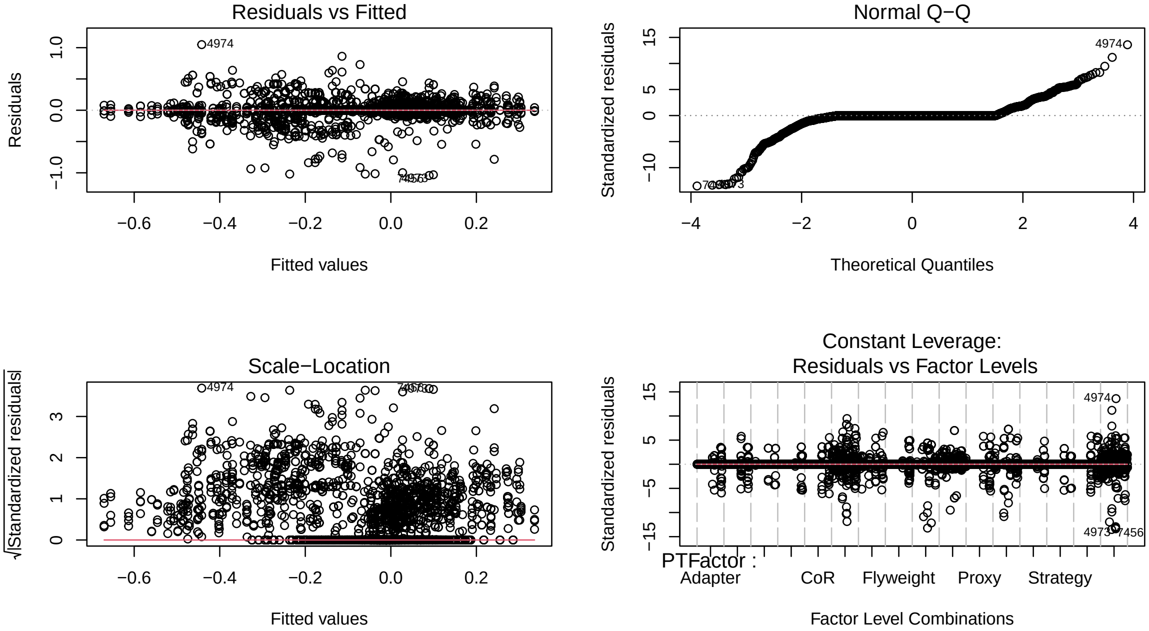


Figure 10.28: Modifiability diagnostic plots.

approximately -0.75) and occur across all pattern types except Prototype. Second, positive changes occur due to the injection of primarily Class and Modular grime. However, these changes only affect the Visitor, Singleton, Prototype, Observer, Decorator, and the Factory Method patterns.

10.4.4.2 Hypothesis Testing Initially, we begin the analysis by determining if using the parametric ANOVA approach is appropriate. We determined this by validating the fundamental assumptions of ANOVA. As noted above in Section 10.2.4, the two fundamental assumptions we are concerned with are the normality and homogeneity of variances assumptions.

Normality Assumption To evaluate this assumption, we plotted the ANOVA model, as depicted in Figure 10.28. The pertinent plot here is the “Normal Q-Q” Plot in the upper right quadrant. Here we see deviations from Normal in the tails of the data, which is a strong indicator of a violation of the normality assumption. This evidence is further confirmed using the Anderson-Darling normality test. The results of this test ($A = 3211$, $p < 2.2\text{e}{-16}$) provides strong evidence to reject the null hypothesis and further confirming the violation of the normality assumption.

Homogeneity of Variances Assumption This assumption is evaluated using a similar process as the Normality assumption. We again look to Figure 10.28, focusing on the "Residual vs. Fitted" plot in the upper-left quadrant. This plot indicates that there is a violation of the assumption. To analytically confirm this, we executed Levene's Test for Homogeneity of Variance. The results ($F(2495, 7488) = 2.5321$, $p < 2.2\text{e}{-16}$) of this test provides strong evidence to reject the null hypothesis that the variances are the same. These results further confirming this assumption has been violated.

Permutation F-Test Analysis The assumption validation steps result in the conclusion that we must either transform the data or use a permutation F-test approach. After several attempts to adjust for the violations, we moved forward with the permutation F-test approach. The overall results of this test ($F(2495, 7488) = 4.636$, $p < 2.2\text{e}{-16}$) indicates strong evidence to reject the null hypothesis that there is no difference in the mean change in Modifiability.

Interaction Effects With the knowledge that a difference in the mean change in Modifiability exists between two or more treatment combinations, we continue by considering any significant interactions. In this case, there is strong evidence ($p < 2.2\text{e}{-16}$) to reject $H_{2,0}$ that there is no difference in the mean change in Modifiability for each level of the three-way interaction effect. With this in mind, we will consider a graphical analysis of these interactions. To plot these interactions, we subdivided them into grime categories:

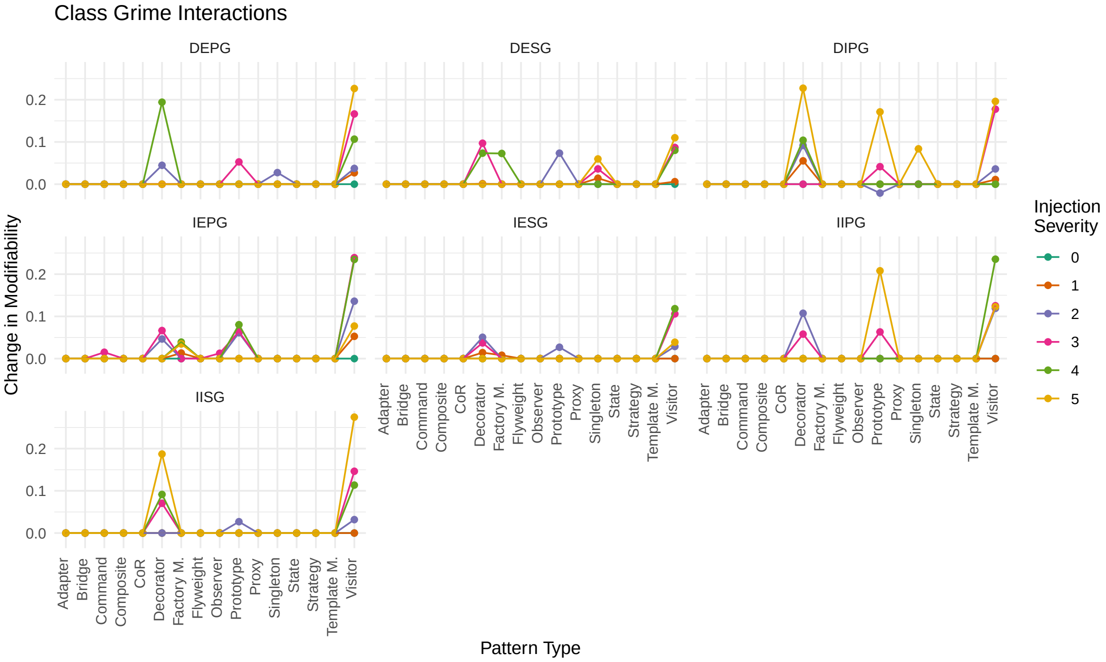


Figure 10.29: Modifiability interaction plots for class grime injection.

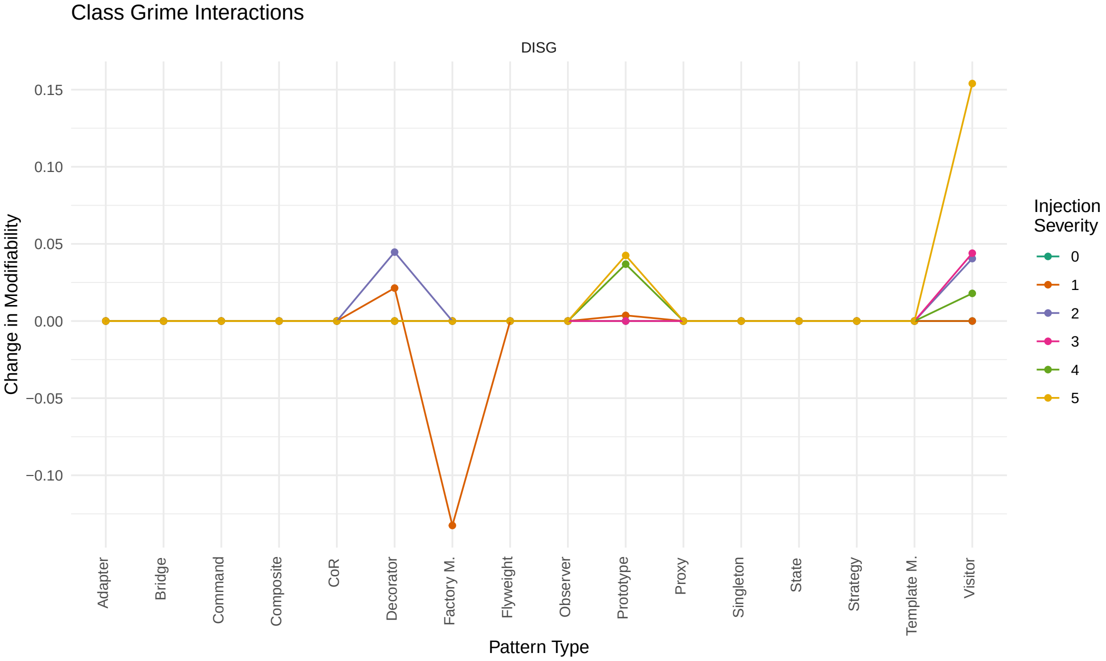


Figure 10.30: Modifiability interaction plots for DISG.

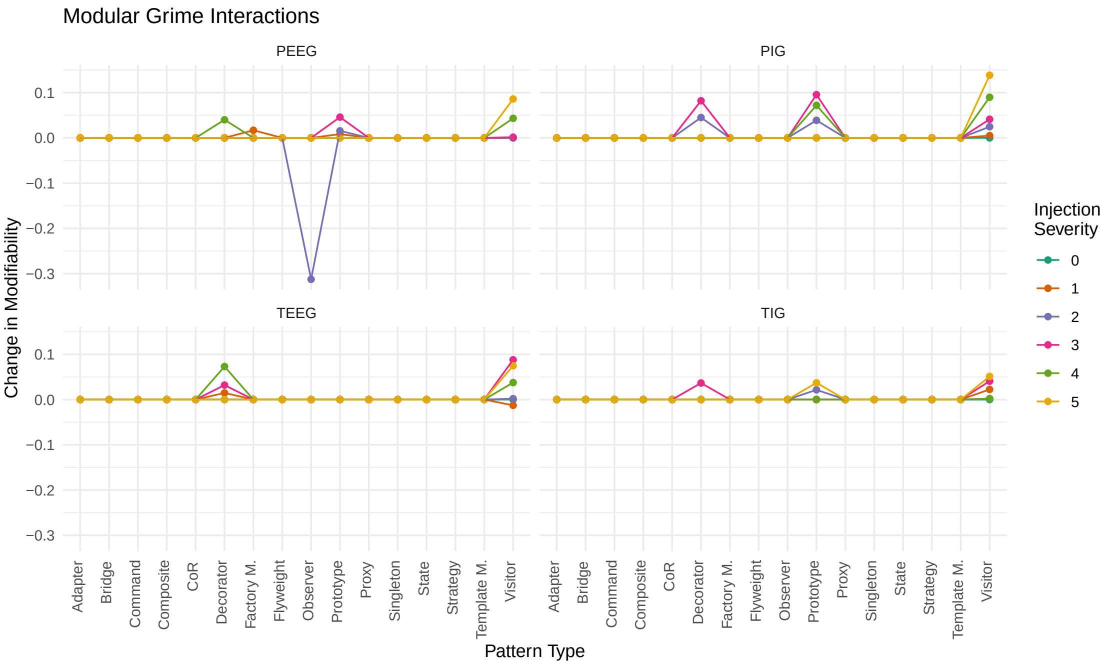


Figure 10.31: Modifiability interaction plots for modular grime injection.

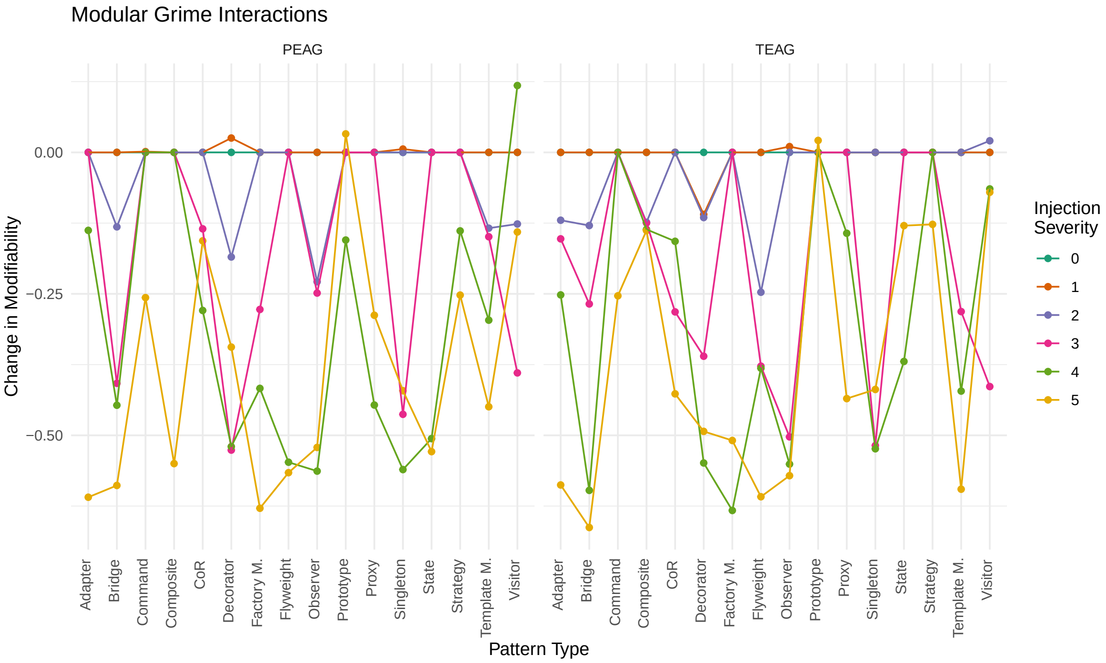


Figure 10.32: Modifiability interaction plots for PEAG and TEAG.

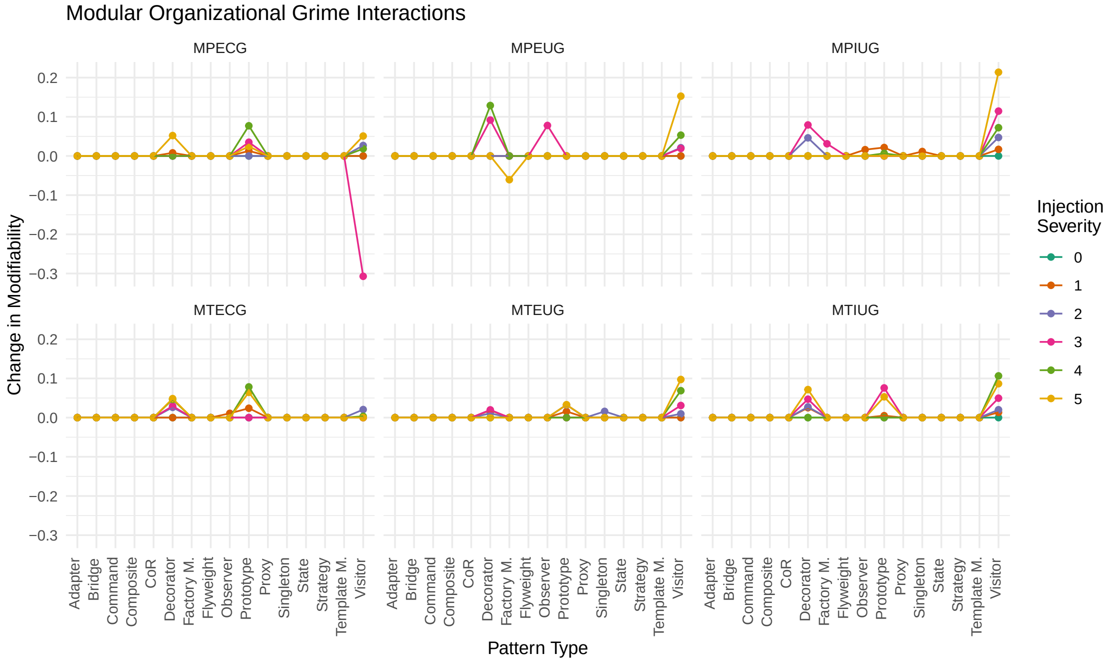


Figure 10.33: Modifiability interaction plots for modular organizational grime injection.

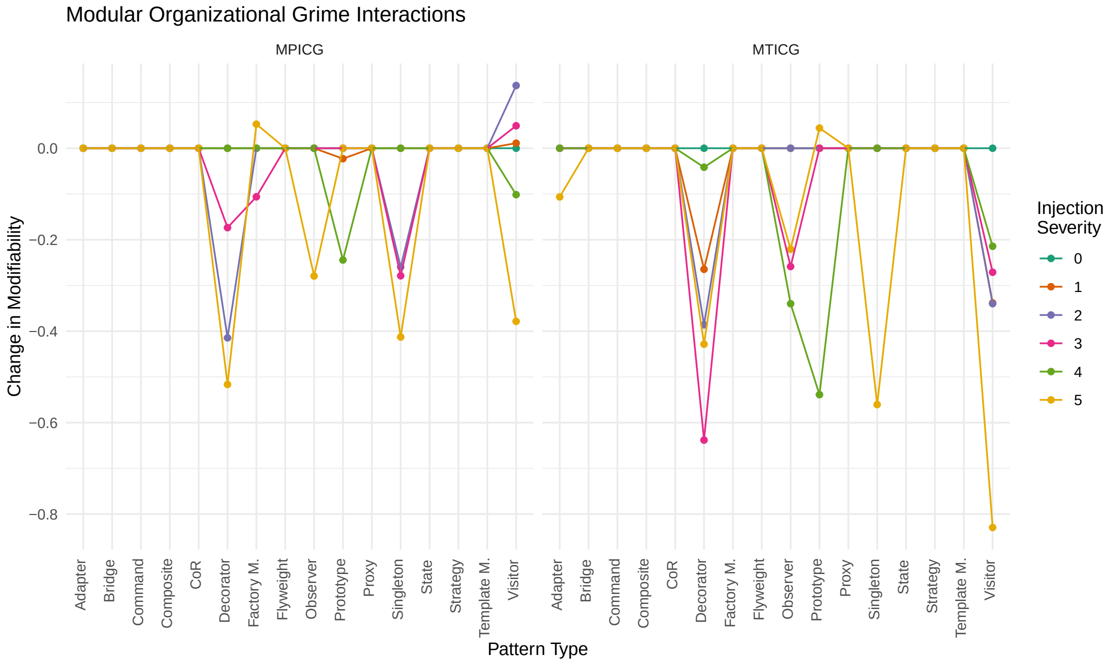


Figure 10.34: Modifiability interaction plots for MTEUG.

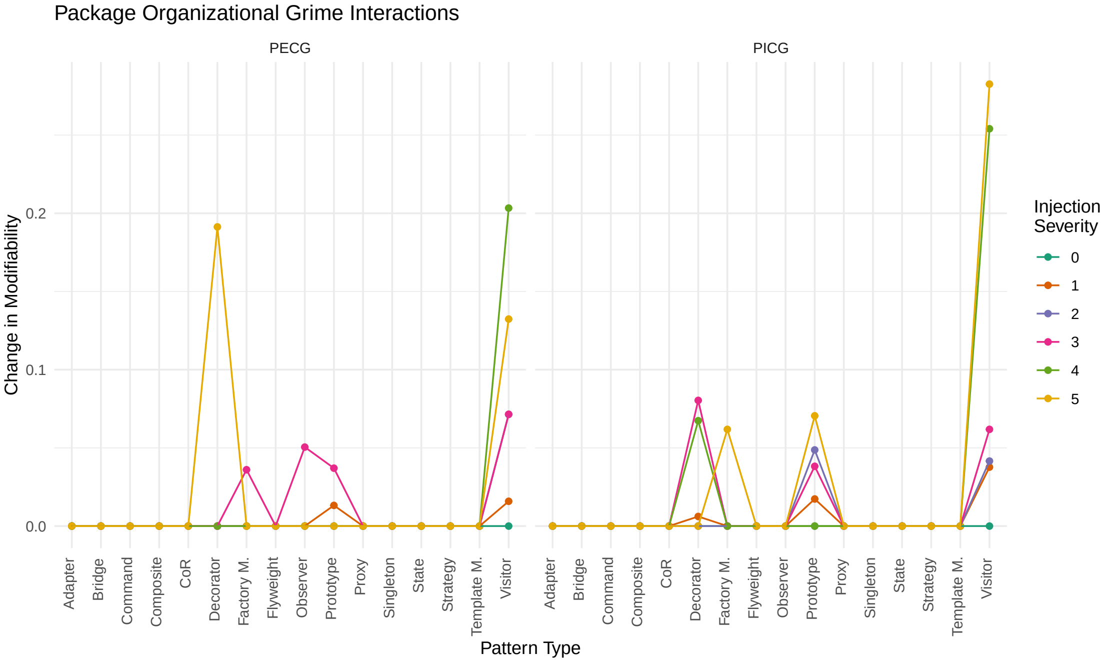


Figure 10.35: Modifiability interaction plots for PECG and PICG.

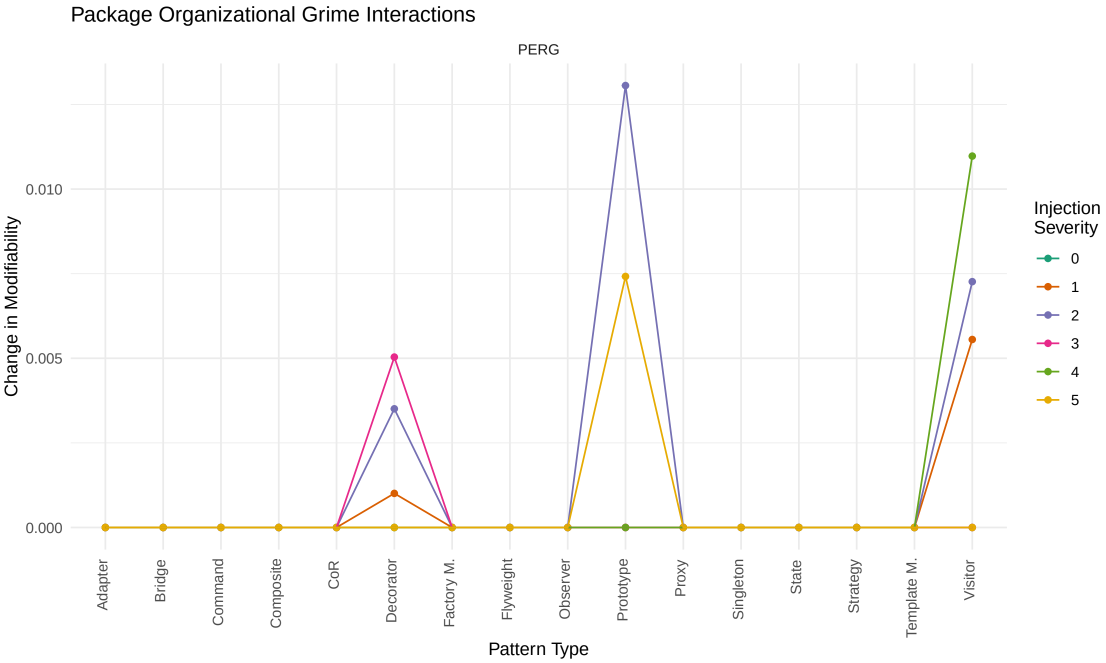


Figure 10.36: Modifiability interaction plots for PERG.

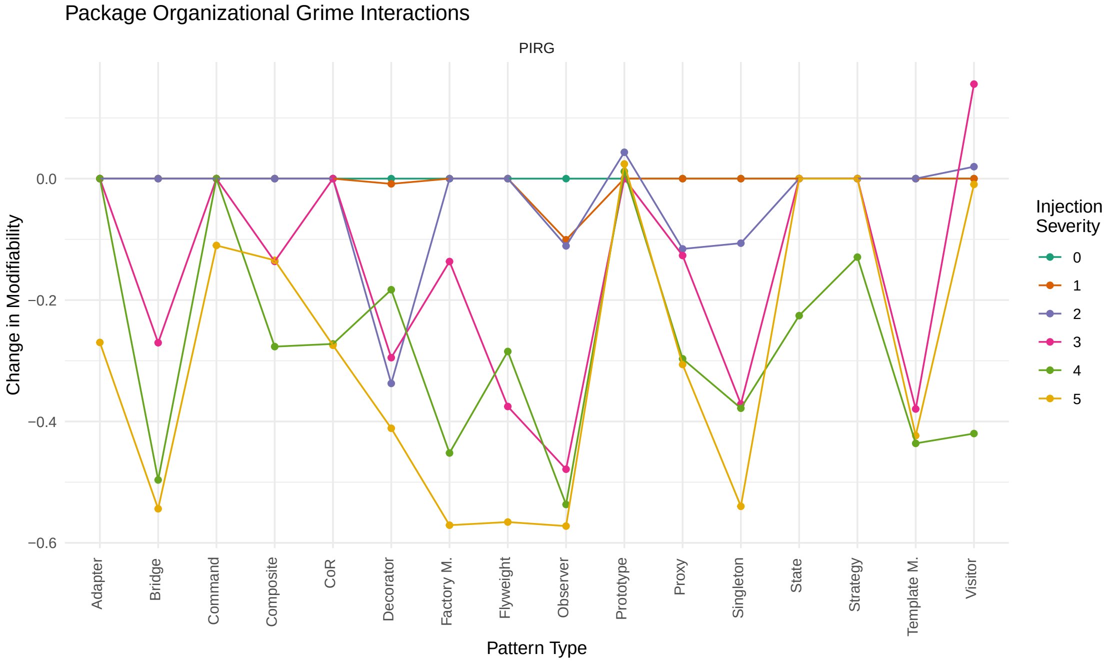


Figure 10.37: Modifiability interaction plots for PIRG.

Class Grime, Modular Grime, and Organizational Grime. Each grime category plot contains a matrix of subplots (one per grime type in the category). The y-axis is the change in Modifiability, the x-axis is the design pattern type, and the points plotted are the values for each injection severity. We will begin with the plots for Class Grime.

We begin with the interaction plots for Class Grime as depicted in Figures 10.29 and 10.30. This figure shows us that not all pattern types afflicted with class grime have an impact on Modifiability. However, it does suggest that for several levels of injection severity, Decorator, Prototype, and Visitor afflicted with Class Grime will have a positive change in Modifiability. Additionally, there is evidence, as depicted in Figure 10.30, that for DISG subtype with Injection Severity level 1 that Factory Method instances negatively impact the Change in Modifiability. Finally, there appears to be no direct relationship between the severity and the change in Modifiability.

Next, we look in detail at the interactions associated with Modular Grime types as depicted in Figures 10.31 and 10.32. Figure 10.32 shows some interesting interactions. We begin by considering the two afferent grime subtypes PEAG and TEAG. In these two cases, injection severity levels of at least four across all pattern types (excluding Prototype) presents significant negative changes in Modifiability. This relationship is in stark contrast to the other four subtypes, depicted in 10.31, which appear to be more similar to the class grime interactions. Again, for these four subtypes (PEEG, TEEG, PIG, and TIG), they have a mostly positive effect on Modifiability and are focus on only a few pattern types, namely Decorator, Prototype, and Visitor. Additionally, there appears to be no direct relationship between injection severity and the change in Modifiability.

Next, we look at Modular Organizational Grime type interactions as depicted in Figures 10.33 and 10.34. Notably, across all subtypes of grime depicted, the patterns Decorator, Prototype, and visitor seem to be the key pattern types. We also note that internal cyclic forms of Modular Organizational Grime (namely MPICG and MTICG) appear to have a negative effect, whereas the other subtypes have a predominately positive effect. However, there appears to be no relationship between injection severity and the effect on the Change in Modifiability for these same subtypes. Additionally, we note that other pattern types appear to be prominent for the internal cyclic forms, particularly Observer, Singleton, and Adapter.

Finally, we look at Package Organizational Grime type interactions as depicted in Figure 10.35. This figure shows that for PECG, PERG, and PICG subtypes, Decorator, Prototype, and Visitor again provide positive effects. Additionally, for the cyclic subtypes (PECG and PICG) Factory Method shows some involvement. The remaining subtype, PIRG, shows significant effects on Modifiability across all pattern types with predominantly adverse effects. The only exceptions to this are the positive effect found for Prototype and Visitor. Thus, again, there appears to be no relationship between injection severity and the change in Modifiability.

Table 10.6: Summary of Modularity data.

Characteristic	Min	Median	Mean	Max	SD
ΔModularity	-2.64236	0.0	-0.6583	1.23418	0.5395721

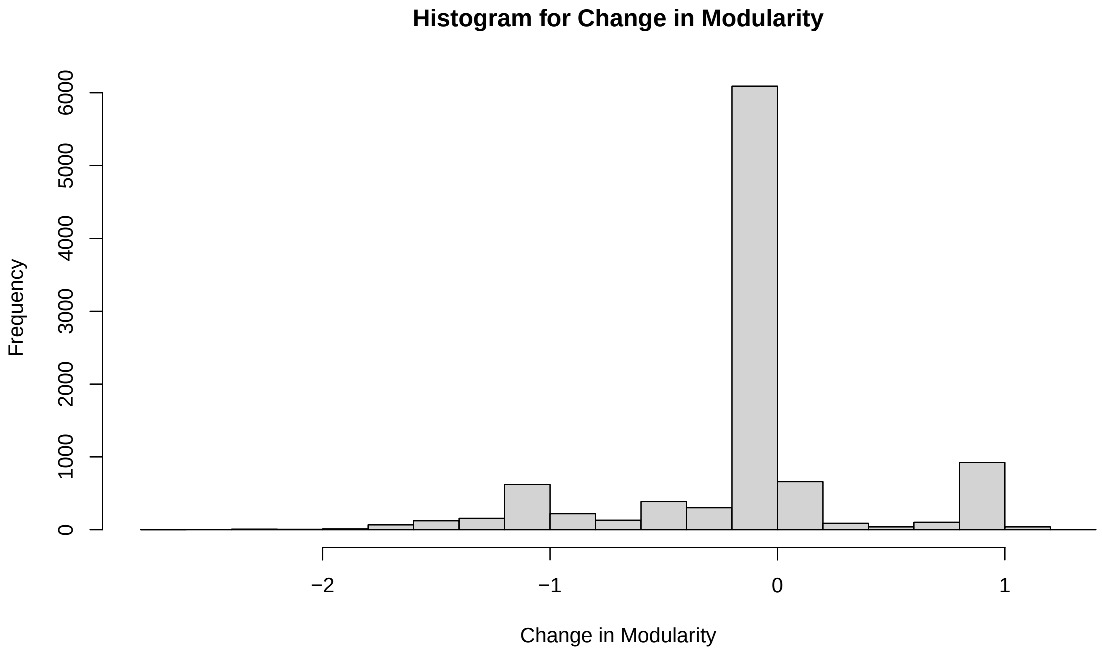


Figure 10.38: Histogram of the change in Modifiability.

10.4.5 Modularity

This subsection describes the results of the Modularity analysis. We subdivided the analysis into a subsection describing the data and descriptive statistics, and a subsection describing hypothesis testing.

10.4.5.1 Descriptive Statistics This section presents the results of the Modularity experiment using descriptive statistics and plots. First, we show the summary of the Change in Modularity (the dependent variable) in Table 10.6. The table shows the basic statistics

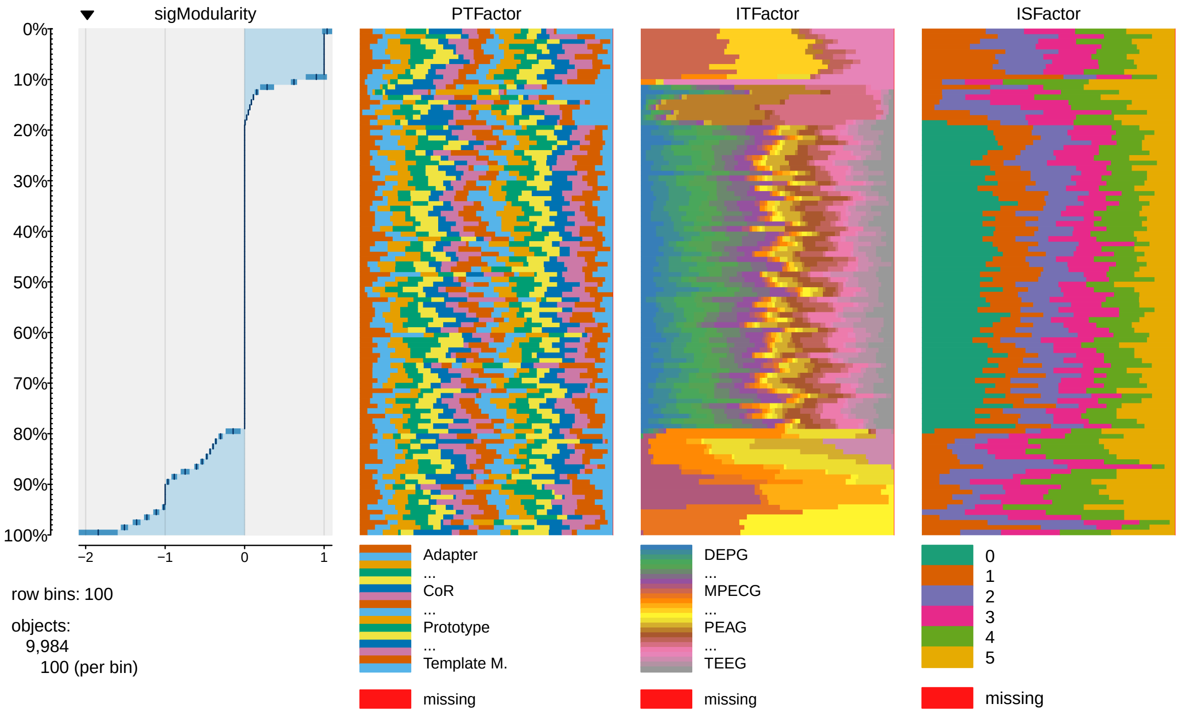


Figure 10.39: Table plot of Modularity data.

across the 9,984 observations. This table suggests that across all observations, the change in Modularity ranges between -2.64236 and 1.23418, and the mean change in Modularity is -0.06583. However, given the distribution of the values being skewed to the right as depicted by the histogram in Figure 10.38, the median value of 0.0 provides a better measure of the centrality of the data. Combining all of this with the standard deviation of 0.5395721, we know the following about this data: i) the majority of the observations showed no change to Modularity; ii) of those observations that showed any change in Modularity, it can be either negative or positive and that the magnitude is greater in the negative direction; and iii) there were some observations which show significant changes in Modularity both in the positive and negative directions. To better understand how this data is distributed, in the context of the independent variables, we constructed two plots: the first is a table plot (see Figure 10.39), and the second is a scatterplot (see Figure 10.40).

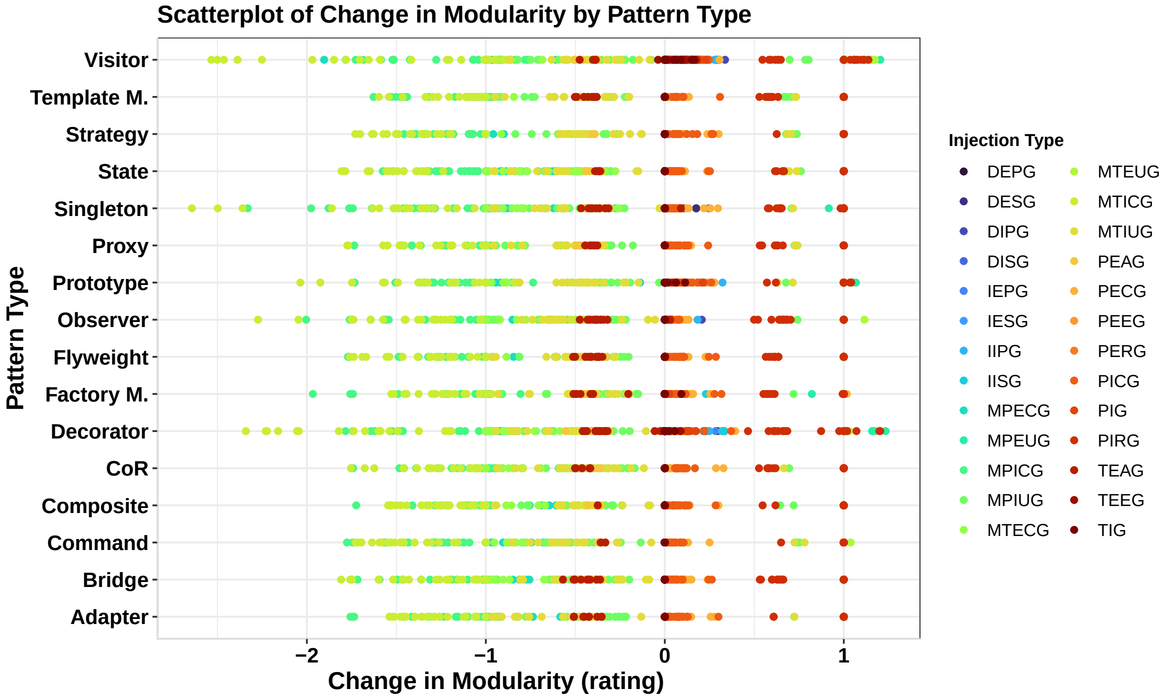


Figure 10.40: Scatterplot of the Change in Modifiability and Pattern Type.

Figure 10.39 depicts a table plot of the dependent and each of the independent variables. Each column of this plot represents a single variable, while each plot row represents a sample of the data. The first column presents a histogram of the Change in Modularity (sigModularity) separated into 100 bins each containing 100 observations. The remaining columns show the values of Pattern Type (PTFactor), Injection Type (ITFactor), and Injection Severity (ISFactor) for the 100 values for each row of the Change in Modularity. This data view allows us to see the distribution of the data and any interesting patterns that may exist across the columns.

In this plot, we initially see that approximately 20% of the change in Modularity is positive, 22% is negative, and the remaining is zero. Between the 0% and approximately 20% marks the change in Modularity is positive and appears to related to only a subset of the injection types. Of these changes there are two groups of injection types that affect the changes separated by apparent difference in the magnitude of the change. However, these changes do not appear to be related to any of the other independent variables. Additionally, the last approximately 22% of the data indicates a negative change in Modularity and appears to be related to a different subset of Injection Types, and again there two groups apparently different in the magnitude of the changes affected. Again, there does not appear to be any relation to either the Pattern Type or the Injection Severity. The remaining 58% of the data has a value of zero. This data is the only data where the Injection Severity level of 0 occurs. Furthermore, it appears that a change of zero cuts across all other Injection Severity levels, all Pattern Types, and all Injection Types as well.

Figure 10.40 shows the scatterplot of the Change in Modularity by Pattern Type, with each point colored according to the Injection Type. This plot shows several key things. First, negative changes occur across all Pattern Types and all Injection Types. However, we can see that the largest magnitude of change is the injection of primarily Modular Organizational Grime (ranging from just below zero to just above -2.5). Additionally, smaller negative changes can be seen and appear to be due to the injection of Modular Grime (ranging from approximately -0.25 to -0.5). Furthermore, there appears to be significant positive changes due to Modular Grime and Package Organizational Grime ranging from just above 0.0 to 1.0.

10.4.5.2 Hypothesis Testing Initially, we begin the analysis by determining if using the parametric ANOVA approach is appropriate by validating its fundamental assumptions. As noted above in Section 10.2.4, the two fundamental assumptions we are concerned with are the normality and homogeneity of variances assumptions.

Normality Assumption To evaluate this assumption, we plotted the ANOVA model, as depicted in Figure 10.18. The pertinent plot here is the “Normal Q-Q” Plot in the upper right quadrant. Here we see deviations from Normal in the tails of the data, which is a strong

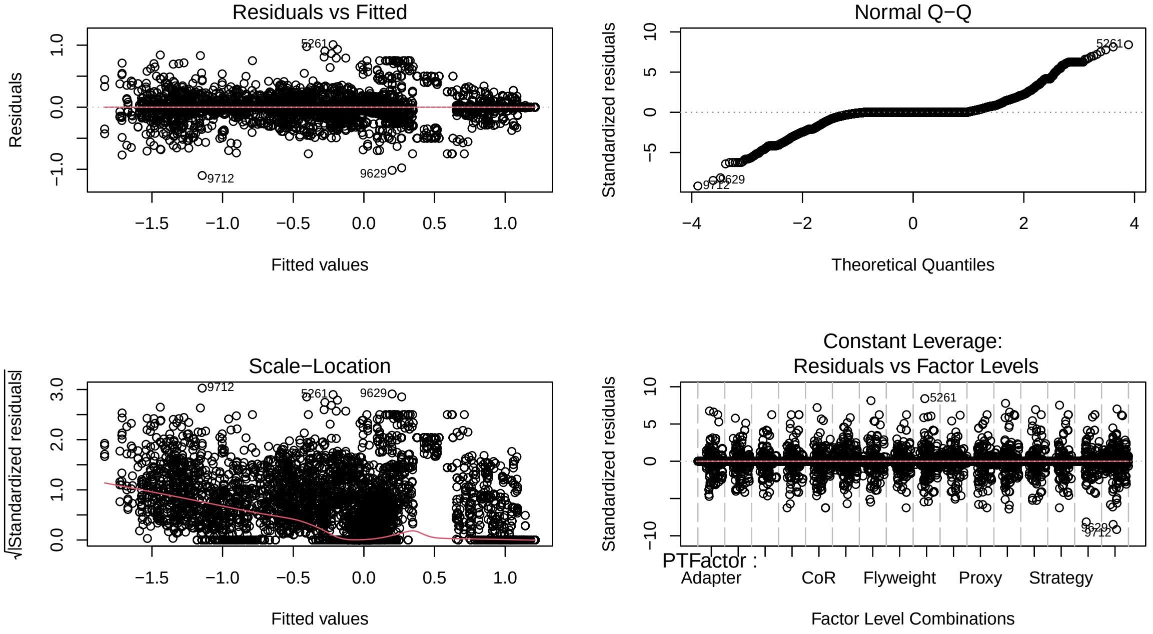


Figure 10.41: Modularity diagnostic plots.

indicator of a violation of the normality assumption. This evidence is further confirmed using the Anderson-Darling normality test. The results of this test ($A = 1122.3$, $p < 2.2\text{e}{-16}$) provides strong evidence to reject the null hypothesis and further confirming the violation of the normality assumption.

Homogeneity of Variances Assumption This assumption is evaluated using a similar process as the Normality assumption. We again look to Figure 10.18, focusing on the “Residual vs. Fitted” plot in the upper-left quadrant. This plot indicates that there is a violation of the assumption. To analytically confirm this, we executed Levene’s Test for Homogeneity of Variance. The results ($F(2495, 7488) = 3.1551$, $p < 2.2\text{e}{-16}$) of this test provides strong evidence to reject the null hypothesis that the variances are the same. These results further confirming this assumption has been violated.

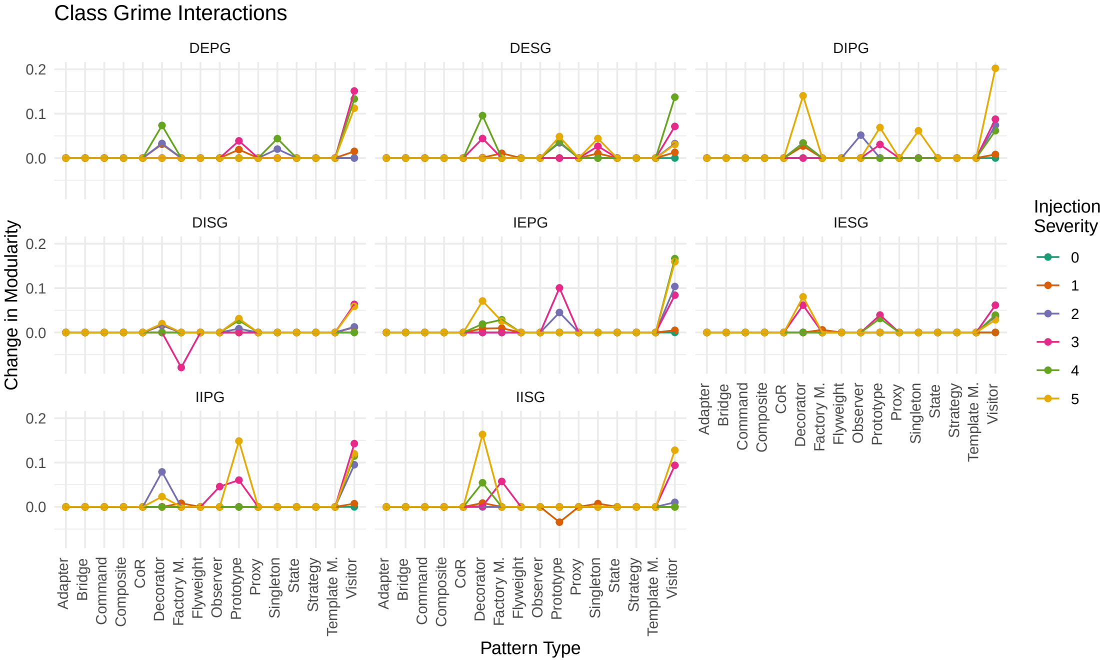


Figure 10.42: Modularity interaction plots for class grime injection.

Permutation F-Test Analysis The assumption validation steps indicate that we must either transform the data or use a permutation F-test approach. After several attempts to adjust for the violations, we moved forward with the permutation F-test approach. The overall results of this test ($F(2495, 7488) = 57.63$, $p < 2.2\text{e}{-16}$) indicates strong evidence to reject the null hypothesis that there is no difference in the mean change in Modularity.

With the knowledge that a difference in the mean change in Modularity exists between two or more treatment combinations, we continue by considering any significant interactions. In this case, there is strong evidence ($p < 2.2\text{e}{-16}$) to reject $H_{2,0}$ that there is no difference in the mean change in Modularity for each level of the three-way interaction effect. With this in mind, we will consider a graphical analysis of these interactions. To plot these interactions, we subdivided them into grime categories: Class Grime, Modular Grime, and Organizational Grime. Each grime category plot contains a matrix of subplots (one per grime type in the category). In these plots, the y-axis is the change in Modularity, the x-axis is the design

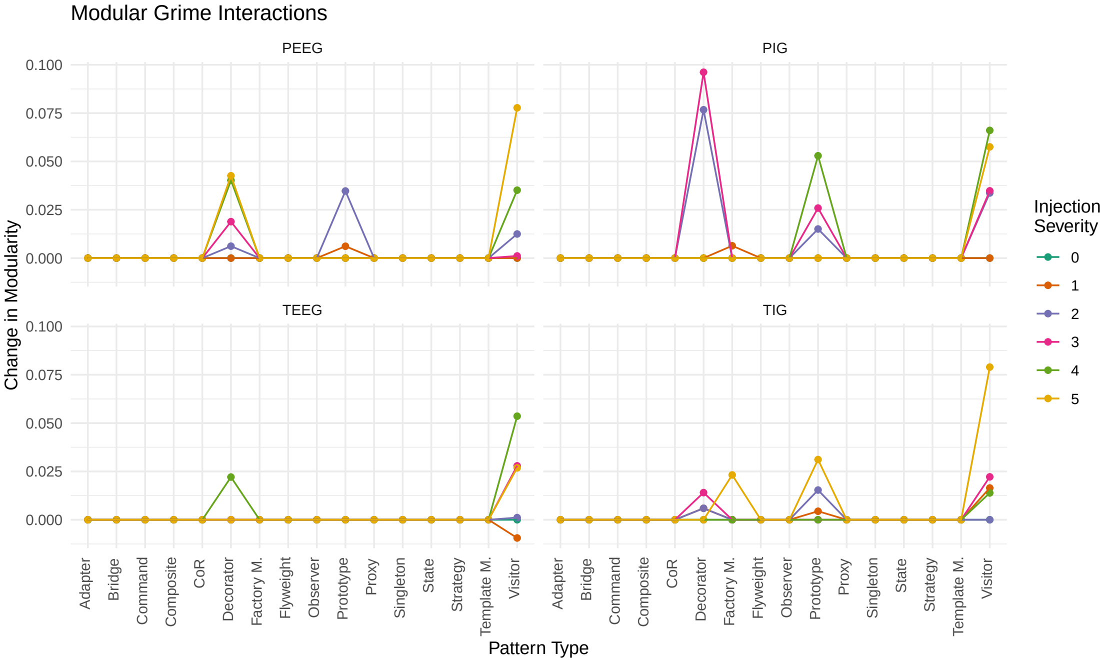


Figure 10.43: Modularity interaction plots for modular grime injection.

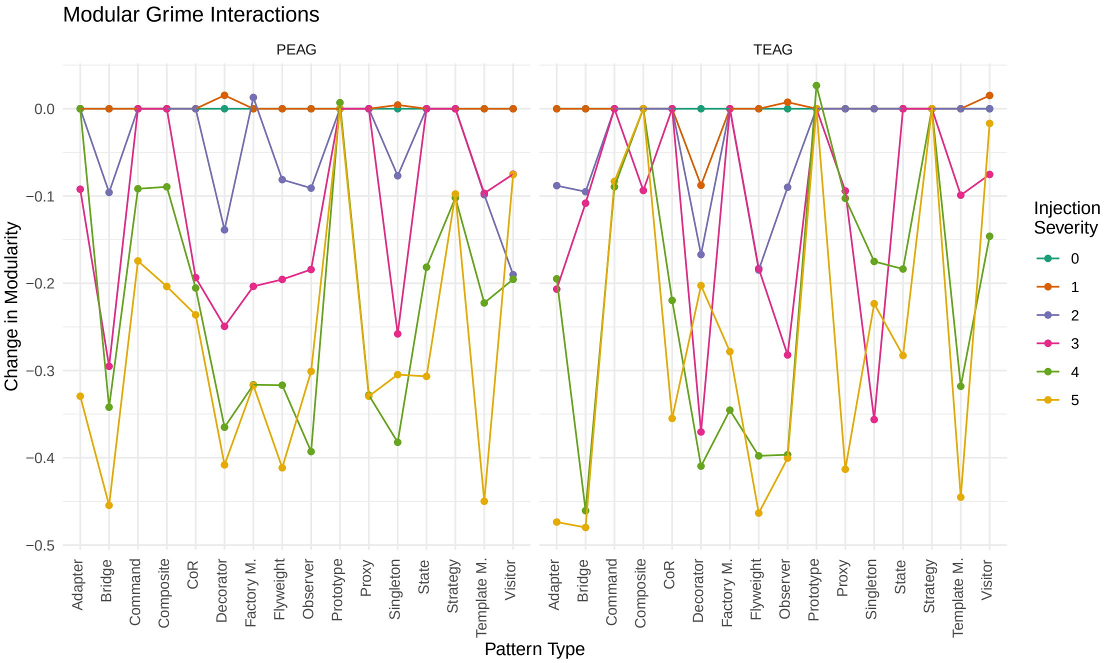


Figure 10.44: Modularity interaction plots for PEAG and TEAG.

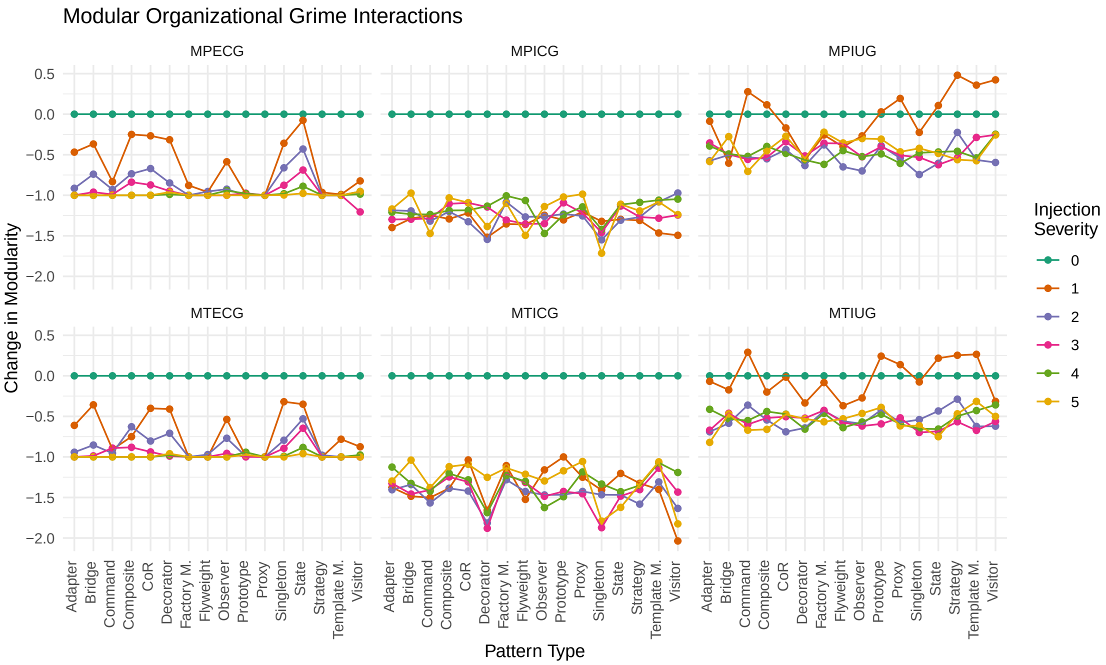


Figure 10.45: Modularity interaction plots for modular organizational grime injection.

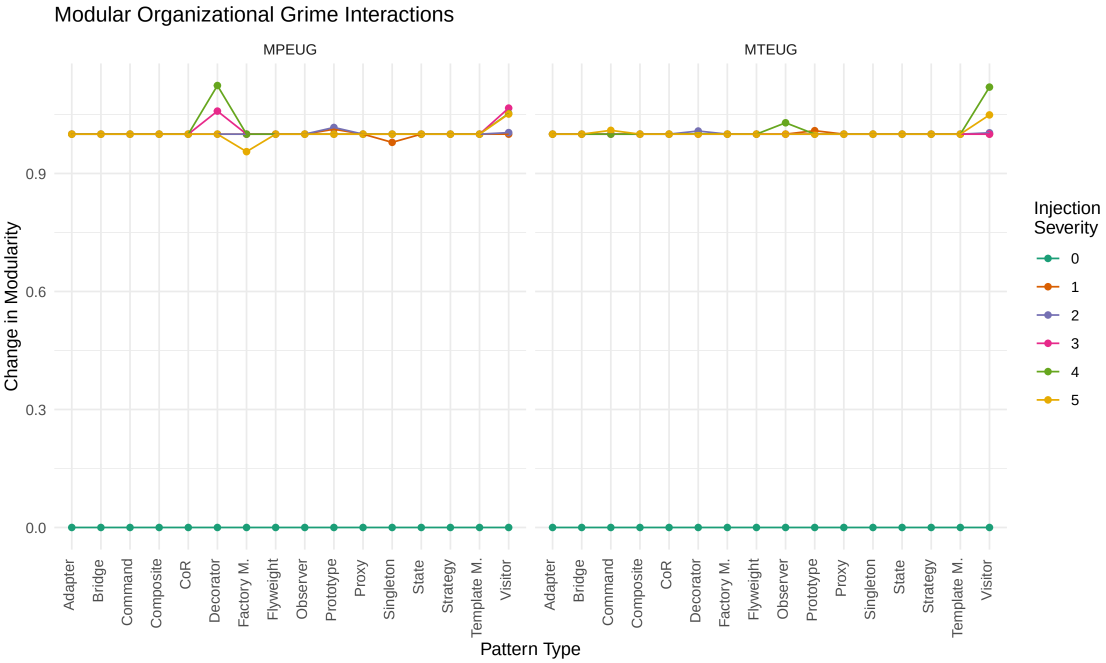


Figure 10.46: Modularity interaction plots for MTEUG.

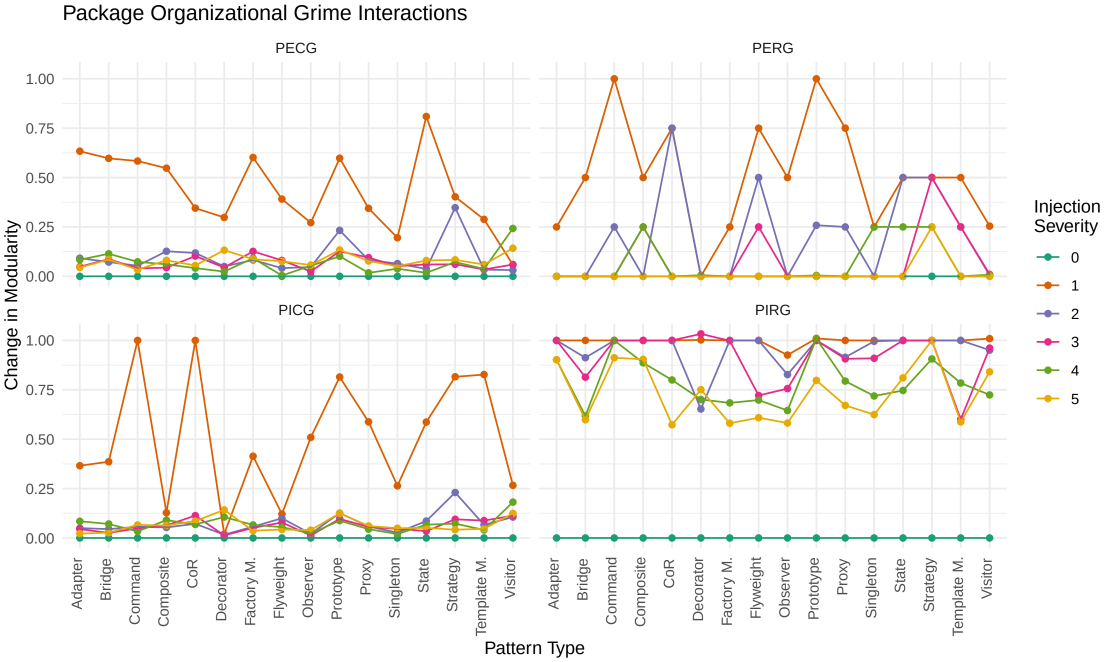


Figure 10.47: Modularity interaction plots for package organizational grime injection.

pattern type, and the points plotted are the values for each injection severity. We will begin with the plots for Class Grime.

We begin with the interaction plots for Class Grime as depicted in Figure 10.42. This figure shows us that not all pattern types afflicted with class grime have an impact on Modifiability. However, it does suggest that for several levels of injection severity, Decorator, Prototype, and Visitor afflicted with Class Grime will have a positive change in Modularity. Interestingly, there appears to be no direct relationship between the severity and the change in Modularity.

Next, we look in detail at the interactions associated with Modular Grime types as depicted in Figures 10.43 and 10.44. Here there are some interesting interactions. Specifically, we consider the two afferent grime subtypes PEAG and TEAG. In these two cases, when injection severity is at least four across all pattern types (excluding Prototype), there are significant negative changes in Modularity. This finding is in stark contrast to the other four subtypes, which appear to be more similar to the class grime interactions. Again, these four subtypes (PEEG, TEEG, PIG, and TIG) have a mostly positive effect on Modularity and focus on only a few pattern types, namely Decorator, Prototype, and Visitor. Additionally, there appears to be no direct relationship between injection severity and the change in Modularity.

Next, we look at Modular Organizational Grime type interactions as depicted in Figures 10.45 and 10.46. Again, we note that only a few key patterns appear to affect a change in Modularity when afflicted with these subtypes of grime. Notably, across all subtypes of grime depicted, the patterns Decorator, Prototype, and visitor seem to be the key pattern types. We also note that internal cyclic forms of Modular Organizational Grime (namely MPICG and MTICG) appear to have a negative effect, whereas the other subtypes have a predominately positive effect. Additionally, we note that other pattern types appear to be prominent for the internal cyclic forms, particularly Observer, Singleton, and Adapter. Additionally, looking at Figure 10.46 we can see that it appears that there is minimal variability across Pattern Types and Injection Severity levels concerning the Change in Modularity. Rather it appears that for all levels, MTEUG affects nearly 1-star positive change in Modularity.

Finally, we look at Package Organizational Grime type interactions as depicted in Figure 10.47. This figure shows that for PECG, PERG, and PICG subtypes, Decorator, Prototype, and Visitor again provide positive effects. Additionally, for the cyclic subtypes (PECG and PICG) Factory Method shows some involvement. The remaining subtype, PIRG, shows significant effects on Modularity across all pattern types with predominantly negative effects. The only exceptions to this are the positive effect found for Prototype and Visitor. Again, there appears to be no relationship between injection severity and the change in Modularity.

Table 10.7: Summary of Reusability data.

Characteristic	Min	Median	Mean	Max	SD
ΔReusability	0.0	0.0	0.0	0.0	0.0

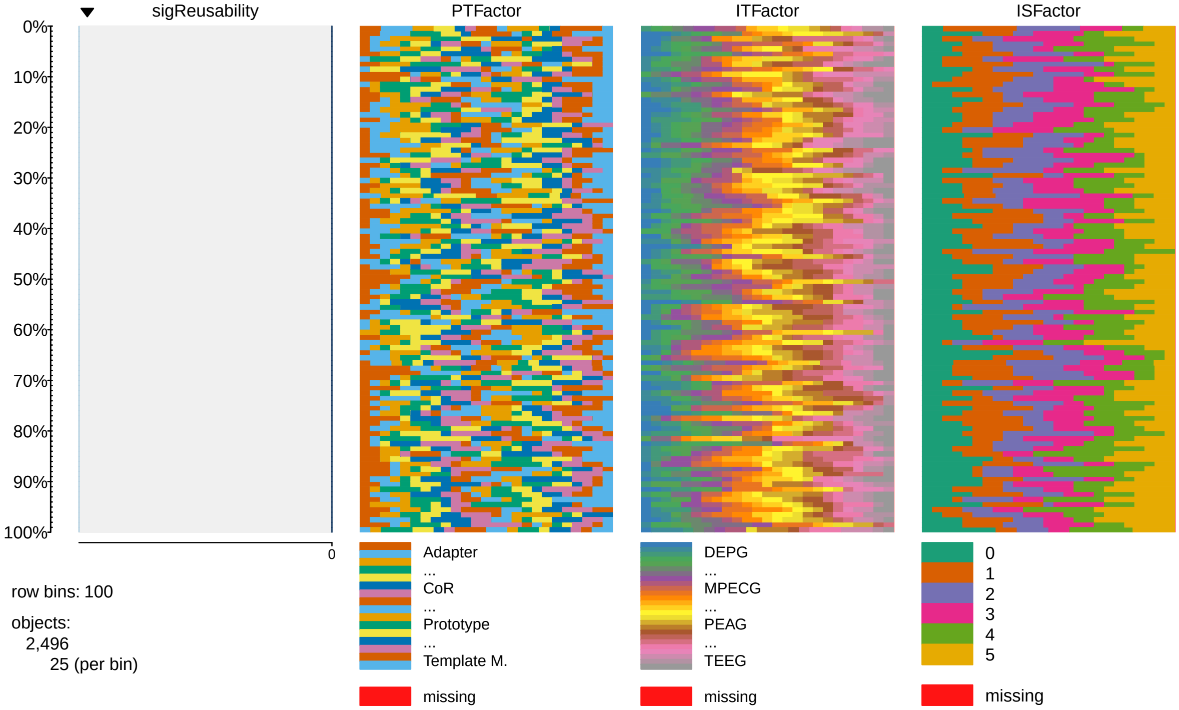


Figure 10.48: Table plot of Reusability data.

10.4.6 Reusability

This subsection describes the results of the Reusability analysis. We subdivided the analysis into a subsection describing the data and descriptive statistics, and a subsection concerns hypothesis testing.

10.4.6.1 Descriptive Statistics This section presents the results of the Reusability experiment using descriptive statistics and plots. First, we show the summary of the Change in Reusability (the dependent variable) in Table 10.7. The table shows the basic statistics

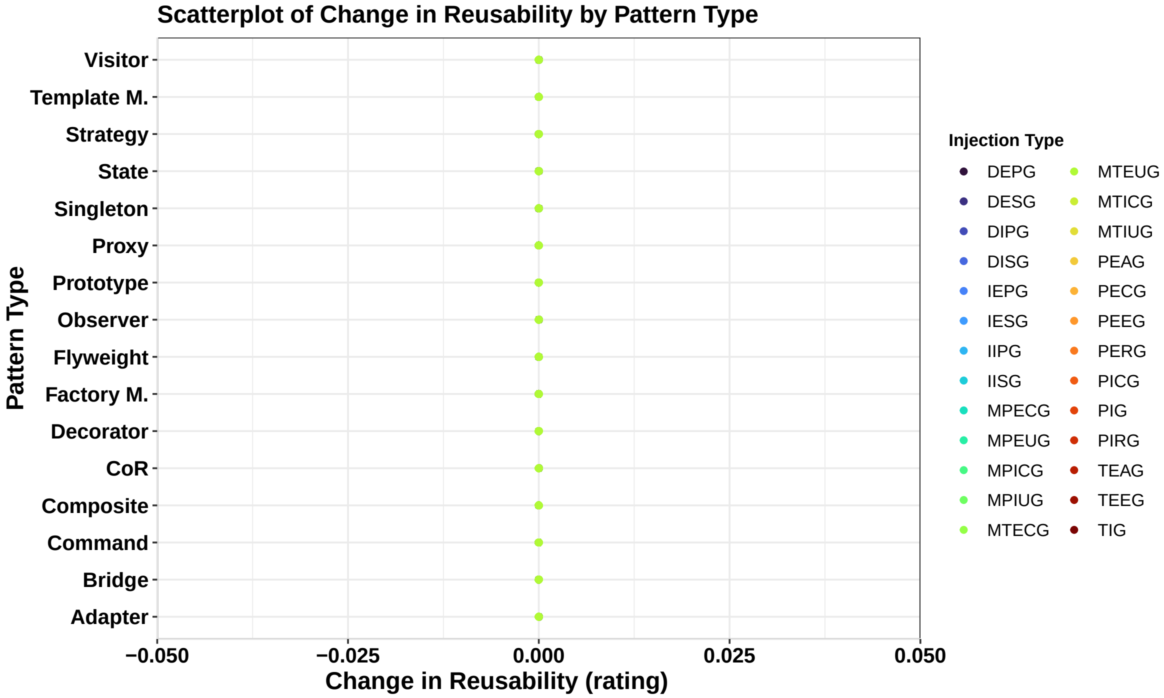


Figure 10.49: Scatterplot of the Change in Reusability and Pattern Type.

across the 2,496 observations. This table suggests that across all observations, the change in Reusability was 0. This observation is further confirmed in both a table plot (see Figure 10.48), and a scatterplot (see Figure 10.49).

Figure 10.48 depicts a table plot of the dependent and each of the independent variables. Each column of this plot represents a single variable, while each plot row represents a sample of the data. The first column presents a histogram of the Change in Reusability (sigReusability) separated into 100 bins each containing 25 observations. The remaining columns show the values of Pattern Type (PTFactor), Injection Type (ITFactor), and Injection Severity (ISFactor) for the 25 values for each row of the Change in Reusability. This data view allows us to see the distribution of the data and any interesting patterns that may exist across the columns. In this plot, we initially see that approximately all of the change in Reusability is zero. This is further confirmed in Figure 10.49. This figure shows that across all pattern types and all injection types the change in reusability is zero.

10.4.6.2 Hypothesis Testing Initially, we begin the analysis by determining if using the parametric ANOVA approach is appropriate. We determined this by validating the fundamental assumptions of ANOVA. As noted above, the two fundamental assumptions we are concerned with are normality and homogeneity of variances assumptions.

Normality Assumption Due to the nature of the data, all values for the change in Reusability being zero, we were unable to evaluate this assumption.

Homogeneity of Variances Assumption Due to the nature of the data, all values for the change in Reusability being zero, we were unable to evaluate this assumption.

Permutation F-Test The nature of the data collected suggests no measurable effect on Reusability when any form of structural grime is injected. To evaluate this, we conducted a permutation F-test with the null hypothesis that there is no difference in the mean change in Reusability due to any combination of pattern type, injection type, and injection severity. Unfortunately, the data did permit the evaluation of this.

10.4.7 Technical Debt Principal

This subsection describes the results of the Technical Debt Principal analysis. We subdivided the analysis into a subsection describing the data and descriptive statistics, and a subsection describing hypothesis testing.

10.4.7.1 Descriptive Statistics This section presents the results of the Technical Debt Principal experiment using descriptive statistics and plots. First, we show the summary of the Change in Technical Debt Principal (the dependent variable) in Table 10.8. The table shows the basic statistics across the 4,992 observations. This table suggests that across all

Table 10.8: Summary of TD Principal data.

Characteristic	**Min**	**Median**	**Mean**	**Max**	**SD**
ΔTD Principal	-0.9351	0.1002	0.3206	12.7850	0.6634359

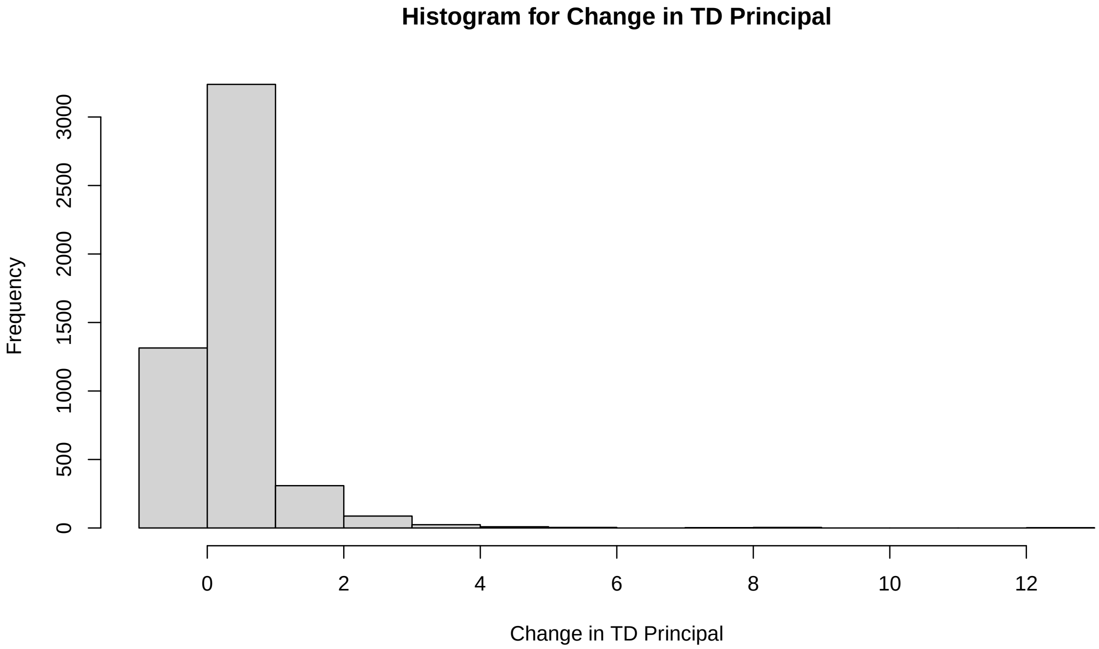


Figure 10.50: Histogram of the change in TD Principal.

observations, the change in Technical Debt Principal ranges between -0.9351 and 12.7850 man-months, and the mean change in Technical Debt Principal is 0.3206 man-months. However, given the distribution of the values being heavily skewed to the left as depicted by the histogram in Figure 10.50, the median value of 0.1002 man-months provides a better measure of the centrality of the data. Combining all of this with the standard deviation of 0.6634359, we know the following about this data: i) the majority of the observations showed a change to Technical Debt Principal; ii) of those observations that showed any change in Technical Debt Principal, it can be either negative or positive and that the magnitude is

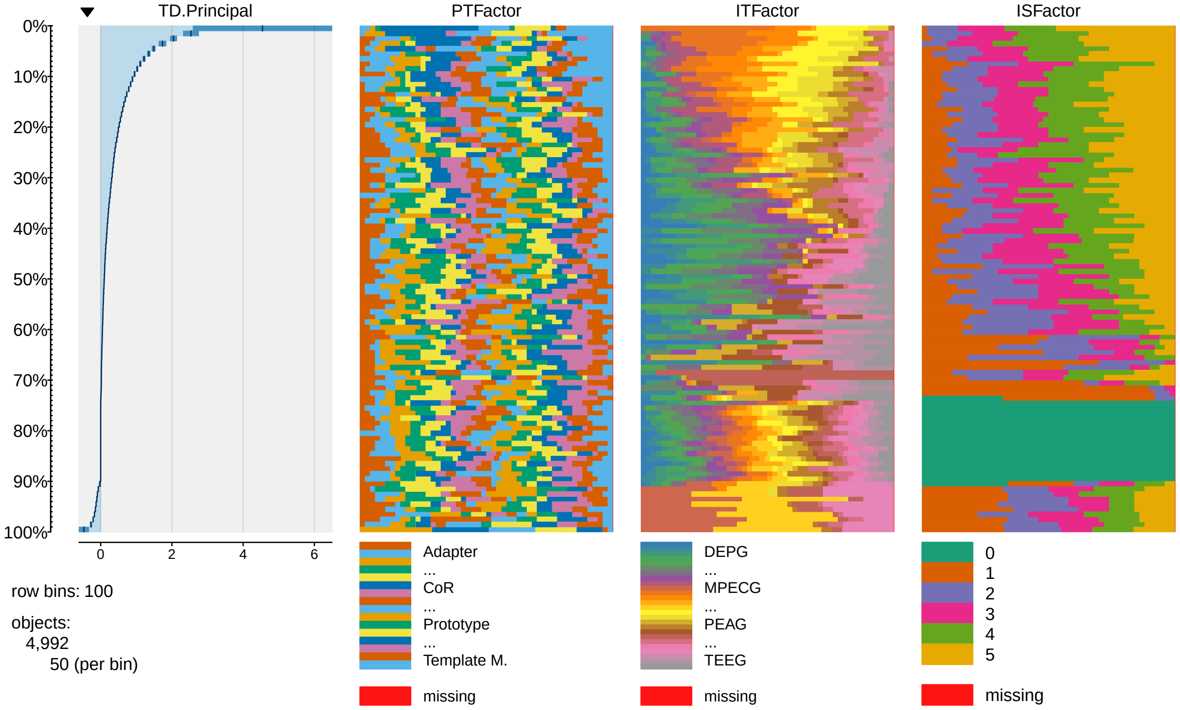


Figure 10.51: Table plot of TD Principal data.

greater in the positive direction; and iii) there were some observations which show significant changes in Technical Debt Principal both in the positive and negative directions. To better understand how this data is distributed, in the context of the independent variables, we constructed two plots: the first is a table plot (see Figure 10.51), and the second is a scatterplot (see Figure 10.52).

Figure 10.51 depicts a table plot of the dependent and each of the independent variables. Each column of this plot represents a single variable, while each plot row represents a sample of the data. The first column presents a histogram of the Change in Technical Debt Principal (TD.Principal) separated into 100 bins each containing 50 observations. The remaining columns show the values of Pattern Type (PTFactor), Injection Type (ITFactor), and Injection Severity (ISFactor) for the 50 values for each row of the Change in Technical Debt Principal. This data view allows us to see the distribution of the data and any interesting

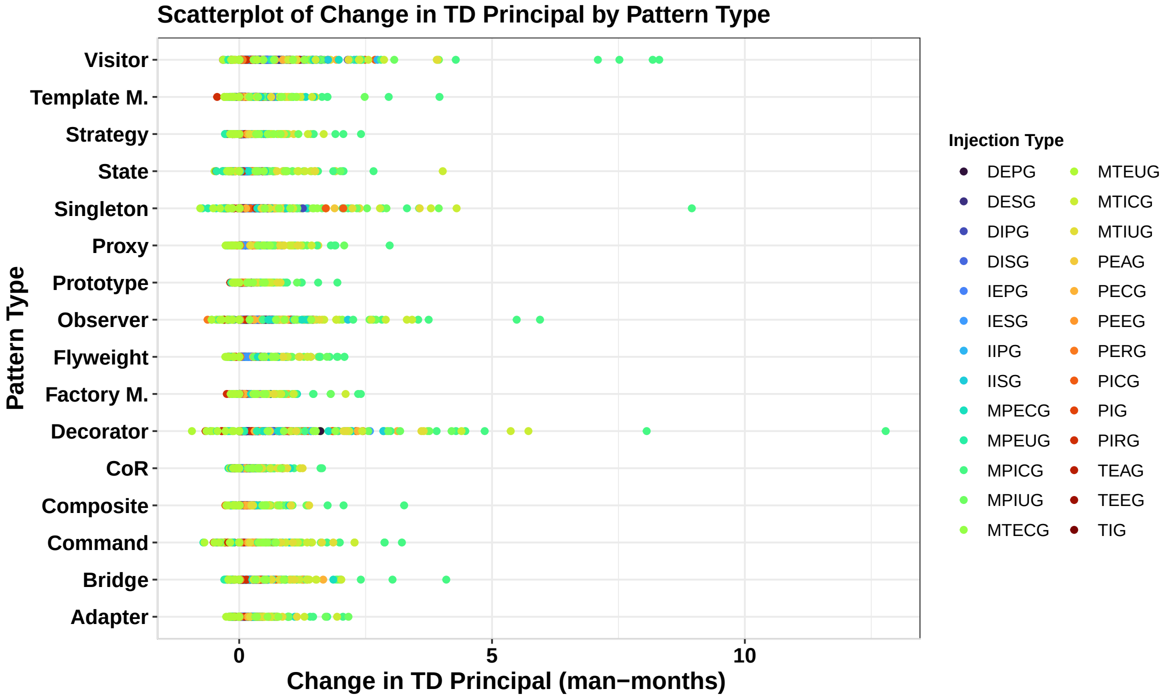


Figure 10.52: Scatterplot matrix of the Change in TD Principal and Pattern Type.

patterns that may exist across the columns.

In this plot, we initially see that approximately 70% of the change in Technical Debt Principal is positive, 10% is negative, and the remaining is zero. Between the 0% and approximately 70% marks the change in Technical Debt Principal is positive and the largest magnitude changes appear to be related to only a subset of the injection types. Additionally, the smallest magnitude positive changes appear to be due in large part to Injection Severity level 1 injections. Beyond these apparent relationships, there does not appear to be any other patterns related to the positive changes in Technical Debt Principle. The last approximately 10% of the data indicates a negative change in Technical Debt Principal and appears to be related to a different subset of Injection Types, and again there two groups apparently different in the magnitude of the changes affected. Again, there does not appear to be any relation to either the Pattern Type or the Injection Severity. The remaining approximately

20% of the data has a value of zero. This data is the only data where the Injection Severity level of 0 occurs. Furthermore, it appears that a change of zero cuts across all other Injection Severity levels, all Pattern Types, and all Injection Types as well.

Figure 10.52 shows the scatterplot of the Change in TD Principal by Pattern Type, with each point colored according to the Injection Type. This plot shows several key things. First, both large positive and small negative changes occur across all Pattern Types and all Injection Types. Additionally, we can see that the largest magnitude of change is the injection of primarily Modular Organizational Grime, with the largest changes due to MTICG. Additionally, the largest spikes in TD Principal occur for the Visitor, Singleton, Observer, and Decorator patterns.

10.4.7.2 Hypothesis Testing Initially, we begin the analysis by determining if using the parametric ANOVA approach is appropriate. We determined this by validating the fundamental assumptions of ANOVA. As noted above, the two fundamental assumptions we are concerned with are the normality and homogeneity of variances assumptions.

Normality Assumption To evaluate this assumption, we plotted the ANOVA model, as depicted in Figure 10.18. The pertinent plot here is the “Normal Q-Q” Plot in the upper right quadrant. Here we see deviations from Normal in the tails of the data, which is a strong indicator of a violation of the normality assumption. This evidence is further confirmed using the Anderson-Darling normality test. The results of this test ($A = 534.18$, $p < 2.2\text{e}{-16}$) provides strong evidence to reject the null hypothesis and further confirming the violation of the normality assumption.

Homogeneity of Variances Assumption This assumption is evaluated using a similar process as the Normality assumption. We again look to Figure 10.18, focusing on the “Residual vs. Fitted” plot in the upper-left quadrant. This plot indicates that there is

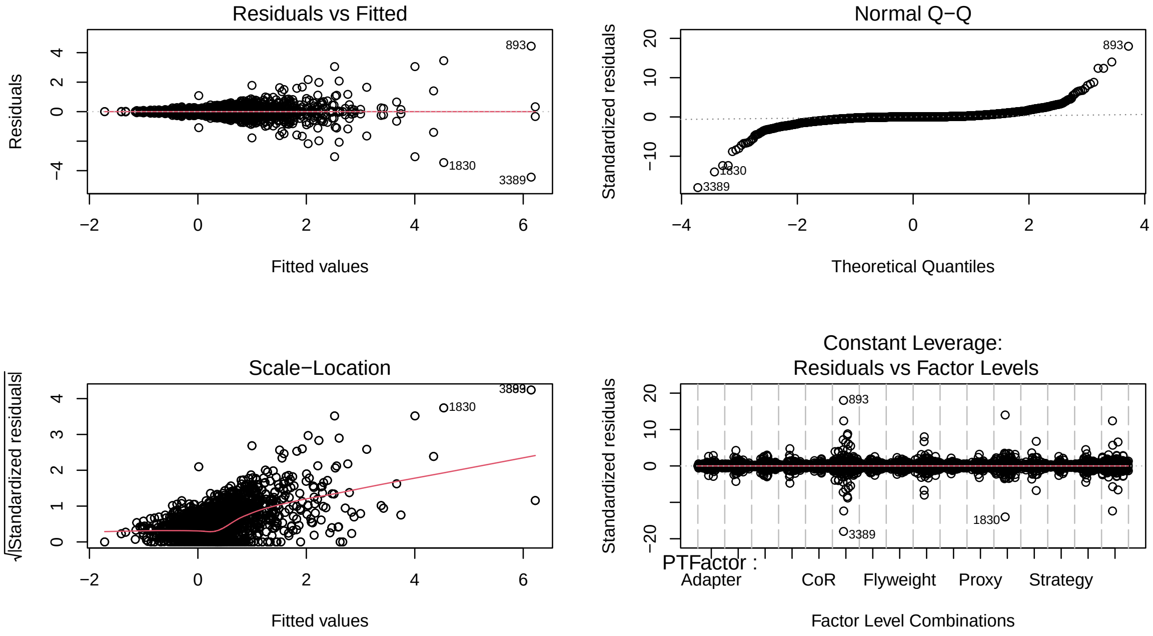


Figure 10.53: TD Principal diagnostic plots.

a violation of the assumption. To analytically confirm this, we executed Levene's Test for Homogeneity of Variance. The results ($F(2495, 2496) = 4.254\text{e}{+}26$, $p < 2.2\text{e}{-}16$) of this test provide strong evidence to reject the null hypothesis that the variances are the same. These results further confirming this assumption has been violated.

Permutation F-Test Analysis The assumption validation steps result in the conclusion that either we transform the data or use a permutation F-test approach. After several attempts to transform the data, we opted to move forward with the permutation F-test approach. The overall results ($F(2495, 2496) = 6.226$, $p < 2.2\text{e}{-}16$) of this test indicate strong evidence to reject the null hypothesis that there is no difference in the mean change in TD Principal.

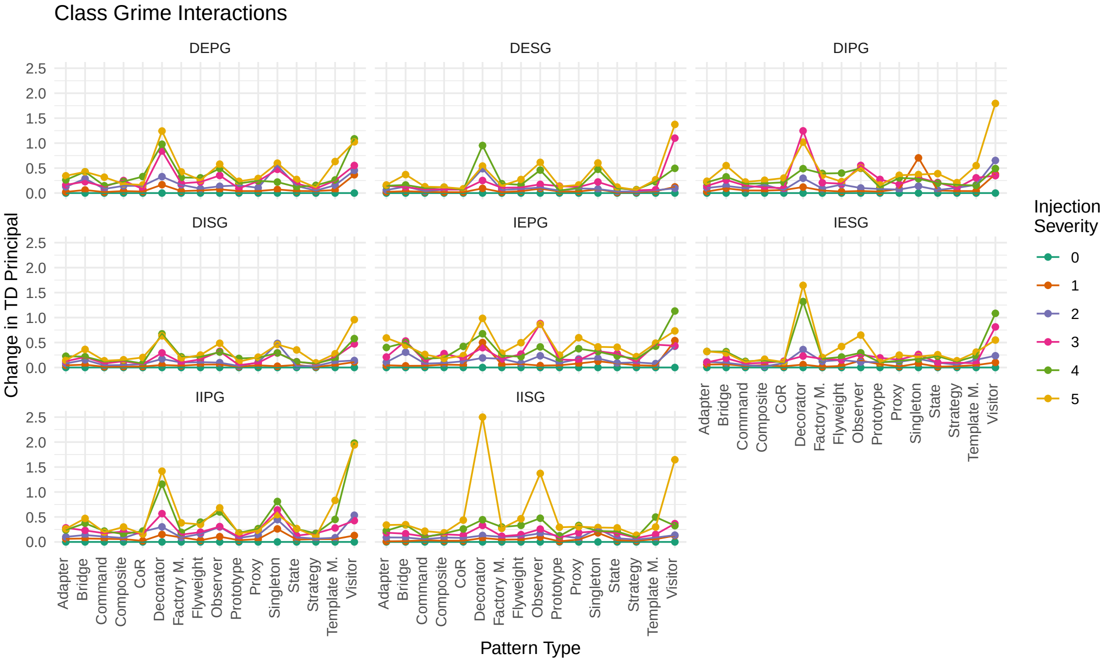


Figure 10.54: TD Principal interaction plots for class grime injection.

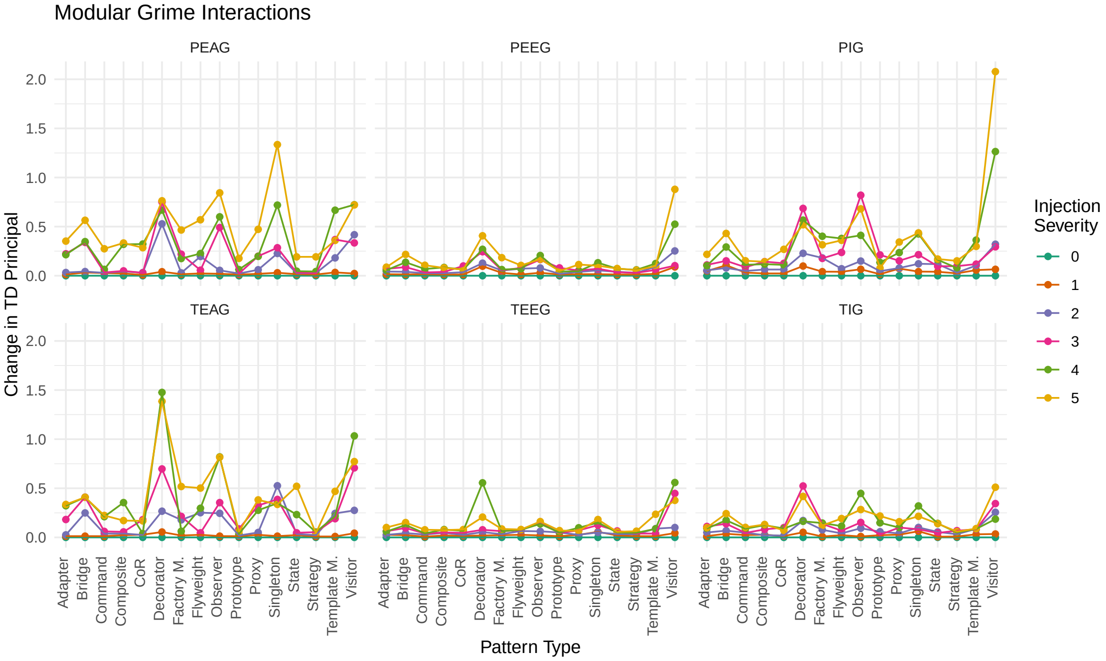


Figure 10.55: TD Principal interaction plots for modular grime injection.

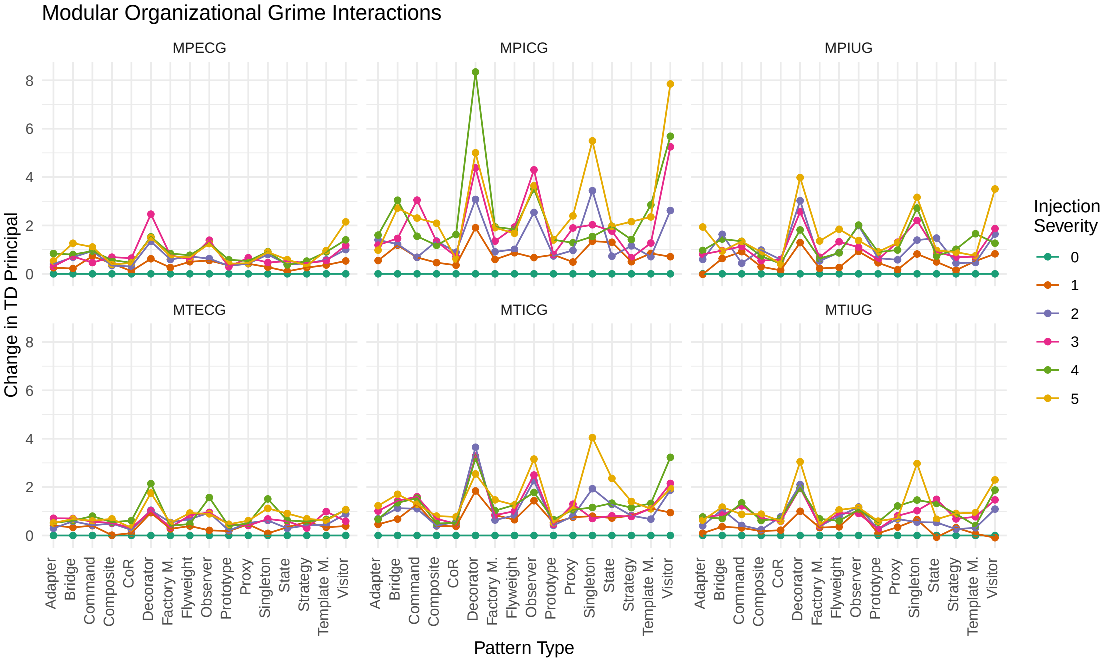


Figure 10.56: TD Principal interaction plots for modular organizational grime injection.

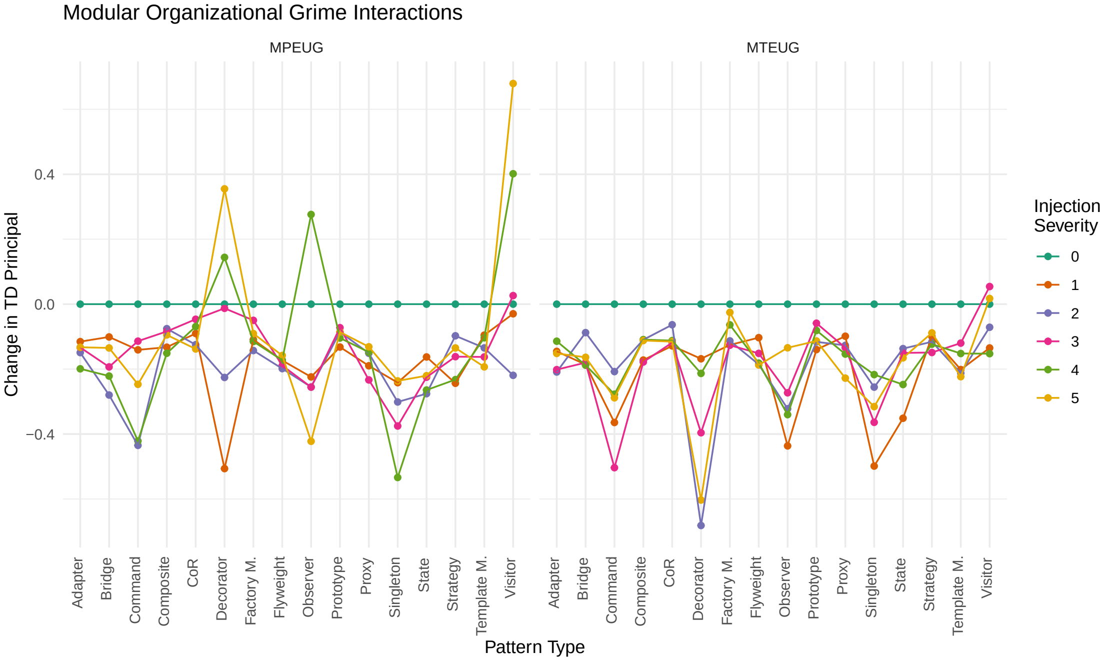


Figure 10.57: TD Principal interaction plots for MTEUG.

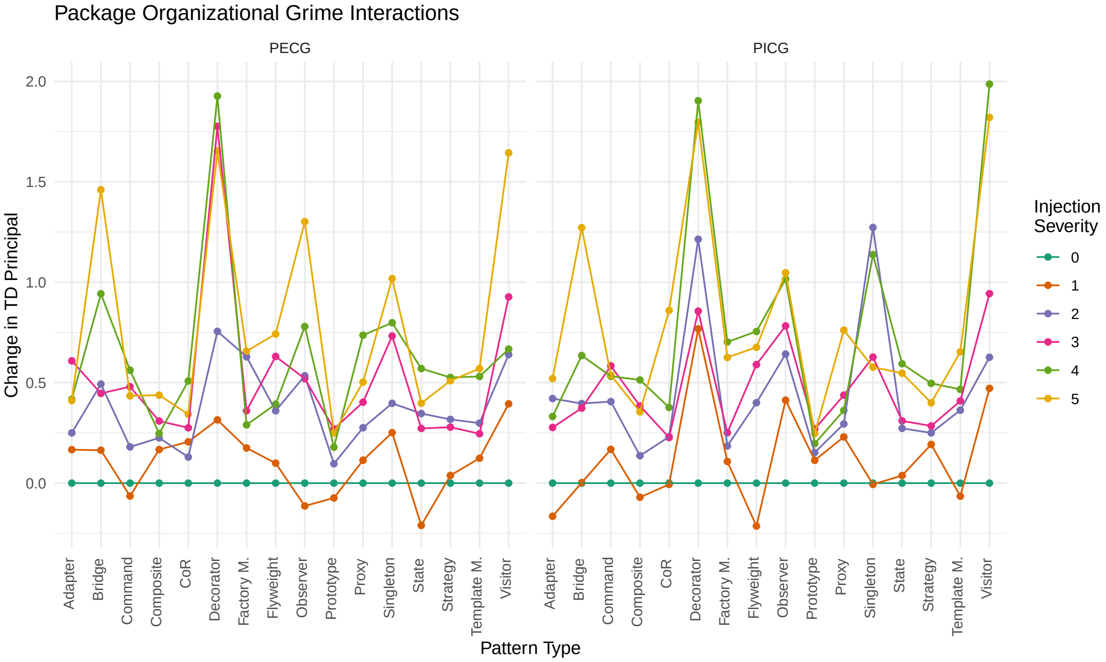


Figure 10.58: TD Principal interaction plots for PECG and PICG.

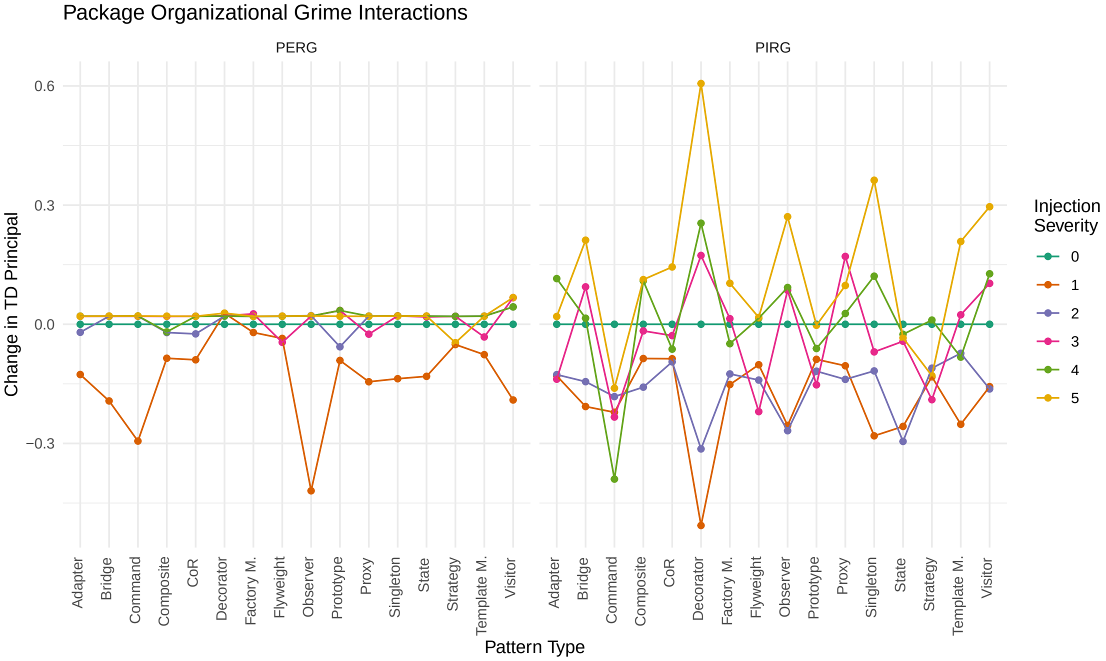


Figure 10.59: TD Principal interaction plots for PERG and PIRG.

Interaction Effects With the knowledge that a difference in the mean change in TD Principal exists between two or more treatment combinations, we continue by determining if significant interactions must be considered. In this case, there is strong evidence ($p <$ 2.2e$-$16) to reject $H_{2,0}$ that there is no difference in the mean change in TD Principal for each level of the three-way interaction effect. With this in mind, we consider a graphical analysis of these interactions. We subdivided by grime class: Class Grime, Modular Grime, and Organizational Grime to plot these interactions. Each grime category plot contains a matrix of subplots (one per grime type in the category). For each plot, the y-axis is the change in TD Principal, the x-axis is the design pattern type, and the points plotted are the values for each injection severity. We will begin with the plots for Class Grime.

We begin with the interaction plots for Class Grime as depicted in Figure 10.54. This figure shows that although for each subtype of Class Grime, all pattern types affect a positive change in TD Principal as the Injection Severity increases, it appears that some pattern types have a greater impact than others. The specific patterns which seem to have spikes in the Change in TD Principal across all grime types appear to be Bridge, Decorator, Observer, Singleton, and Visitor. A final note for Class Grime is that the level of change in TD Principal is minimal, ranging from 0.0 to approximately 0.0155 man-months. We next look at the interactions associated with Modular Grime subtypes.

Modular Grime subtype interactions are depicted in Figure 10.55. This figure shows similar information as was seen in the Class Grime interactions. Again we note that for each grime type, across each pattern type, the are apparent positive effects on the change in TD Principal for Injection Severity levels 1 - 5. We also note that there are similar spikes for each type of grime for the Bridge, Decorator, Observer, Singleton, and Visitor patterns. Additionally, the values of the level of Change in TD Principal associated with Modular Grime types tend to be very small and ranges from 0.0 to approximately 0.01 man-months. We next discuss the interactions associated with Modular Organizational Grime subtypes.

Figures 10.56 and 10.57 depict Modular Organizational Grime subtype interactions. Figure 10.56 can be further decomposed into three groups. The first group shows the largest effects on TD Principal from the MPICG, MPIUG, and MTICG types. The first group ranges from 0.0 to approximately 0.07 man-months. The second group, MPECG, MTECG, and MTIUG, show moderate changes in TD Principal. The range of the change in TD Principal for this group is between 0.0 and approximately 0.02 man-months. The third group composed of only MTEUG is the most interesting in that it shows slight negative changes in TD Principal. Again, the key spikes for each type of grime occur mainly in the Decorator, Observer, Singleton, and Visitor. The range for this group is quite small and cannot necessarily be identified from Figure 10.56. Instead, we have plotted just the MTEUG interactions as shown in Figure 10.57. In this figure, the negative spikes become visible with Command, Decorator, Observer, Singleton, and Visitor showing the most distinct spikes, including a positive spike for Visitor at Injection Severity level 4. We next discuss the interactions associated with the Package Organizational Grime subtypes.

Finally, Package Organization Grime subtype interactions are depicted in Figures 10.58 and 10.59. Figure 10.58 shows similar positive effects on the Change in TD Principal for PECG and PICG subtypes. It is also noteworthy that for an Injection Severity level 1, several Pattern Types indicate a negative effect on TD Principal for both PECG and PICG. On the other hand, Figure 10.59 shows that PERG and PIRG effects grow from negative to positive as Injection Severity increases across pattern types. The most interesting set of interactions is those of the PIRG subtype, which indicates that lower values (1 - 3) of Injection Severity have a negative impact on TD Principal. Change in TD Principle borders near or above zero as the Injection Severity levels increase beyond 3 for each pattern type. Additionally, we see similar spikes across each subtype depicted at the pattern types Bridge, Decorator, Observer, Singleton, and Visitor.

Table 10.9: Summary of TD Interest data.

Characteristic	**Min**	**Median**	**Mean**	**Max**	**SD**
ΔTD Interest	-0.0072254	0.0006794	0.0025232	0.0962732	0.005841174

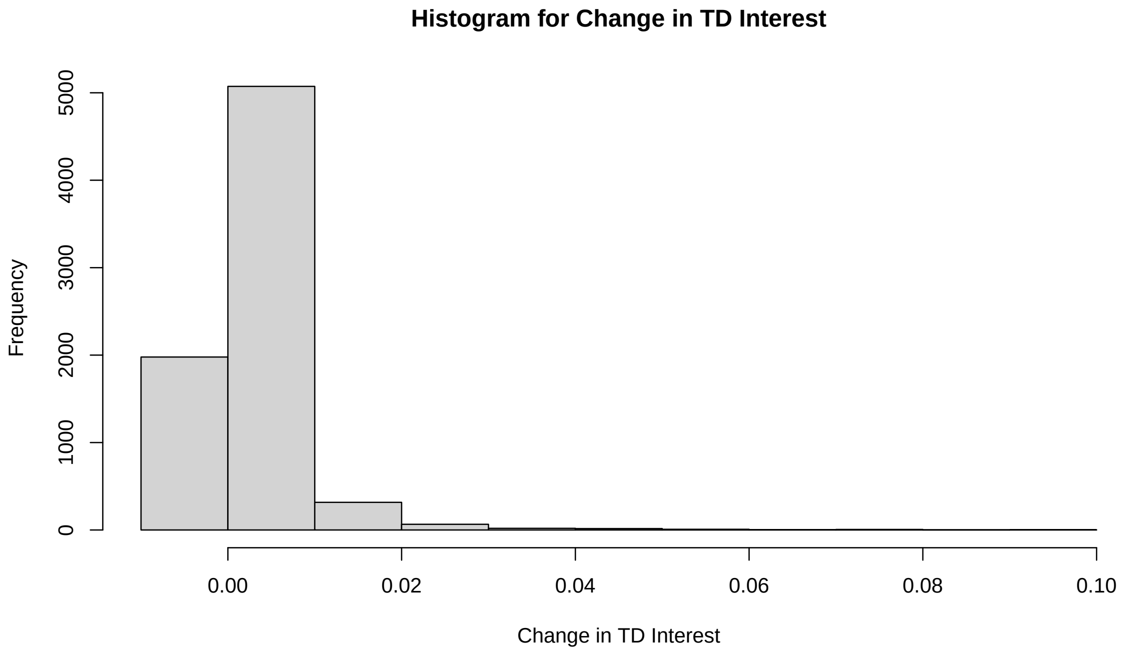


Figure 10.60: Histogram of the change in TD Interest.

10.4.8 Technical Debt Interest

This subsection describes the results of the Technical Debt Interest analysis. We subdivided the analysis into a subsection describing the data and descriptive statistics, and a subsection describing hypothesis testing.

10.4.8.1 Descriptive Statistics This section presents the results of the Technical Debt Interest experiment using descriptive statistics and plots. First, we show the summary of the Change in Technical Debt Interest (the dependent variable) in Table 10.9. The table

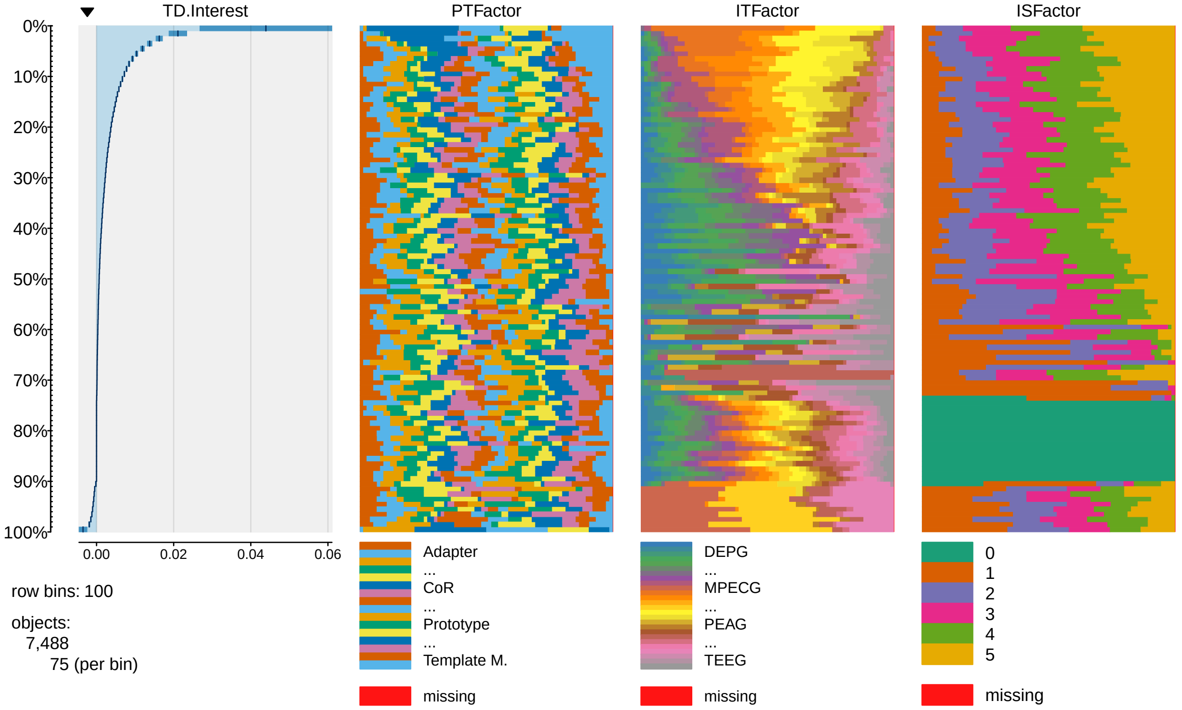


Figure 10.61: Table plot of TD Interest data.

shows the basic statistics across the 7,488 observations. This table suggests that across all observations, the change in Technical Debt Interest ranges between -0.0072254 and 0.0962732 man-months, and the mean change in Technical Debt Interest is 0.0025232 man-months. However, given the distribution of the values being heavily skewed to the left as depicted by the histogram in Figure 10.60, the median value of 0.0006794 man-months provides a better measure of the centrality of the data. Combining all of this with the standard deviation of 0.005841174, we know the following about this data: i) the majority of the observations showed a change to Technical Debt Interest; ii) of those observations that showed any change in Technical Debt Interest, it can be either negative or positive and that the magnitude is greater in the positive direction; and iii) there were some observations which show significant changes in Technical Debt Interest both in the positive and negative directions. To better understand how this data is distributed, in the context of the independent variables, we

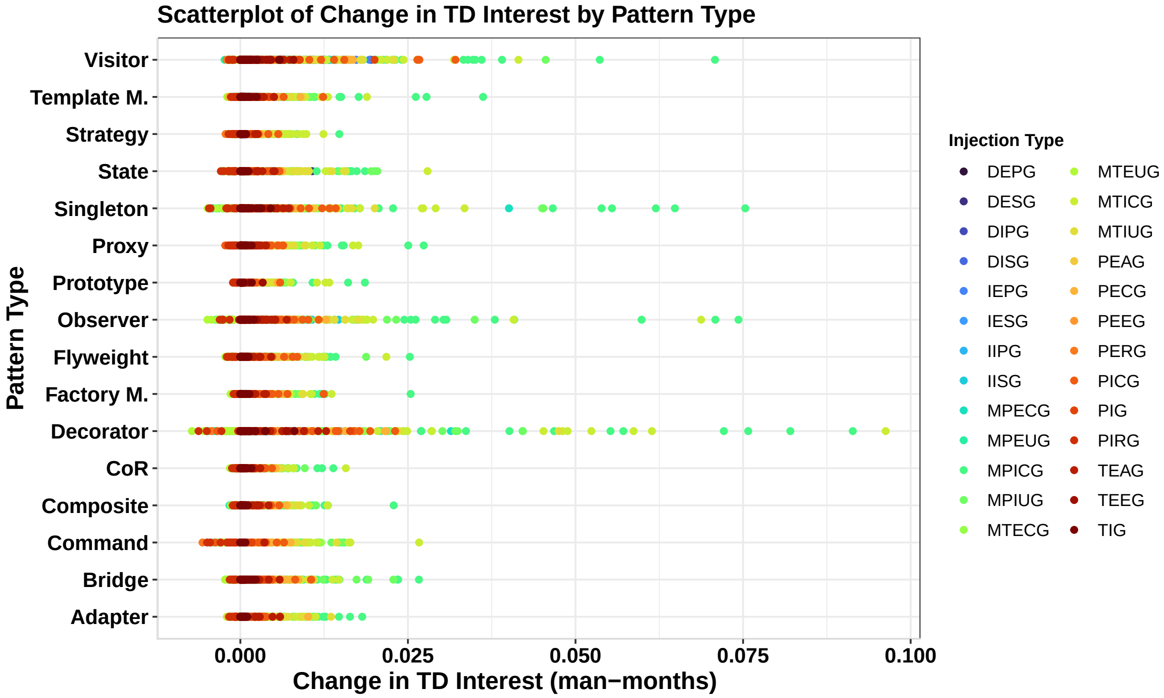


Figure 10.62: Scatterplot of the Change in TD Interest and Pattern Type.

constructed two plots: the first is a table plot (see Figure 10.61), and the second is a scatterplot (see Figure 10.62).

Figure 10.61 depicts a table plot of the dependent and each of the independent variables. Each column of this plot represents a single variable, while each plot row represents a sample of the data. The first column presents a histogram of the Change in Technical Debt Interest (TD.Interest) separated into 100 bins each containing 75 observations. The remaining columns show the values of Pattern Type (PTFactor), Injection Type (ITFactor), and Injection Severity (ISFactor) for the 75 values for each row of the Change in Technical Debt Interest. This data view allows us to see the distribution of the data and any interesting patterns that may exist across the columns.

In this plot, we initially see that approximately 70% of the change in Technical Debt Interest is positive, 10% is negative, and the remaining is zero. Between the 0% and approximately 70% marks the change in Technical Debt Interest is positive and the largest magnitude changes appear to be related to only a subset of the injection types. Additionally, the smallest magnitude positive changes appear to be due in large part to Injection Severity level 1 injections. Beyond these apparent relationships, there does not appear to be any other patterns related to the positive changes in Technical Debt Principle. The last approximately 10% of the data indicates a negative change in Technical Debt Interest and appears to be related to a different subset of Injection Types, and again there two groups apparently different in the magnitude of the changes affected. Again, there does not appear to be any relation to either the Pattern Type or the Injection Severity. The remaining approximately 20% of the data has a value of zero. This data is the only data where the Injection Severity level of 0 occurs. Furthermore, it appears that a change of zero cuts across all other Injection Severity levels, all Pattern Types, and all Injection Types as well.

Figure 10.62 shows the scatterplot of the Change in TD Interest by Pattern Type, with each point colored according to the Injection Type. This plot shows several key things. First, both large positive and small negative changes occur across all Pattern Types and all Injection Types. Additionally, we can see that the largest magnitude of change is due to the injection of primarily Modular Organizational Grime, with the largest changes due to MTICG and MPICG. Additionally, the largest spikes in TD Interest occur for the Visitor, Singleton, Observer, and Decorator patterns.

10.4.8.2 Hypothesis Testing Initially, we begin the analysis by determining if using the parametric ANOVA approach is appropriate by validating its fundamental assumptions. As noted above in Section 10.2.4, the two fundamental assumptions we are concerned with are the normality and homogeneity of variances assumptions.

Normality Assumption To evaluate this assumption, we plotted the ANOVA model, as depicted in Figure 10.18. The pertinent plot here is the “Normal Q-Q” Plot in the upper

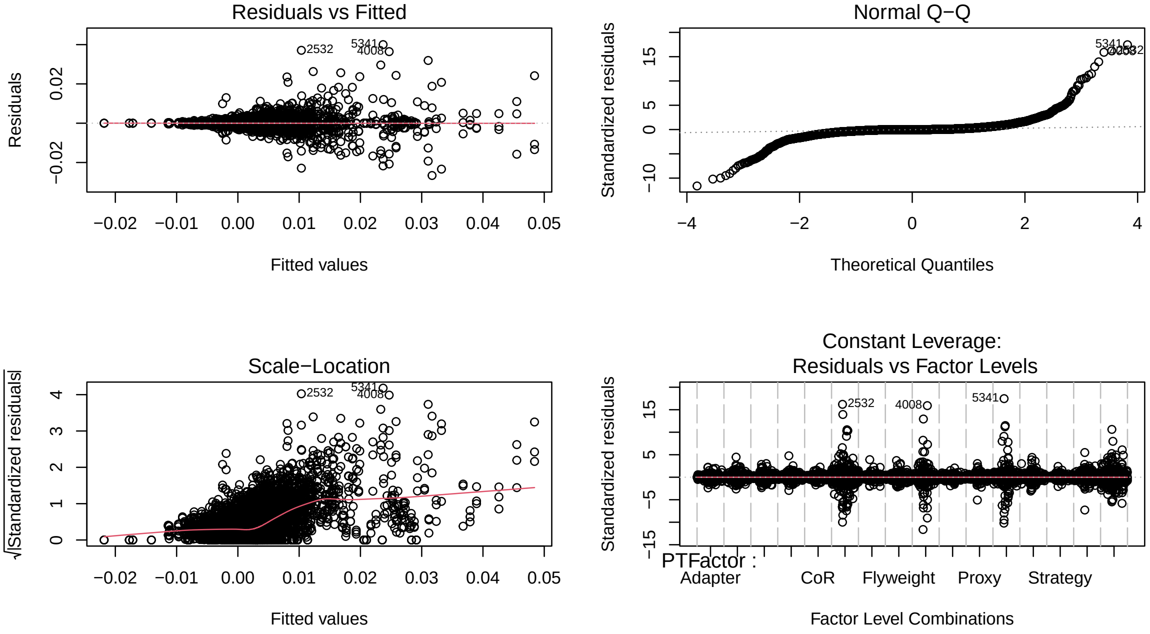


Figure 10.63: TD Interest diagnostic plots.

right quadrant. Here we see deviations from Normal in the tails of the data, which is a strong indicator of a violation of the normality assumption. This evidence is further confirmed using the Anderson-Darling normality test. The results of this test ($A = 964.57$, $p < 2.2\text{e}{-16}$) provides strong evidence to reject the null hypothesis and further confirming the violation of the normality assumption.

Homogeneity of Variances Assumption This assumption is evaluated using a similar process as the Normality assumption. We again look to Figure 10.18, focusing on the "Residual vs. Fitted" plot in the upper-left quadrant. This plot indicates that there is a violation of the assumption. To analytically confirm this, we executed Levene's Test for Homogeneity of Variance. The results ($F(2495, 4992) = 1.369$, $p < 2.2\text{e}{-16}$) of this test provides strong evidence to reject the null hypothesis that the variances are the same. These results further confirming this assumption has been violated.

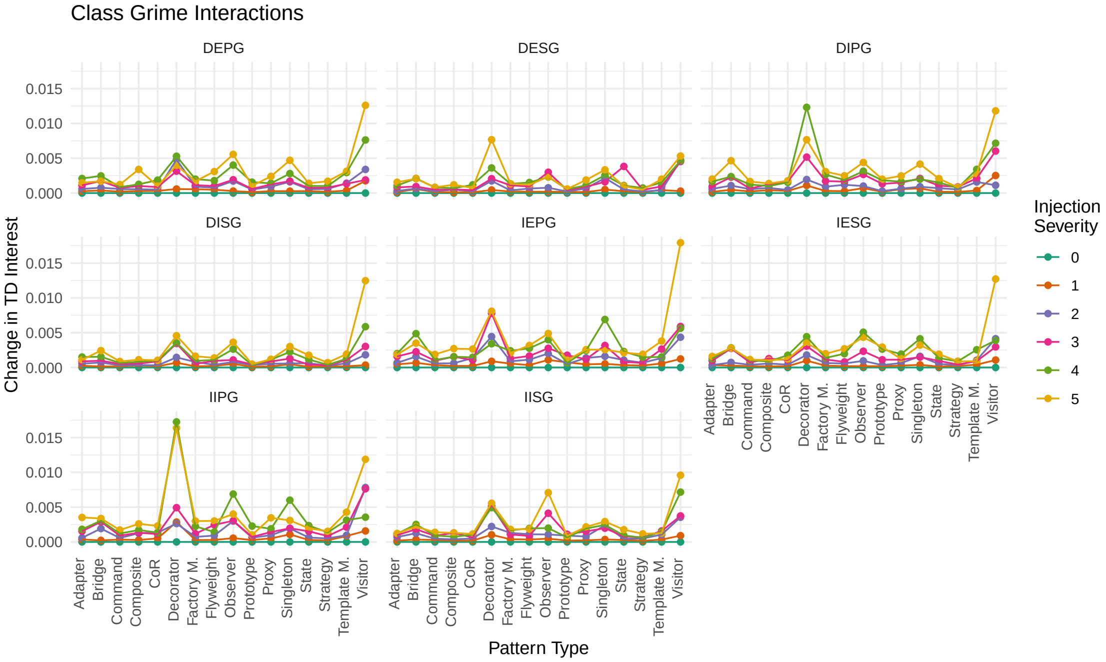


Figure 10.64: TD Interest interaction plots for class grime injection.

Permutation F-Test Analysis The assumption validation steps result in the conclusion that either we need to transform the data or use a permutation F-test approach. After several attempts to adjust for the violations, we opted to conduct a permutation F-test. The overall results of this test ($F(2495, 4992) = 11.01$, $p < 2.2\text{e}{-16}$) indicates strong evidence to reject the null hypothesis that there is no difference in the mean change in TD Interest.

Interaction Effects Knowing that a difference in the mean change in TD Interest exists between two or more treatment combinations, we continue considering any significant interactions. In this case, there is strong evidence ($p < 2.2\text{e}{-16}$) to reject $H_{2,0}$ that there is no difference in the mean change in TD Interest for each level of the three-way interaction effect. With this in mind, we will consider a graphical analysis of these interactions. To plot these interactions, we subdivided them into grime categories: Class Grime, Modular Grime, and Organizational Grime. Each grime category plot contains a matrix of subplots (one per

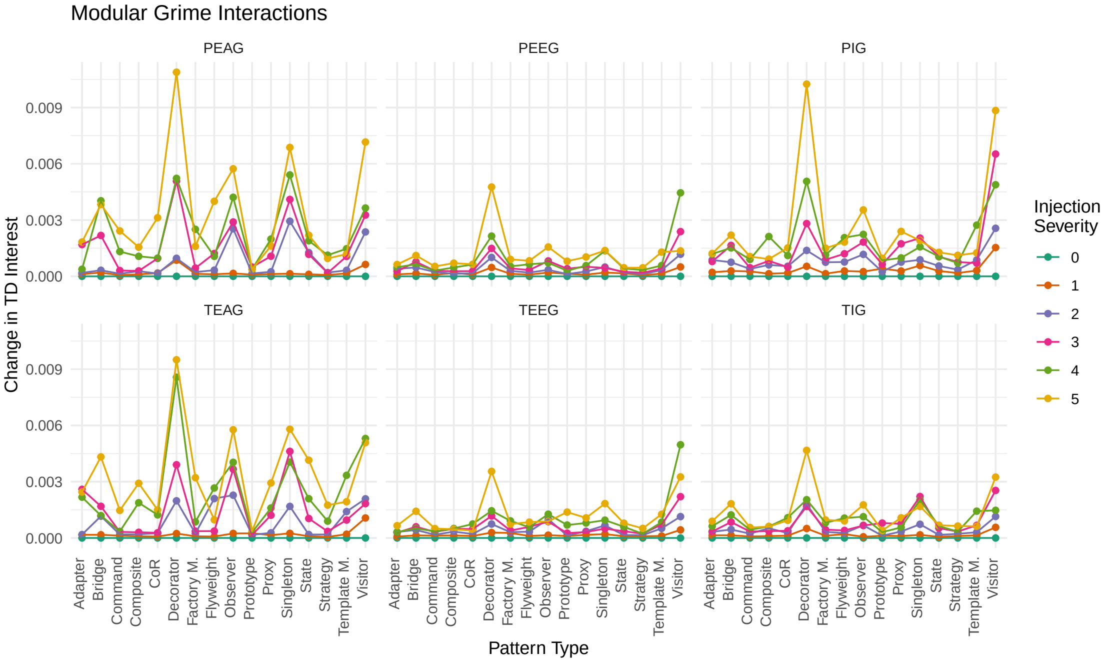


Figure 10.65: TD Interest interaction plots for modular grime injection.

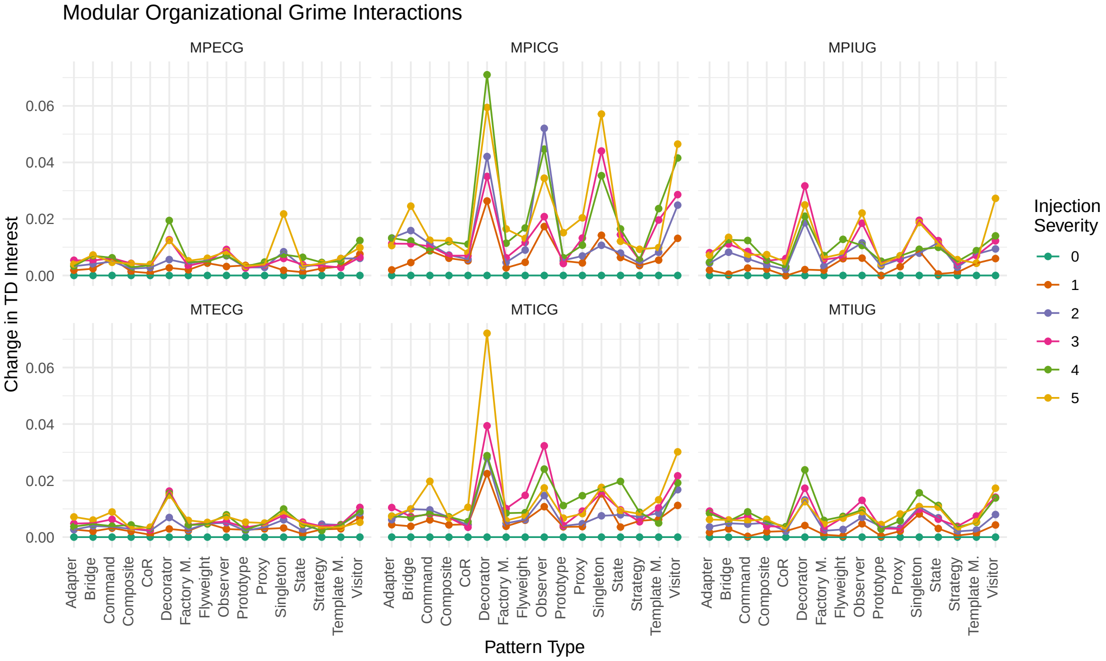


Figure 10.66: TD Interest interaction plots for modular organizational grime injection.

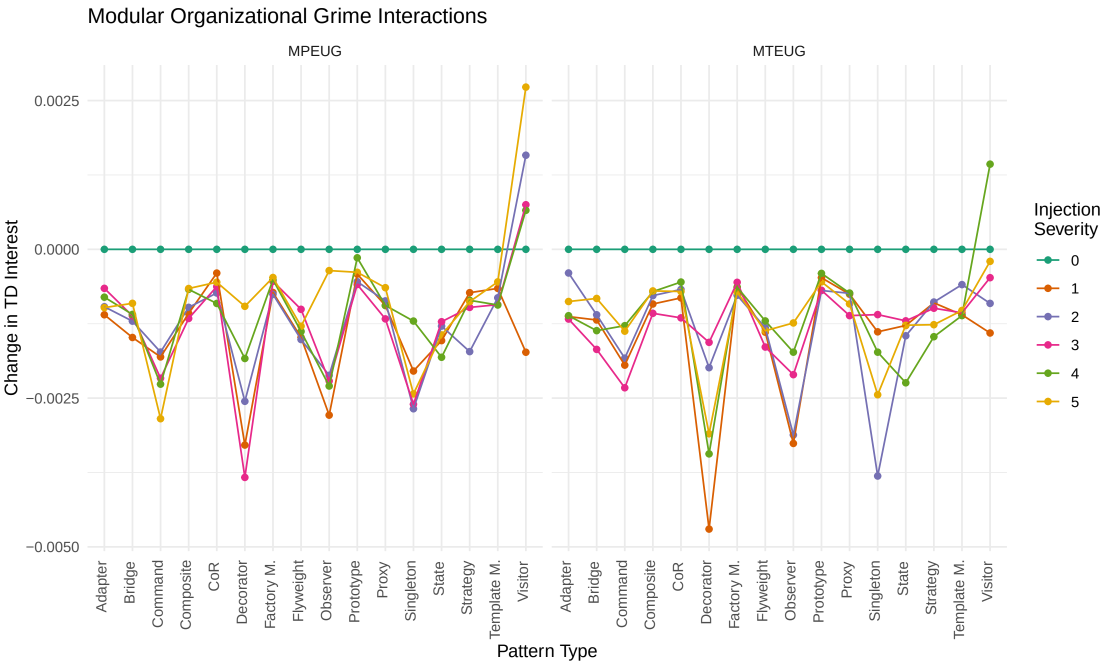


Figure 10.67: TD Interest interaction plot for MTEUG.

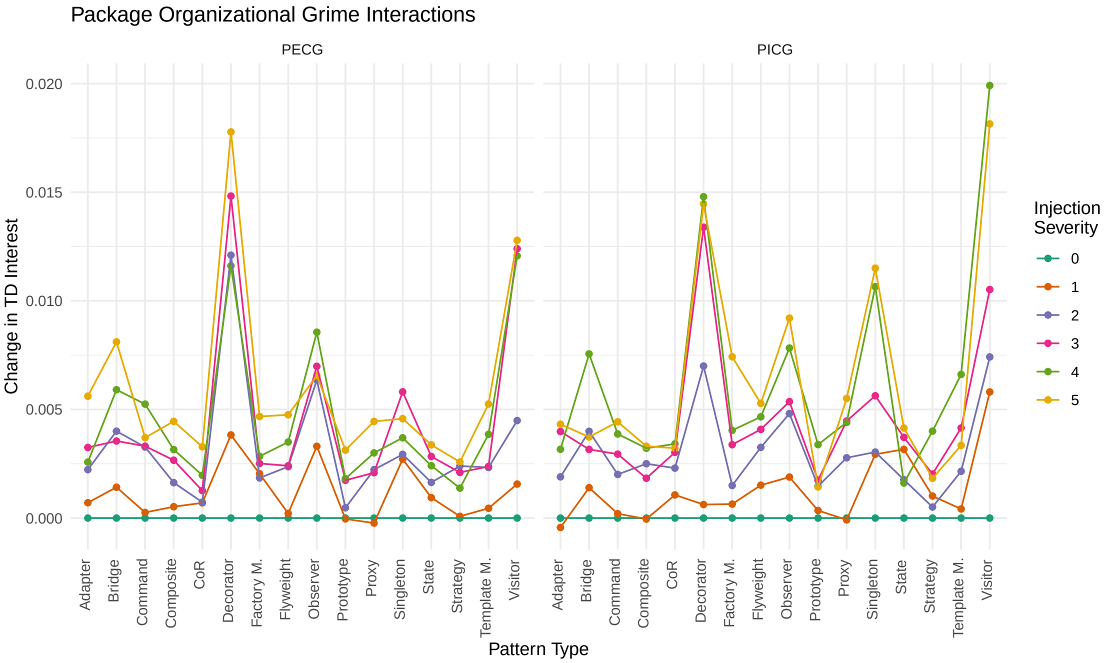


Figure 10.68: TD Interest interaction plots for PECG and PICG.

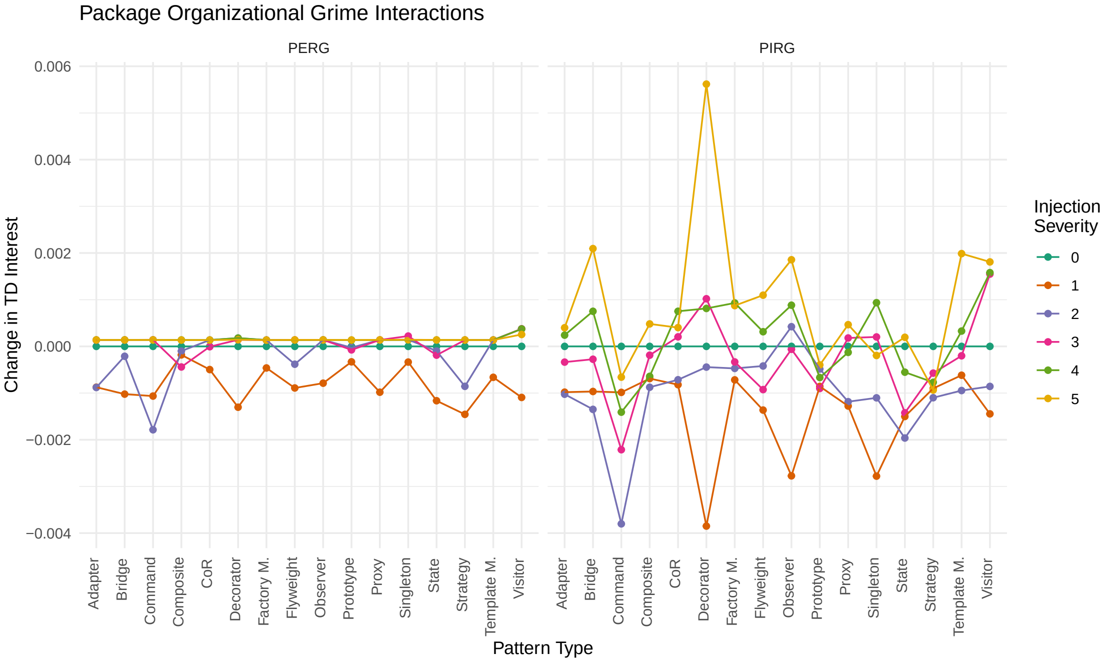


Figure 10.69: TD Interest interaction plots for PERG and PIRG.

grime type in the category). In each interaction plot, the y-axis is the change in TD Interest, the x-axis is the design pattern type, and the points plotted are the values for each injection severity. We will begin with the plots for Class Grime.

We begin with the interaction plots for Class Grime as depicted in Figure 10.64. This figure shows that although for each subtype of Class Grime, all pattern types affect a positive change in TD Interest as the Injection Severity increases, it appears that some pattern types have a greater impact than others. The specific patterns which seem to have spikes in the Change in TD Interest across all grime types appear to be Bridge, Decorator, Observer, Singleton, and Visitor. A final note for Class Grime is that the level of change in TD Interest is minimal, ranging from 0.0 to approximately 0.0155 man-months. We next look at the interactions associated with Modular Grime subtypes.

Modular Grime subtype interactions are depicted in Figure 10.65. This figure shows similar information as was seen in the Class Grime interactions. Again we note that for each grime type, across each pattern type, the are apparent positive effects on the change in TD Interest for Injection Severity levels 1 - 5. We also note that there are similar spikes for each type of grime for the Bridge, Decorator, Observer, Singleton, and Visitor patterns. Additionally, the values of the level of change in TD Interest associated with Modular Grime types tend to be very small and ranges from 0.0 to approximately 0.01 man-months. We next discuss the interactions associated with Modular Organizational Grime subtypes.

Figures 10.66 and 10.67 depict Modular Organizational Grime subtype interactions. As shown, MPICG, MPIUG, and MTICG have the largest effects on TD Interest ranging from 0.0 to approximately 0.07 man-months. MPECG, MTECG, and MTIUG show a moderate effect on the Change in TD Interest ranging between 0.0 and approximately 0.02 man-months. We have plotted just the MTEUG interactions as shown in Figure 10.67. As shown, MTEUG has a slightly negative effect on the Change in TD Interest. Again, the key spikes/dips for each type of grime occur mainly in the Command, Decorator, Observer, Singleton, and Visitor showing the most distinct spikes/dips, including a positive spike for Visitor at Injection Severity level 4. We next discuss the interactions associated with the Package Organizational Grime subtypes.

Finally, Package Organization Grime subtype interactions are depicted in Figures 10.68 and 10.69. Figure 10.68 shows similar positive effects on the Change in TD Interest for PECG and PICG subtypes. It is also noteworthy that for an Injection Severity level 1, several Pattern Types indicate a negative effect on TD Interest for both PECG and PICG. On the other hand, Figure 10.69 shows that PERG and PIRG effects grow from negative to positive as Injection Severity increases across pattern types. The most interesting set of interactions is those of the PIRG subtype, which indicates that lower (values between 1 - 3) levels of Injection Severity negatively impact the Change in TD Interest. However, as the levels increase beyond 3 for each pattern type, the Change in TD Interest value is near zero. Additionally, we see similar spikes across each subtype depicted at the pattern types Bridge, Decorator, Observer, Singleton, and Visitor.

10.5 Interpretation

In this section we interpret the results identified in Section 10.4. This section is further subdivided into three subsections. The first subsection (Section 10.5.1) evaluates the results from Section 10.4 from the perspective of each of the research questions. The second subsection (Section 10.5.2) describes the threats to validity for this study. Finally, the third subsection (Section 10.5.3) will describes how these results generalize given in the context of the findings and threats to validity.

10.5.1 Evaluation of Results and Implications

In this subsection we discuss the results as they pertain to the research questions identified in Section 10.2.1. We begin with research questions **RQ2.1** – **RQ2.3** followed by a summary of these results as they pertain to the more general question **RQ2**. Following this, we analyze the results in the context of research questions **RQ3.1** – **RQ3.3** followed by a summary of how these results pertain to the more general question **RQ3**.

10.5.1.1 RQ2.1 How does each type of Grime affect design pattern quality for each of the selected Maintainability sub-characteristics?

We found that across all grime types that grime has a negative effect on Analyzability. Additionally, we found that the injection of grime negatively impacts the Testability of pattern instances across all pattern types and grime types. This provides further support for the implications to testability first identified by Izurieta and Bieman [128, 133]. We note that with the exceptions of MPECG, MPICG, MPIUG, MTECG, MTICG, MTIUG, and PERG types, the effect grime has on Testability is dependent on Pattern Type. Furthermore, the Testability of instances of Bridge, Decorator, Observer, Singleton, and Visitor pattern types appears affected by grime more than instances of other pattern types.

Furthermore, grime affects Modifiability both negatively and positively but is dependent on Pattern Type. Specifically, instances of the Decorator, Prototype, and Visitor patterns will have their Modifiability positively impacted by all types of grime except PEAG, TEAG, MPICG, MTICG, and PIRG. Of these, MPICG and MTICG will negatively impact these same pattern types, but the PEAG, TEAG, and PIRG negatively impact the Modifiability of all pattern types. Modifiability, in the SIG Model, is affected by three properties: Unit Complexity, Duplication, and Module Coupling. Of these, only Module Coupling is affected by the injection process. The injection process focuses on injecting structural components rather than behavioral components, thus avoids creating excess complexity or duplication. However, it does add in new couplings between classes, which can affect *afferent coupling* which is the underlying metric used to measure Module Coupling. Furthermore, when Class Grime is injected it increases the intra-class connections between fields and methods of a class, without increasing the number of inter-class couplings. This then increases the overall volume of the system. The rating value increases because the overall volume of the system increases without increasing the module coupling, thus decreasing the risk. As for the other forms of grime for which Modifiability increased, a similar reasoning may be given. Specifically, for the forms of Organizational and Modular Grime (excluding those which decreased), the injection process most likely added new classes and/or packages into the system, increasing the volume at a rate greater than by which the Module Coupling value was increased (if increased at all) when new afferent inter-class couplings were formed.

Additionally, we have found that Modularity is affected by all forms of grime. This impact on Modularity may be positive or negative, depending on the type of grime, pattern type, and injection severity.

10.5.1.2 RQ2.2 What level of injection severity affects a change in design pattern quality for each of the Maintainability sub-characteristics? Across all grime types and pattern types, we note that for all Injection Severity levels, wherein grime was injected, there was a negative change in Analyzability. The only exception to this was for PECG and the Prototype Pattern Type, for which an Injection Severity level of 1 resulted in a mean change in Analyzability of 0. Similarly, across all grime types and pattern types, we note that there was a negative change in Testability for all Injection Severity levels in which Grime was injected. On the other hand, Modifiability is not affected for all patterns and types of grime at all levels of Injection Severity. Rather, it may be positively or negatively affected for specific pattern types (see RQ2.3 below) and when the Injection Severity level is 3 or more. The effects of Grime on Modularity, in general, occur at all levels of injection severity. When considering Class Grime, the positive effects often depend upon the level of injection severity. When considering the adverse effects from grime on Modularity, these occur across all patterns and, in general, all levels of injection severity.

10.5.1.3 RQ2.3 What is the difference between the effects of the grime types and their subtypes on maintainability sub-characteristics? The effect of Grime on Analyzability differs in magnitude depending on the grime category. The most significant impact on Analyzability stems from the Organizational Grime category. In comparison, the effects of Class and Modular Grime are quite small. However, when evaluating these forms of grime, it becomes apparent that we must consider pattern type, as the Analyzability of Bridge, Decorator, Observer, Singleton, and Visitor instances tends to be affected more than others.

The effect of Grime on Testability differs in magnitude depending on which type of grime is injected. Of the three grime categories (Class, Modular, and Organizational), the most significant impact on Testability stems from the Organizational Grime category. In comparison, the effects of class and modular Grime are quite small. However, when evaluating these forms of grime, it becomes apparent that Pattern Type must be considered as the Testability of Bridge, Decorator, Observer, Singleton, and Visitor instances tends to be affected more than others.

The critical differences between grime types and their subtypes when considering the effect on Modifiability do not lie between the categories of grime but rather within. That said, there is one stark difference between the negatively affecting grime Types PEAG, TEAG, MPICG, and MTICG, and all others in that they have the most considerable magnitude of change. Additionally, PEAG and TEAG affect all pattern types rather than a select few. The remaining grime types are relatively indistinguishable in that they affect Modifiability positively with nearly the same level of impact and for only a few pattern types noted above.

The key differences between how grime types and their subtypes affect Modularity are sharply divided between Organizational Grime types and those found in Class and Modular Grime (excluding PEAG and TEAG). Class and Modular Grime (excluding PEAG and TEAG) affect Modularity in a small but positive way. On the other hand, the PEAG and TEAG forms of Modular Grime, along with the Modular Organizational Grime subtypes, have significant negative effects on Modularity. Of these, both MTICG and MPICG are the most pronounced. Conversely, MTEUG and the Package Organizational Grime subtypes have a significantly positive effect on Modularity.

10.5.1.4 RQ2 Summary In summary, design pattern grime can be a concern for the overall Maintainability of a design pattern instance but may also be beneficial depending on a developer's/teams specific concerns. Except for Reusability, the level of concern developers or teams should depend on the type of pattern affected, the type of grime it is afflicted with, and the amount of grime present. We know that, in general, as severity increases, grime's effects (either positive or negative) will increase in magnitude.

However, we also note that neither Grime Type or Injection Severity had any discernible effect on the Reusability of a pattern instance. Reusability, in the SIG Maintainability Model is affected by Unit Size (as measured in SLOC) and Unit Interfacing (as measured by Number of Parameters) properties. These properties, are not directly affected by the injection process. The reason for this is that the injection process, as currently implemented, focuses on the injection of structural aspects rather than on behavioral aspects. Thus, when methods (or Units in the SIG terminology) are injected they tend to be mainly stub methods, and thus provide no change to Unit Size (as they have no size). Unlike Unit Size, Unit Interfacing could be affected by the Injection Process, if parameter injection is selected for the injection of temporary couplings between classes. However, as noted injected methods tend to be stubs and will follow the basic definition from a pattern specification. In this case, the initial number of method parameters will be zero or very small, thus adding in an additional parameter will not be enough to change the rating for Unit Interfacing. For these reasons, Reusability does not appear to be affected by injected structural forms of grime.

10.5.1.5 RQ3.1 How does each type of grime affect design pattern technical debt principal and interest? The impact of all grime categories on TD Principle and TD Interest are very similar. All grime subtypes excluding MTEUG, PECG, and PICG appear to affect a positive change in TD Principal across all pattern types, which increases as the severity of injected grime increases. These results are in line with the work from Dale and Izurieta [62] which showed similar results for Modular Grime. The remaining three subtypes are much more interesting, specifically MTEUG. MTEUG negatively impacts TD Principal and Interest, suggesting that MTEUG reduces technical debt across all pattern types and for all injection severity levels. This finding will require further investigation.

10.5.1.6 RQ3.2 What level of grime severity affects a change in design pattern technical debt principal and interest? As noted above in RQ3.1, all severity levels affect technical debt, and except for the MTEUG subtype, they affect a positive change in TD Principal and

Interest. Additionally, similar to the results of Dale and Izurieta [62] for all grime subtypes, as the injection severity increases, the change in TD Principal and Interest also increases in magnitude (either positively or negatively).

10.5.1.7 RQ3.3 What is the difference between the effects of the grime types and their subtypes on technical debt principal and interest? The key difference in the effects of Grime on TD Principal and Interest between subtypes can be separated into two groups. The first group contains the subtypes for Class and Modular Grime. There is relatively little difference between the effects on either TD Principal or Interest across grime types in this group. Instead, the primary difference is in the effects of pattern type and injection severity. The key pattern types of concern are the Decorator, Observer, Singleton, and Visitor patterns. The effect on TD Principal and Interest is markedly higher than for other patterns affected by the same type of Grime. The second group is the Organizational Grime subtypes, in which they follow a similar trend as that of the first group (except for MTEUG), but the magnitude of the effects tends to be much higher. Lastly, MTEUG has minor effects on TD Principal and Interest, but these effects are negative rather than positive. Additionally, MTEUG also dips on Command rather than the Bridge pattern and the Template Method pattern.

10.5.1.8 RQ3 Summary In summary, design pattern grime is something of serious concern for technical debt management. Grime, in general, will increase a pattern instance's TD Principal and Interest. Developers should keep this in mind, especially when working with pattern instances of type Bridge, Decorator, Observer, Singleton, or Visitor. One caveat to this is that MTEUG does reduce both the TD Principal and Interest of a pattern instance regardless of the pattern type. Thus, if this form of Grime manifests within a pattern instance already afflicted by other forms of Grime, it may mask the real issues at hand. These issues are of specific concern as it has been shown in prior work that Grime will continue to build up unless remediated [77, 133, 234].

10.5.2 Limitations of the Study

This section describes the limitations and threats to the validity of this study. Specifically, we focus on threats to the conclusion, internal, construct, content, and external validity, per the frameworks proposed by Campbell and Cook [44], Campbell and Stanley [45], and Wohlin et al. [276].

10.5.2.1 Conclusion Validity

Conclusion validity is concerned with establishing statistical significance between the independent and dependent variables. There is a threat to conclusion validity stemming from the fact that we utilized a control level that always resulted in zero, as expected, ensuring that we could not meet the Homogeneity of Variance assumption for the ANOVA model. Additionally, using classification and regression tree analyses can further help mitigate internal threats beyond a permutation F-test.

10.5.2.2 Internal Validity

Internal validity is concerned with the relationship between the treatments and the outcomes and whether this relationship is causal or due to other factors. Although the experiments were fully controlled due to the number of replications conducted to ensure that the power of the experiments was at the required level, we simultaneously ensured that we would see some change. Additionally, when considering the effects of the control level injecting many zeroes into the results, we can mitigate this further by exploring other techniques.

10.5.2.3 Construct Validity

Construct validity is concerned with the meaningfulness of measurements and the quality choices made about independent and dependent variables such that these variables are representative of the underlying theory. In this experiment, we used a Java$^{\text{TM}}$ specific calibration of the ISO/IEC 25010 quality model maintainability sub-characteristics via our implementation of the SIG Maintainability measurement method. In order to measure the expected small changes in each maintainability subcharacteristic, we modified the original SIG approach to use a ratio scale based on a linear projection during the rating phase (c.f. 5.3.2.1). Although this approach does not assume an equal distance between each rating interval, it does assume that the relationship is linear between ratings. Thus, this creates difficulty in interpreting the values and meaning, which is a clear threat to construct validity.

Additionally, there is a threat in using Nugroho's approach for measuring TD Interest. Specifically, the conversion of the value for the $QualityLevel$ from an interval value to a ratio value as QF through exponentiation violates measurement theory. Additionally, this is affected by the same issue we noted with the linear projection for the SIG values. Finally, we used the same approach for measuring $QualityLevel$, simply the Maintainability rating, causing another violation of construct validity.

10.5.2.4 Content Validity Content validity is concerned with how well the selected measures cover the content domain. In this experiment, we measure software maintainability using an implementation of the SIG Maintainability Model. We calibrated this model (c.f. 5.3.2.3) for the Java™ language using open source projects from the Qualitas Corpus [255]. This model adequately covers the five maintainability sub-characteristics of concern. Furthermore, the IT, IS, and PT metrics are well defined and cover the content domain within the underlying tools' limitations. However, the PT metric only covers 16 of the 23 GoF design patterns, as this is the limit of the Pattern4 design pattern detection tool, which threatens content validity.

10.5.2.5 External Validity External validity is concerned with the ability to generalize the results of a study. We conducted the experiment using generated instances of Java™ design pattern implementations. However, we did not generate the studied instances within patterns extracted from existing software systems. Thus, we cannot directly generalize to either open source or industry software, which threatens external validity. Additionally, other forms of grime may not have been accounted for, including Behavioral Grime and other unknown forms. Both issues are threats to the external validity of the study. Furthermore, using only Java™ limits our ability to generalize to Java™ instances of design patterns, which is another threat.

10.5.3 Inferences

In each, each experimental unit is a design pattern instance whose size selected template, and internal components are subject to pseudo randomization within the specification of the pattern to be implemented. Additionally, each injection uses pseudo randomization to select elements into which grime is injected (constrained by the definition of the specific grime type). This is akin to random sampling from a population. Thus, one may infer the difference in the change in Analyzability, Testability, Modifiability, Modularity, TD Principal, and TD Interest was caused by the difference in Injection Type, Injection Severity, and Pattern Type. Because the subjects were generated randomly based on Java™ design pattern templates designed to mimic the structure of design patterns found in real software, there is a theoretical case that these results may extend to pattern instances found in professional-grade Java™ software. However, there is not a statistical case for such a generalization.

10.6 Conclusion and Future Work

In this study, we conducted seven experiments to understand the effects of the relationship of design pattern grime on software maintainability and technical debt. Each experiment was conducted as a full-factorial design using the software injection process to inject each of the 26 design pattern grime types at 6 severity levels into 16 generated pattern types. The results showed that indeed design pattern grime has significant effects on both software maintainability sub-characteristics as well as technical debt interest and principal. However, this effect is dependent upon the grime type, pattern type, and level of severity.

Our results have made significant strides towards addressing the problem identified in Section 1.1.1. Furthermore, these results have confirmed prior works from Izurieta and Bieman [128, 133] as well as Dale and Izurieta [62]. However, the underlying experimental conditions leave us unable to know if these results apply within existing software systems.

Future work will consist of conducting case studies to further study Grime in existing software systems and design pattern instances to explore these results further. Additionally, we need to return to these experiments and focus on the interesting issues that we have identified. Specifically, further explorations of the relationships between the MTIUG type and TD Principal and Interest.

CHAPTER ELEVEN

VERIFICATION STUDY

I am pretty sure there is a difference between "this has not been proven" and "this is false."

–Ron Jeffries

11.1 Introduction

Prior studies involving design pattern grime injection including the study in Chapter 10 and those found in the literature [61,101] are limited to the simulation of grime accumulation. Though such experiments are necessary to develop an understanding of the effects of grime, these results do not speak to the effects of grime in actual software systems.

As described in Chapter 8, we have developed an approach to resolve this limitation via, what we term, "Verification Studies." In this chapter we detail a Verification Study that serves two distinct functions. The first is to build upon the injection studies by validating that the results found in the previous experiments hold within real software systems. This verification is done by comparing the differences in our attributes of concern (maintainability and technical debt subcharacteristics). We extract a pair of design pattern instances from a single chain, wherein there is a difference in the grime accumulated from one version to the next. We then evaluate the change in the attributes of concern between the extracted versions of the pattern instance and compare this change to what is expected given the results for the same pattern type, grime type(s), and grime severity (of the change in grime) for the pair studied. The second function is to verify that the results of injected grime match the results of actual instances of that grime. As in the first comparison, we extract sequential pattern instances from a pattern chain where a difference in the accumulated grime is identified.

The earlier version becomes the base while the later version is noted as the normal path of evolution. We then copy the base version, and use the difference in grime between the extracted versions to guide injection of grime into the copy of the base version. We then calculate the difference between the normal path of evolution and the base, and between the injected evolution and the base. We then compare these values to determine how well the effects of the injection process correspond to the effects of the real changes in development. The goal of this study was stated in Chapter 1, and for the reader's convenience we restate it here:

RG4: Analyze design pattern instances for the purpose of comparing injected and observed instances of grime with respect to their ISO/IEC 25010 Maintainability subcharacteristics attributes and Technical Debt Principal and Interest from the perspective of researchers in the context of open source Java™ software projects.

Following the GQM process this goal then leads to our main question of interest and its corresponding rationale:

RQ4: Do observed and injected grime have a similar effect on the Maintainbility subcharacteristics and Technical Debt Principal and Interest?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on Maintainability and Technical Debt Principal and Interest as the natural process of grime accumulation.

With this question in mind, this chapter is organized as follows. Section 11.2 describes the design of this study. Section 11.3 describes the selection criteria for each case selected as a part of this study. Section 11.4, defines the data collection procedures for this study. Section 11.5 defines the analysis procedures for this study. Section 11.6 describes the results and their analysis for this study. Section 11.7 describes the threats to the validity and other limitations of this study. Finally, Section 11.8 provides a summary and concluding remarks for this study.

11.2 Design

In this study, we are interested in evaluating individual pattern chains meeting a defined selection criteria. The context of each pattern chain are their containing systems. This links directly to **RG4** reiterated in section 11.1 and further connects to the main question as we compare the effects of both injected and observed grime in each case. In the spirit of the GQM, we have further refined the main question into a series of directly answerable questions, their guiding rationale, and a set of metrics defined to facilitate answering these questions. The questions and metrics are as follows:

RQ4.1: Does grime injection have a similar effect on *Analyzability* as the observed effect of grime on *Analyzability*?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on *Analyzability* as the natural process of grime accumulation.

RQ4.2: Does grime injection have a similar effect on *Testability* as the observed effect of grime on *Testability*?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on *Testability* as the natural process of grime accumulation.

RQ4.3: Does grime injection have a similar effect on *Modifiability* as the observed effect of grime on *Modifiability*?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on *Modifiability* as the natural process of grime accumulation.

RQ4.4: Does grime injection have a similar effect on *Modularity* as the observed effect of grime on *Modularity*?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on *Modularity* as the natural process of grime accumulation.

RQ4.5: Does grime injection have a similar effect on *Reusability* as the observed effect of grime on *Reusability*?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on *Reusability* as the natural process of grime accumulation.

RQ4.6: Does grime injection have a similar effect on *Technical Debt Principal* as the observed effect of grime on *Technical Debt Principal*?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on *Technical Debt Principal* as the natural process of grime accumulation.

RQ4.7: Does grime injection have a similar effect on *Technical Debt Interest* as the observed effect of grime on *Technical Debt Interest*?

Rationale: Evaluate the assertion that the process of grime injection reflects the same effect on *Technical Debt Interest* as the natural process of grime accumulation.

M4.1: *Analyzability* – as defined in Section 10.2.1.

M4.2: *Testability* – as defined in Section 10.2.1.

M4.3: *Modifiability* – as defined in Section 10.2.1.

M4.4: *Modularity* – as defined in Section 10.2.1.

M4.5: *Reusability* – as defined in Section 10.2.1.

M4.6: *Technical Debt Principal* – as defined in Section 10.2.1.

M4.7: *Technical Debt Interest* – as defined in Section 10.2.1.

11.3 Selection

We selected software systems from which to extract design patterns instances for analysis from the *Qualitas Corpus*. The *Qualitas Corpus* was selected, as it contains well-known Open Source Java™ systems, each of which has multiple versions. The selected systems provide the context within which studied pattern chains exist. For this study we have selected the systems from the *Qualitas Corpus* shown in Table 11.1.

Table 11.1: Software systems and their version ranges selected for evaluation from the *Qualitas Corpus*.

	Versions				**Versions**		
System	**Min**	**Max**	**Evaluated**	**System**	**Min**	**Max**	**Evaluated**
jgrapht	0.4.0	0.8.3	3	**trove**	0.0.1	3.0.0	5
jmoney	0.2.0	0.4.4	3	**jag**	2.2.0	6.1.0	3
fitjava	1.0	1.1	3	**quickserver**	1.0	1.4.7	12
drawswf	1.0.1	1.1.1	3	**sunflow**	0.05.1	0.07.2	2
webmail	0.6.1	0.7.6	3	**informa**	0.2.0	0.7.0alpha1	6
nekohtml	0.9.5	1.9.18	4	**marauroa**	2.6.3	3.8.1	2
jsXe	0.1.1	0.4beta	11	**sablecc**	3.1	4_beta.4	2

The subjects of this study are not simply entire systems, nor are they only individual design pattern instances, but rather pattern instance pairs from within the same pattern chain. Thus, we require the ability to identify pattern chains for each system studied. The range of each system's versions is shown in Table 11.1 starting with the min version number and ending with the max version number. Next, we extract adjacent pairs of pattern instances from each chain to form study units via the data collection process. Within each study unit, we call the earlier version of the pattern the “Base Version” and the latter version the “Natural Evolution Version”. Pairs must only meet the requirement that there is a difference in the accumulated grime between each instance. With this in mind, we next describe the data collection process used in this study.

11.4 Data Collection

The following subsections describes the data collection process, how this data is to be stored, and the data to be collected.

11.4.1 Data Collection Process

An overview of the data collection process is depicted in Figure 11.1. As shown, the process is broken down into two phases. The first phase extracts pattern chains from separate versions of selected projects. The second phase conducts the verification study.

Phase 1: Study Unit Extraction This process follows the path indicated by the numbers encircled in green, as follows: 1.) Initially, we have selected several systems from the *Qualitas Corpus*, as listed in Table 11.1. For each system version, we use the Arc framework to 2.) extract Java artifact information, 3.) identify design patterns using the Pattern4 tool, and 4.) collect design pattern grime data for each pattern instance found. 5.) Once each system and its versions have been analyzed, we begin identifying all pattern chains. This process uses the Pattern Chaining Algorithm (Algorithm 4.3), connecting pattern instances across system versions. 6.) For each pattern chain in the system, a comparative grime analysis is conducted. This analysis compares the grime detected in each pair of instances in a chain to identify when detected grime changed between versions. Pairs for which such changes are identified are marked for extraction as study units. 7.) These marked pairs

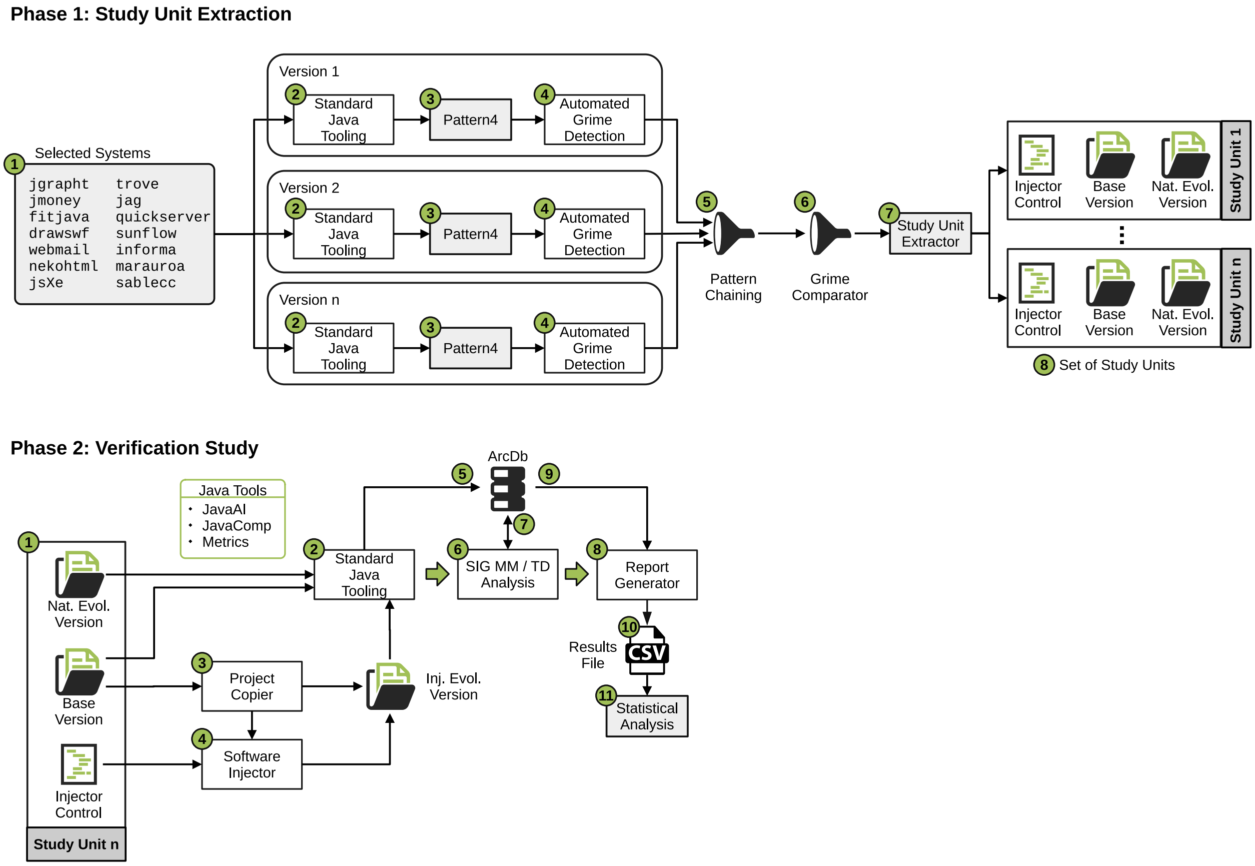


Figure 11.1: Data collection process.

are then extracted into study unit projects along with an injector control file. The lower version number is extracted as the "Base Version," and the higher version is extracted as the "Natural Evolution Version". The injector control file lists the grime introduced between versions and the locations where the grime occurred. The extracted pattern instances are provided with the standard gradle/maven project structure and a gradle build file. Once all study units are extracted, the Verification Study phase may be commenced.

Phase 2: Verification Study This phase follows the path indicated by the numbers encircled in green. The thick green arrows indicate a separation of phases, wherein the prior steps must complete execution prior to traversing the green arrow. This process evaluates

Table 11.2: An example data table (note: this represents a complete table, that was separated into two for space concerns, thus the Unit column is the same for both versions).

	Natural Evolution Version							**Injected Evolution Version**						
Unit	Analyz.	Test.	Modif.	Modul.	Resus.	TD Prin.	TD Int.	Analyz.	Test.	Modif.	Modul.	Resus.	TD Prin.	TD Int.
0	0.01	0.2	0.2	0.0	0.25	1.0	0.03	0.01	0.2	0.2	0.0	0.25	1.0	0.03
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

each study unit as follows: 1.) Projects are constructed for the study unit's "Base Version" and "Natural Evolution Version". 2.) The "Natural Evolution Version" and "Base Version" are analyzed using the Standard Java Tooling. 3.) The "Base Version" project is then copied and an "Injected Evolution Version" project is created. 4.) The newly created "Injected Evolution Version" is injected with the grime specified by the "Injector Control File". The injected project is then analyzed using the Standard Java Tooling. 5.) Each time a project is analyzed using the Java Standard Tooling, the collected data is stored in the associated ArcDb. 6.) Once the basic Java analysis is complete for all three projects of the study unit, the Metrics, SIG Maintainability Model, and TD Analyses are conducted. 7.) The results of these analyses are stored in the ArcDb. 8.) The process extracts these results and combines them into a report, which 10.) is stored as the results CSV file (as described in the following subsection). 11.) Finally, these results are analyzed according to the analysis procedures defined in Section 11.5.

11.4.2 Data to be Collected

For each pattern chain under study, we extract the chain identifier, quality attribute values, technical debt principal, and technical debt interest values from the PatternInstance and Measures tables of the ArcDb. This data is then accumulated into a table, similar to the example shown in Table 11.2, with the following specifications:

- Each row of the table represents a study unit analyzed.
- The first column of the table is the unique study unit identifier.
- The second through sixth columns of the table represent the maintainability subcharacteristic values measured for the “Normal Evolution Version” of the pattern chain.
- The seventh and eighth columns of the table represents the technical debt principal and interest values, respectively, measured for the “Normal Evolution Version” of the pattern chain.
- The ninth through thirteenth columns of the table represent the maintainability subcharacteristic values measured for the “Injected Evolution Version” of the pattern chain.
- The fourteenth and fifteenth columns of the table represents the technical debt principal and interest values, respectively, measured for the “Injected Evolution Version” of the pattern chain.

11.5 Analysis Procedure

We will use the following analysis procedures in answering the questions posed in Section 11.2. The collected data represents the differences between the “Normal Evolution Version” and the “Base Version” and the “Injected Evolution Version” and the “Base Version” on software quality attributes and technical debt principal and interest. To answer research questions **RQ4.1–RQ4.7** we need a method to compare each pair of results. We expect that if the injected grime effects present an accurate representation of the behavior of grime, then we should see similar changes in the measured qualities of the injected project as we see in the natural evolution of the code. However, due to the nature of the injection process, the magnitude of the changes will not necessarily be the same. Thus, we are looking for agreement in the changes for both the natural grime and injected grime systems. In order to evaluate the agreement, we will use Cohen's κ measure of agreement [54].

Cohen's κ has several assumptions. The first is that the data is nominal scale data. The second is that the raters have been deliberately selected. In the case of this study, the rater is the mapping process from raw attribute change data to the nominal scale. This mapping converts each raw measure of change into one of three values: **L** if the difference is less than zero, **E** if the difference is equal to zero, and **G** if the difference is greater than 0. The third assumption is that this is an inter-rater or intra-rater problem. An inter-rater problem is one in which two raters (typically an individual) rate the same items. We wish to determine the level of agreement between these two individuals within the context of the rated items. The intra-rater problem focuses on a single rater rating two different items and evaluating the degree of agreement between two evaluations by the same rater. In the case of this study, we have a rater which is the quality and technical debt measurements system. The ratings are then the mapping to L, E, or G classes. We should expect that the ratings should be the same if the underlying injection system mimics the actual changes that occur as grime evolves. Thus, this study meets all the assumptions for using Cohen's κ.

The κ statistic is then calculated using the following formula:

$$\kappa = \frac{p_a - p_e}{1 - p_e}$$

p_a is the percent agreement between raters, and p_e is the theoretical probability of chance agreement. The values of p_a and p_e are found by constructing a confusion matrix for the ratings. The columns of the matrix are the ratings of the first rater, and the rows represent the ratings of the second rater. Each cell in the matrix contains a count of the number of units rated with the intersecting ratings from each rater. Using this matrix, p_a becomes the sum of values along the diagonal divided by the sum of all the cells.

Additionally, p_e is then the sum of the probability for each rating. For each rating, we find the percentage for the column and the row of that rating. That is, we sum the column values for the rating and divide by the total. We also sum the row values for the rating and divide this value by the total. Next, we find the product of these values to find the probability of the rating. We then sum all probabilities of the ratings together to find p_e. To make this more salient, we next present an example.

In order to make the calculation and assessment of Cohen's κ clear, let us consider an example. Using the data from Table 11.7, below, we can derive the confusion matrix shown in Table 11.3. In this table, each cell represents the count of pairs with the assigned ratings. So, for example, 3 study units were assigned an L for the Natural Evolution and an L for the Inject Evolution. The cells along the right margin represent row totals (or Natural Evolution totals), and the cells along the bottom margin represent the column totals (or Injected Evolution totals). The lower right cell represents the total number of values (in this case, 8). Next we can calculate p_a as the sum of the diagonal divided by the total number of values, or $p_a = \frac{4}{8} = 0.5$. Next, we calculate, p_e which is the sum of the probabilities for each rating. In this case, $p_e = p_L + p_E + p_G$. These values are calculated as follows:

Table 11.3: Example confusion matrix with margin values for use in calculating Cohen's Kappa for Analyzability.

		Injected Evolution			
	Ratings	L	E	G	Nat Total
Natural Evolution	L	3	0	1	4
	E	0	1	1	2
	G	2	0	0	2
	Inj Total	5	1	2	8

$$\begin{aligned} p_L &= \frac{\Sigma L_{Nat}}{Total} * \frac{\Sigma L_{Inj}}{Total} \\ &= \frac{4}{8} * \frac{5}{8} \\ &= 0.5 * 0.625 \\ &= 0.3125 \end{aligned}$$

$$\begin{aligned} p_E &= \frac{\Sigma E_{Nat}}{Total} * \frac{\Sigma E_{Inj}}{Total} \\ &= \frac{2}{8} * \frac{1}{8} \\ &= 0.25 * 0.125 \\ &= 0.03125 \end{aligned}$$

$$\begin{aligned} p_G &= \frac{\Sigma G_{Nat}}{Total} * \frac{\Sigma G_{Inj}}{Total} \\ &= \frac{2}{8} * \frac{2}{8} \\ &= 0.25 * 0.25 \\ &= 0.0625 \end{aligned}$$

Thus, $p_e = 0.3125 + 0.03125 + 0.0625 = 0.40625$ which tells us that the theoretical level of chance agreement is 40.625%. Using this value and p_a from above, we can calculate κ as follows:

$$\begin{aligned}
\kappa &= \frac{p_a - p_e}{1 - p_e} \\
&= \frac{0.5 - 0.40625}{1 - 0.40625} \\
&= \frac{0.09375}{0.59375} \\
&= 0.157894737
\end{aligned}$$

This reaches the same approximate value for κ produced by R and found in Table 11.8.

The level of agreement can then be interpreted using the mappings in Table 11.4, as set forth by Landis and Koch [158]. In the case of perfect agreement, κ would be 1, while in the case of no agreement (or no better than chance agreement), κ would be equal to 0. We note that κ may be less than zero, which indicates that the level of agreement was less than the level of the agreement due to chance.

11.6 Results and Discussion

11.6.1 Study Unit Extraction

As the first part of this study, we executed the pattern chain detection and study unit extraction process. The results of this process are shown in Table 11.5 and are summarized as follows. This process led to the evaluation of a total of 62 software project versions. From these versions, we extracted a total of 230 pattern chains comprised of 705 design pattern instances. However, upon analyses of the chains and contained instances, we could only identify eight instance pairs between which a change in the grime occurred.

Table 11.4: Cohen's κ agreement level mappings.

κ	**Agreement Level**
<0	Less than chance agreement
0	Chance agreement
0.01 – 0.20	Slight
0.21 – 0.40	Fair
0.41 – 0.60	Moderate
0.61 – 0.80	Substantial
0.81 – 1.0	Perfect

These identified instance pairs came from only three projects: fitjava, drawswf, and quickserver, as shown in Table 11.5. We extracted each pair, forming study units for the verification study phase. The details of each extracted study unit are shown in Table 11.6. In this table, we note the unit number, the system, the base (lower) version number, the natural evolution (upper) version number, the pattern type, and the grime type (and the number of instances to be injected in parentheses, if other than one) to be injected. The types of grime injected are based on the change in grime between versions. Interestingly, only Package and Class Grime types appear to change within the evaluated pattern chains.

11.6.2 Verification Study

The resulting data collected during the verification study phase is shown in Table 11.7. In this table, each pair of rows corresponds to a single unit of study and is identified by the row pair number under the Unit column (please note that the pattern type associated with each study unit is written below the unit number in italics). The top row of each pair corresponds to the Natural (Nat) evolution of the pattern, while the bottom row corresponds to the Injected (Inj) evolution of the pattern instance. The remaining columns correspond

Table 11.5: Selected projects and the number of versions, patterns identified, pattern chains identified, and the number of study units identified in each project.

Project	**Versions Evaluated**	**Instances**	**Chains**	**Units**
jgrapht	3	27	9	0
jmoney	3	6	2	0
fitjava	3	12	4	**5**
drawswf	3	6	2	**1**
webmail	3	84	28	0
nekohtml	4	12	3	0
jsXe	11	66	6	0
trove	5	140	28	0
jag	3	24	8	0
quickserver	12	24	2	**2**
sunflow	2	118	59	0
informa	6	42	7	0
marauroa	2	106	53	0
sablecc	2	38	19	0
Total	**62**	**705**	**230**	**8**

to each of the quality/technical debt attributes of concern. The columns are in the following order: Analyzability (Ana.), Testability (Test.), Modularity (Modul.), Modifiability (Modif.), Reusability (Reus.), Technical Debt Principal (TD P.), and Technical Debt Interest (TD I.). The values in each table cell are then the ratings derived from the raw data collected during the study. The ratings in each cell are based on the mapping described in Section 11.5. Additionally, we identify matched ratings for each attribute within a study unit by bold type.

Table 11.6: Study units extracted.

Unit ID	System	Version		Pattern Type	Δ Grime
		Lower	Upper		
1	quickserver	1.0	1.1	State	PECG
2	quickserver	1.0	1.1	State	MPEUG, PICG
3	fitjava	1.0	1.0RC1	Singleton	PECG (2), MPECG, MTECG
4	fitjava	1.0RC1	1.1	Adapter	PECG
5	fitjava	1.0	1.0RC1	Template M.	MPECG (3), MTECG (3), PECG
6	fitjava	1.0RC1	1.1	Template M.	PECG
7	drawswf	1.1.0	1.1.1	State	PIRG
8	fitjava	1.0RC1	1.1	Singleton	IEPG (7), PECG

Table 11.7 also serves to compare the effects observed for both the natural and injected evolution versions against the results from the experiments in Chapter 10. Those cells marked with light blue have a result that appears to correspond with the results of the experiments. This comparison considers the pattern type and grime type for the unit of study in question. Additionally, the entire column for Reusability is marked in gray, as there are no corresponding results from the experiments. This marking provides the following key findings:

- The percentage of the natural pattern evolution instances demonstrate a similar effect as seen in the simulation experiments. This percentage allows us to evaluate how closely the experimental results from Chapter 10 represent the effects of grime on pattern instances in actual software. The results show that only 41.07% of the instances reflect

Table 11.7: Verification study results.

Unit		**Ana.**	**Test.**	**Modul.**	**Modif.**	**Reus.**	**TD P.**	**TD I.**
1	Nat	E	E	E	**E**	E	E	E
State	Inj	G	L	G	**E**	G	G	G
2	Nat	**L**	**L**	**L**	**E**	**G**	**G**	**G**
State	Inj	**L**	**L**	**L**	**E**	**G**	**G**	**G**
3	Nat	G	**L**	E	**E**	G	G	G
Singleton	Inj	L	**L**	L	**E**	L	L	L
4	Nat	**E**	E	E	E	E	E	E
Adapter	Inj	**E**	L	L	G	L	L	L
5	Nat	L	G	G	**G**	G	G	G
Template M.	Inj	G	L	L	**G**	L	L	L
6	Nat	**L**	G	**L**	**G**	G	G	G
Template M.	Inj	**L**	L	**L**	**G**	L	L	L
7	Nat	**L**	G	**L**	**E**	**L**	G	G
State	Inj	**L**	L	**L**	**E**	**L**	L	L
8	Nat	G	G	E	**G**	**L**	G	G
Singleton	Inj	L	L	L	**G**	**L**	L	L

the experimental results.

- The percentage of injected pattern evolution instances that reflect the experimental results in actual pattern instances. This percentage evaluate how well the injection approach works within actual pattern instances. The data shows that only 52.083% of the instances reflect the experimental results.

Next, these results are analyzed using Cohen's Kappa on each pair of each unit for each

Table 11.8: Analysis results

Attribute	κ	**Agreement Level**
Analyzability	0.16	Slight
Testability	0.0	None
Modularity	0.048	Slight
Modifiability	0.75	**Substantial**
Reusability	0.091	Slight
TD Principal	-0.077	None
TD Interest	-0.077	None

attribute of concern to measure the agreement between pairs. Table 11.8 presents the results of this analysis. As one would expect from looking at Table 11.7, there is little agreement to be found, with two notable exceptions. The first is the substantial agreement across all units for Modifiability ($\kappa = 0.75$). The second, as shown in Table 11.7, is that for Unit 2 there is 100% agreement across all attributes of concern. Furthermore, each unit analyzed has at least one attribute (and several with two or more) with matching ratings.

11.6.3 Discussion

With these results in hand, we now look to address the research questions described in Section 11.1. These results suggest slight agreement between injected and natural evolution for Analyzability, Modularity, and Reusability. However, the number of natural evolution instances corresponding to the experimental results for these attributes is relatively low (excluding Reusability for which there is no data). Thus, there is a lack of evidence to confirm questions RQ 4.1, 4.3, and 4.5. Similarly, Testability, TD Principal, and TD Interest show no agreement and a lack of correspondence to the experimental data for the natural evolution instances. Thus, we have little evidence to confirm RQ 4.2, 4.6, and 4.7. However, there does appear to be strong evidence for RQ 4.3. Both the level of agreement between the natural and injected evolution instances match, and there is a significant correspondence between the natural evolution instances and the experimental data.

One possible issue that may have led to the disconnect between the Normal Evolution and Injected Evolution results and between the Normal Evolution and Experimental results is as follows. There is an implicit assumption that the change in grime when there are multiple types is additive. The comparison to experimental results relied upon an informal comparison of the amount of grime injected, pattern type of the instance, and grime type adding the expected changes to determine if it would net increase, decrease, or effectively zero for comparison to what was observed. However, if the assumption that grime effects are additive is false, these results are invalid. This particular assumption applies to those study units that include multiple types of grime (units 2, 3, 5, and 8). Future studies, will need to be address this issue to improve the validity and reliability of the results.

These results overall indicate that the approach is not without merit. However, they strongly indicate that further research is necessary. Specifically, these results suggest that the results of the experiments do not hold outside of those experiments (i.e., within existing software systems). The final takeaway from this is that several additional iterations will be needed to refine and improve the injection process to represent real-world phenomena.

11.7 Threats to Validity

This section describes the limitations and threats to the validity of this study. Specifically, we focus on threats to conclusion validity, internal validity, construct validity, content validity, external validity, and reliability in accordance with the frameworks proposed by Campbell and Cook [44], Campbell and Stanley [45], and Wohlin et al. [276] for experimentation and the insights of Yin [285] and Runeson et al. [230] concerning case studies.

Internal Validity

This validity check is concerned with the ability to show a causal relationship between outcomes and treatments. The prior studies showed the causal relationship between grime and quality and technical debt, via the injection process. In this study we validated this relationship with evidence that the nature of the injection process matches that of observed occurrences of grime in open source systems. Thus, there are no threats to internal validity.

Construct Validity

This validity check is concerned with using correct operational measures for the concepts studied. We utilized detection techniques based directly on the definitions of each form of grime presented in Chapter 9 and by Schanz and Izurieta [234]. Furthermore, operationalized measures for the main quality characteristics from the ISO/IEC 25010 specification as operationalized by our implementation of the SIG Maintainability Model. As noted in Section 10.5.2, there are threats to construct validity due to our implementation of the SIG Maintainability Model and due to our use of Nugroho et al.'s approach to measuring TD Interest. An additional threat to construct validity is the assumption that the change in grime between the Base Version and the Normal Evolution Version is accumulation. This assumption poses a severe challenge to the validity of the results.

Content Validity

Content validity is concerned with how well the selected measures cover the content domain. The selected measures are based on the ISO/IEC 25010 definitions for the subcharacteristics of maintainability and based on current knowledge concerning technical debt including both measures of principal and interest. Thus, there are no threats to content validity.

External Validity

This validity check is concerned with the ability to generalize the results of a study. This study was conducted across multiple software systems and included multiple types of design patterns. Unfortunately, as the study considered only open-source software systems implemented in Java™, we cannot extend our results beyond open-source Java software systems, which pose a threat to the external validity of this study. Additionally, the small sample size, the lack of grime type coverage, and the lack of pattern type coverage threaten external validity and the generalizability of the results.

Reliability

Reliability is concerned with the dependence between specific researchers and the data and analysis. In this study we utilized the Arc Framework to gather quantitative data in an automated fashion. Thus, the data collected is not reliant on any researcher nor are the values collected left to interpretation. Furthermore the analysis conducted utilized scripts written for the R statistical processing software. The version of R used and the scripts used will be made available to all researchers. Although, each of these mitigate any potential threats to reliability, one threat remains due to the manual detection of design pattern grime during the first phase of the data collection process.

11.8 Conclusion

In this Chapter, we have conducted an example of a Verification Experiment from Phase 3 of the process described in Chapter 8. This study was the continuation of the experiments completed in Chapter 10. This study compared the effects of grime from a pattern instance's natural evolution to that of an injected evolution on the Analyzability, Testability, Modifiability, Modularity, Reusability, TD Principal, and TD Interest. We found that with the current implementation of the Software Injection approach (see Chapter 6), there is an apparent disconnect between the effects induced by injection and the effects observed in existing software systems. We note that one primary exception to this was substantial agreement (as measured by Cohen's κ) for Modifiability. Thus, while the injection approach does require refinement, this speaks to the need for the process defined in Chapter 8.

This process, by its very nature, is inherently iterative. Therefore, we should expect to use the results of the verification study to improve the experimental process until we reach the point of achieving substantial or near-perfect agreement for all attributes of concern. In addition, we should use this process to ensure that the injected instances are representative of actual phenomena. In short, for our future work, we will need to return to the software injector, making refinements until the verification study reaches its goals. We are also interested in collecting additional study units such that we capture each pattern type, each grime type, and multiple combinations of grime as well. Once the injection process is refined, we can then turn our attention to Phases 4 and 5 of the process. In these phases, we will consider evaluating other properties of the design pattern grime within the context of multiple and longitudinal case studies.

CHAPTER TWELVE

CONCLUSIONS AND FUTURE WORK

Optimism is an occupational hazard of programming: feedback is the treatment.

–Kent Beck

We have detailed the underlying methods to study and extend our knowledge of design pattern disharmonies known as design pattern grime. To facilitate this, we developed a general software artifact disharmony and issue research process. Following this process we have extended the theoretical design pattern grime taxonomy to include extensions for both class and organization grime and operationalized a software framework we call Arc. From this taxonomy and an initial study involving design pattern grime [101], we further developed the underlying goals and questions used to guide the remainder of this research.

Initial studies also helped inform the tooling which was later refined into the Arc Framework. This framework forms the backbone for collecting the data from each of the experiments and case studies conducted by following our research process. As a part of this framework, and using the concepts from the taxonomy, we developed tools to integrate data from several sources including: existing issue detection tools such as SpotBugs and PMD, Pattern4 pattern detection tool, software metrics, quality and technical debt analysis, and design pattern grime detection. We conducted several experiments and a verification study using generated design patterns and those collected from open source software systems written in the Java™programming language.

The remainder of this chapter is organized as follows: Section 12.1 synthesizes our results and their relationship to existing work concerning design pattern grime. Section 12.2 describes the impact of the results and methods developed herein to researchers and practitioners and summarizes the limitations of the results of the experiments and verification study. Finally, Section 12.3 describes the immediate future work and extensions we are proposing for this research.

12.1 Relationship to Existing Evidence

We developed an overarching process that allows for both experimentation and case study research. With the goals to provide a consistent platform that allows for predictable replication and to form a single approach with the capability of leading to a deeper understanding than either approach may provide separately. We realized this process into a concrete framework which we call the Arc Framework. Within the Arc Framework, we have implemented several quality models and metrics, including an implementation of Quamoco (based on our prior work [129]), QMOOD (based on our prior work [103]), and the SIG Maintainability Model. Additionally, we have implemented two methods of measuring Technical Debt. The first is an implementation of the CAST method for measuring Technical Debt Principal (based on our prior work [103]), and the second is the first known implementation of Nugroho's approach for measuring both Technical Debt Principal and Interest. Finally, underlying these quality and technical debt models are the metrics (based on our prior work [104, 105]) measure the quality of the analyzed system. Thus, the Arc Framework provides the underlying platform for the execution of the studies found within this dissertation.

The studies found herein focused on the phenomena known as Design Pattern Grime. The first phase of the process developed/refined the taxonomies of Class, Modular, and Organizational Grime as first described by Izurieta [132]. We further refined Class Grime based on a taxonomy from our earlier work [101]. Next, we refined the taxonomy for Modular Grime based on the work of Schanz and Izurieta [234]. Finally, we developed the initial taxonomy of Organizational Grime [134]. Additionally, we defined each type of grime within a unified logical framework. Next, using the Arc Framework, we developed an approach to simulate the effects of grime on software design patterns, which we call Software Injection. This approach was based on prior work of Griffith and Izurieta [101], and Dale and Izurieta [62] but with the new capability of directly injecting into source code as opposed to data structures in memory or within object bytecode. We use this approach to study grime *in-vitro*.

Using this capability and the addition of the Arc Framework's ability to integrate several collected data from existing tools, we conducted seven experiments evaluating generated design pattern instances injected with design pattern grime. These experiments constituted Phase 2 of the general process. Furthermore, the results of these experiments showed similar results concerning the effects of modular Grime on Technical Debt Principal as that of Dale and Izurieta [62]. As the severity of grime increases, there is a corresponding increase in Technical Debt Principal. Additionally, the experimental results appear to provide further support for the implications to testability first identified by Izurieta and Bieman [128, 133].

Finally, we executed a verification study using the Arc Framework to identify design pattern instances across software versions (forming design pattern chains) found in real systems. These studies utilized automated design pattern grime detection to identify when grime changed between versions of a pattern. The ability to automate grime detection was based on the work of Marinescu [179] and our prior work [104, 105]. The implementation of automated detection for grime constitutes a significant capability that has hindered prior research. We then evaluated identified instances to further explore how well the experimental results match the results from existing software systems and if the software injection system requires fine-tuning by comparing the results of the natural evolution of a pattern instance to those of an injected instance.

12.2 Impact and Limitations

As noted in the prior section, we have developed an approach to studying software phenomena and their properties. Using the approach, we developed a concrete framework, that allows for consistency of experimentation, to serve as the platform to conduct such studies. Using this framework, we conducted both a series of simulation experiments and a verification study to understand Design Pattern Grime. The experimental results showed that design pattern grime can significantly affect both software Maintainability and Technical Debt. These effects depend on the grime severity level, the grime type, and the design pattern type. The results of the verification study suggest that the current software injection system requires some fine-tuning and calibration to align it completely with what occurs in existing software systems.

These results have the following implications on both engineers working in industry and on other researchers. For researchers, we have developed a promising approach that can be applied to various software phenomena across many domains within software engineering. Examples can include security vulnerabilities, architectural issues, and build automation issues. This process also fills a gap by providing a means by which empirical researchers in these areas may bridge the divide between experimentation and case study research. For practitioners in particular, as part of this research, we have developed an open implementation of the SIG Maintainability Model and the tools to calibrate it, which with some work can be transferred to industry to provide both empirically driven quality and technical analysis of software systems. Additionally, there may be implications for the evolution of design patterns grime, which requires further study.

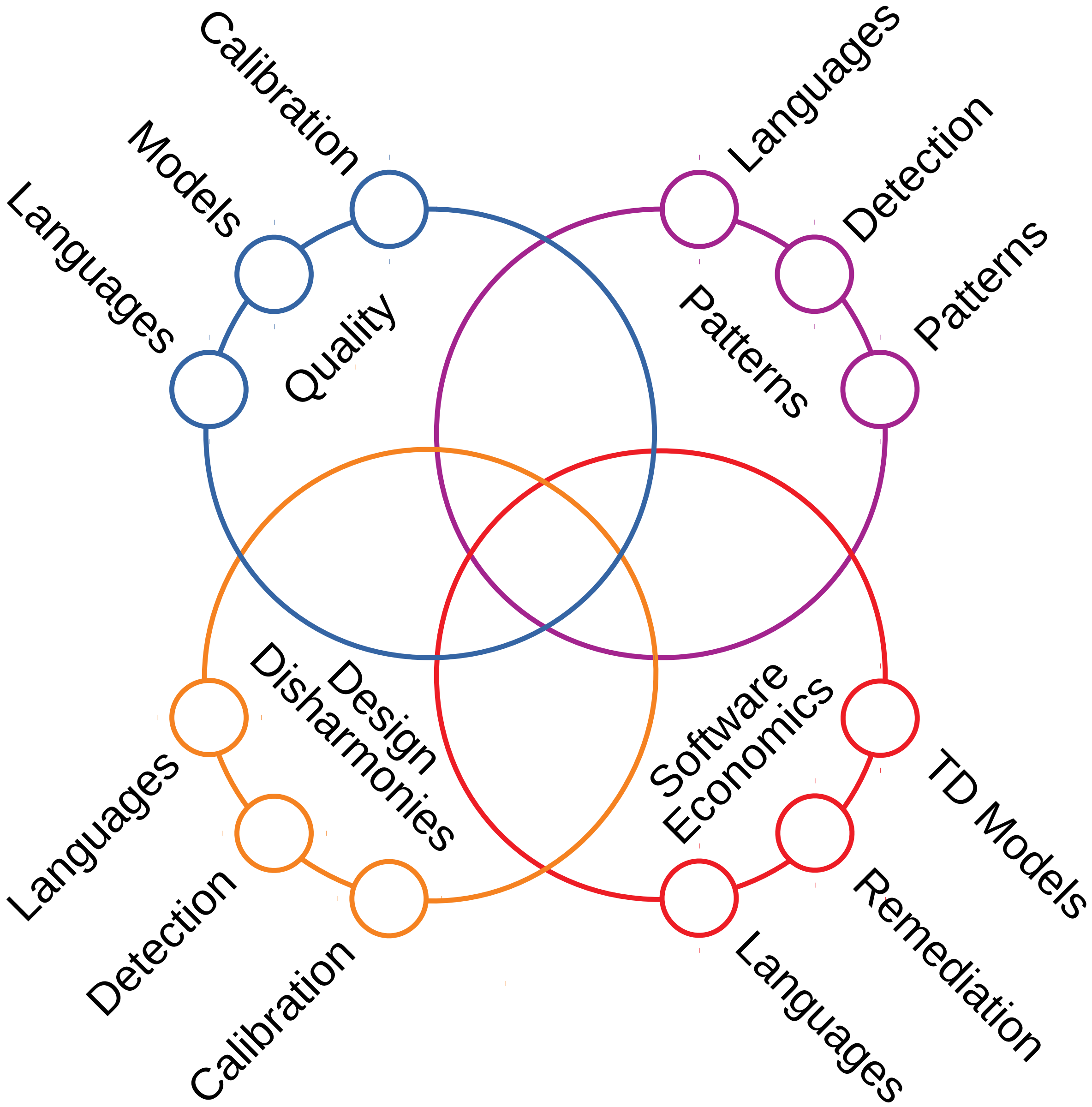


Figure 12.1: Dimensions of future work.

12.3 Future Work

In the future, we plan to extend this work along four dimensions, as depicted in Figure 12.1. The first is in the domain of Software Quality Analytics. Though we have developed implementations of the Quamoco, QMOOD, and SIG Maintainability quality measurement frameworks, as described in Chapter 5 several extensions are needed to facilitate continued research. First, Quamoco provides a quality modeling meta-model. Using this meta-model, we intend to capture our other implementations as instances of the meta-model while also capturing other more recent quality measurement approaches such as QATCH [242], its fork PIQUE [1], among others. Next, although the Quamoco framework is language agnostic, its underlying models are not. Thus we intend to extend the capabilities by developing community vetted models for languages beyond Java™. Next, we intend to improve the capabilities of Quamoco through research into multi-level calibration techniques. Finally, we intend to resolve our current limitations in technical debt measurement by developing implementations of current and proposed techniques.

The second dimension is that of design pattern analysis. Currently, we are utilizing the SSA method implemented in the tool Pattern4 for design pattern detection. Our goal is to re-implement this technique using our underlying Arc Data Model, without relying on language-specific techniques, to provide a language-agnostic approach to design pattern detection. We also intend to extend our data cleansing techniques to incorporate validation of patterns using specifications provided in RBML and Elemental Design Patterns [245]. The goal of this is to improve the detection results and extend the number of design patterns capable of being detected.

The third dimension is that of Design Disharmonies. Here we are concerned with the analysis of grime and further exploring Design Pattern Rot and the relations between these and other well-known disharmonies (i.e. code smells, antipatterns, and modularity violations). This research has the goal of improving detection, knowledge, and prediction of their effects on quality.

The final dimension focuses on technical debt analysis. We intend to incorporate design

[1] `https://github.com/msusel-pique/msusel-pique`

pattern grime and rot into improved models of technical debt. The goal is to develop practitioner-level models which aid in the remediation of technical debt across languages but which also provide similar measures of Technical Debt Principal and Interest that models such as Nugroho et al.'s approach [203] provide.

REFERENCES CITED

[1] *OMG Unified Modeling Language™ (OMG UML), Infrastructure*, Object Management Group, 2011.

[2] *Object Constraint Language*, Object Management Group, 2014.

[3] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, *Automatic package coupling and cycle minimization*, 2009 16th working conference on reverse engineering, 2009Oct, pp. 103–112.

[4] Md Abdullah Al Mamun, Christian Berger, and Jorgen Hansson, *Explicating, understanding, and managing technical debt from self-driving miniature car projects*, 2014 sixth international workshop on managing technical debt, 2014, pp. 11–18.

[5] Ethem Alpaydin, *Introduction to machine learning*, 2nd ed, Adaptive computation and machine learning, MIT Press, Cambridge, Mass, 2010.

[6] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shull, and Carolyn Seaman, *Identification and management of technical debt: A systematic mapping study*, Information and Software Technology **70** (February 2016), 100–121 (en).

[7] Nicolli S.R. Alves, Leilane F. Ribeiro, Vivyane Caires, Thiago S. Mendes, and Rodrigo O. Spínola, *Towards an ontology of terms on technical debt*, 2014 sixth international workshop on managing technical debt, 2014, pp. 1–7.

[8] Theodoros Amanatidis, Nikolaos Mittas, Athanasia Moschou, Alexander Chatzigeorgiou, Apostolos Ampatzoglou, and Lefteris Angelis, *Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities*, Empirical Software Engineering **25** (September 2020), no. 5, 4161–4204 (en).

[9] Apostolos Ampatzoglou, Olia Michou, and Ioannis Stamelos, *Building and mining a repository of design pattern instances: Practical and research benefits*, Entertainment Computing **4** (April 2013), no. 2, 131–142 (en).

[10] Areti Ampatzoglou, Apostolos Ampatzoglou, Paris Avgeriou, and Alexander Chatzigeorgiou, *Establishing a framework for managing interest in technical debt.*

[11] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Paris Avgeriou, Pekka Abrahamsson, Antonio Martini, Uwe Zdun, and Kari Systa, *The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study*, 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD), October 2016, pp. 9–16.

[12] Areti Ampatzoglou, Alexandros Michailidis, Christos Sarikyriakidis, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou, *A framework for managing interest in technical debt: an industrial validation*, Proceedings of the 2018 International Conference on Technical Debt, May 2018, pp. 115–124 (en).

[13] Areti Ampatzoglou, Nikolaos Mittas, Angeliki-Agathi Tsintzira, Apostolos Ampatzoglou, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou, Paris Avgeriou, and Lefteris Angelis, *Exploring the Relation between Technical Debt Principal and Interest: An Empirical Approach*, Information and Software Technology **128** (December 2020), 106391 (en).

[14] T. W. Anderson and D. A. Darling, *A Test of Goodness of Fit*, Journal of the American Statistical Association **49** (December 1954), no. 268, 765.

[15] G. Antoniol, R. Fiutem, and L. Cristoforetti, *Design pattern recovery in object-oriented software*, Proceedings. 6th international workshop on program comprehension. iwpc'98 (cat. no.98tb100242), 1998, pp. 153–160.

[16] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino, *Comparing and experimenting machine learning techniques for code smell detection*, Empirical Software Engineering (June 2015) (en).

[17] Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Stamatia Bibi, Alexander Chatzigeorgiou, and Ioannis Stamelos, *Monitoring Technical Debt in an Industrial Setting*, Proceedings of the Evaluation and Assessment on Software Engineering, April 2019, pp. 123–132 (en).

[18] A. Asencio, S. Cardman, D. Harris, and E. Laderman, *Relating expectations to automatically recovered design patterns*, Ninth working conference on reverse engineering, 2002. proceedings., 2002, pp. 87–96.

[19] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta, *An empirical study on the evolution of design patterns*, Proceedings of the the 6th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of software engineering, 2007, pp. 385–394.

[20] Paris Avgeriou, Apostolos Ampatzoglou, Areti Ampatzoglou, and Alexander Chatzigeorgiou, *Establishing a Framework for Managing Interest in Technical Debt:*, Proceedings of the Fifth International Symposium on Business Modeling and Software Design, 2015, pp. 75–85.

[21] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman, *Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)*, Dagstuhl Reports **6** (2016), no. 4, 110–138. Place: Dagstuhl, Germany Publisher: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] Paris C. Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimaki, Darius Daniel Sas, Saulo Soares de Toledo, and Angeliki Agathi Tsintzira, *An Overview and Comparison of Technical Debt Measurement Tools*, IEEE Software **38** (May 2021), no. 3, 61–71.

[23] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser, *Standardized code quality benchmarking for improving software maintainability*, Software Quality Journal **20** (June 2012), no. 2, 287–307 (en).

[24] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy, *A probabilistic software quality model*, 2011 27th ieee international conference on software maintenance (icsm), 2011, pp. 243–252.

[25] J. Bansiya and C.G. Davis, *A hierarchical model for object-oriented design quality assessment*, IEEE Transactions on Software Engineering **28** (January 2002), no. 1, 4–17.

[26] Jagdish Bansiya, *Automating design-pattern identification*, Dr. Dobb's journal **23** (1998), no. 6.

[27] F. Hutton Barron and Bruce E. Barrett, *Decision quality using ranked attribute weights*, Management Science **42** (1996), no. 11, 1515–1523.

[28] Victor R. Basili, *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*, University of Maryland at College Park, College Park, MD, USA, 1992.

[29] D. Beyer, A. Noack, and C. Lewerentz, *Simple and efficient relational querying of software structures*, 10th working conference on reverse engineering, 2003. wcre 2003. proceedings., 2003, pp. 216–225.

[30] James M. Bieman and Byung-Kyoo Kang, *Cohesion and reuse in an object-oriented system*, ACM SIGSOFT Software Engineering Notes **20** (August 1995), no. SI, 259–262 (en).

[31] J.M. Bieman, G. Straw, H. Wang, P.W. Munger, and R.T. Alexander, *Design patterns and change proneness: an examination of five evolving systems*, Proceedings. 5th international workshop on enterprise networking and computing in healthcare industry (ieee cat. no.03ex717), 2003, pp. 40–49.

[32] D. Bijlsma, *Indicators of issue handling efficiency*, Master's Thesis, 2010.

[33] A. Blewitt, A. Bundy, and I. Stark, *Automatic verification of java design patterns*, Proceedings 16th annual international conference on automated software engineering (ase 2001), 2001, pp. 324–327.

[34] Barry Boehm and Li Guo Huang, *Value-based software engineering: A case study*, Computer **36** (2003), no. 3, 33–41.

[35] C. E. Bonferroni, *Il calcolo delle assicurazioni su gruppi di teste*, Studi in onore del professore salvatore ortu carboni, 1935, pp. 13–60.

[36] Cédric Bouhours, Hervé Leblanc, and Christian Percebois, *Bad smells in design and design patterns*, Journal of Object Technology **8** (2009), no. 3, 43–63.

[37] Cédric Bouhours, Hervé Leblanc, and Christian Percebois, *Sharing bad practices in design to improve the use of patterns*, Proceedings of the 17th conference on pattern languages of programs, 2010.

[38] Cédric Bouhours, Hervé Leblanc, and Christian Percebois, *Spoiled patterns: how to extend the GoF*, Software Quality Journal (August 2014) (en).

[39] L.C. Briand, J.W. Daly, and J.K. Wust, *A Unified Framework for Cohesion Measurement in Object-Oriented Systems*, Empirical Software Engineering **3** (1998), no. 1, 65–117. 10.1023/A:1009783721306.

[40] L.C. Briand, S. Morasca, and V.R. Basili, *Measuring and assessing maintainability at the end of high level design*, 1993 conference on software maintenance, 1993, pp. 88–87.

[41] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter, *Exploring the relationships between design measures and software quality in object-oriented systems*, Journal of Systems and Software **51** (May 2000), no. 3, 245–273 (en).

[42] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka, *Managing technical debt in software-reliant systems*, Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 47–52.

[43] Dénes Bán and Rudolf Ferenc, *Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability*, Computational Science and Its Applications – ICCSA 2014, 2014, pp. 337–352.

[44] D. Campbell and T. D. Cook, *Quasi-experimentation: Design and Analysis Issues for Field Settings*, Houghton Mifflin Company, 1979.

[45] D. Campbell and J. Stanley, *Experimental and Quasi-experimental Designs for Research*, Rand-McNally, 1963.

[46] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou, *Assessing code smell interest probability: a case study*, Proceedings of the XP2017 Scientific Workshops, May 2017, pp. 1–8 (en).

[47] Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Areti Ampatzoglou, and Theodoros Amanatidis, *Estimating the breaking point for technical debt*, 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), October 2015, pp. 53–56.

[48] Shyam R. Chidamber and C.F. Kemerer, *Towards a metrics suite for object oriented design*, 1991, pp. 197–211 (en).

[49] S.R. Chidamber and C.F. Kemerer, *A metrics suite for object oriented design*, Software Engineering, IEEE Transactions on **20** (June 1994), no. 6, 476 –493.

[50] S. Chin, E. Huddleston, W. Bodwell, and I. Gat, *The Economics of Technical Debt*, Cutter IT Journal **23** (2010), no. 10, 11–15.

[51] Zadia Codabux and Christopher Dutchyn, *Profiling Developers Through the Lens of Technical Debt*, Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), October 2020, pp. 1–6 (en).

[52] Zadia Codabux and Byron Williams, *Managing technical debt: An industrial case study*, Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013, pp. 8–15.

[53] Zadia Codabux, Byron J. Williams, Gary L. Bradshaw, and Murray Cantor, *An empirical assessment of technical debt practices in industry*, Journal of Software: Evolution and Process **29** (October 2017), no. 10, e1894 (en).

[54] Jacob Cohen, *A coefficient of agreement for nominal scales*, Educational and Psychological Measurement **20** (1960), no. 1, 37–46, available at `https://doi.org/10.1177/001316446002000104`.

[55] Jose Pedro Correia, Yiannis Kanellopoulos, and Joost Visser, *A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics*, 2009 ieee international conference on software maintenance, 2009, pp. 61–70.

[56] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, *Design pattern recovery by visual language parsing*, Ninth european conference on software maintenance and reengineering, 2005, pp. 102–111.

[57] ———, *Case studies of visual language based design patterns recovery*, Conference on software maintenance and reengineering (csmr'06), 2006, pp. 10 pp.–174.

[58] Ward Cunningham, *The WyCash portfolio management system*, SIGPLAN OOPS Mess. **4** (December 1992), no. 2, 29–30.

[59] B. Curtis, J. Sappidi, and A. Szynkarski, *Estimating the Principal of an Application's Technical Debt*, Software, IEEE **29** (December 2012), no. 6, 34 –42.

[60] ———, *Estimating the size, cost, and types of Technical Debt*, Managing Technical Debt (MTD), 2012 Third International Workshop on, June 2012, pp. 49 –53.

[61] Melissa Dale, *Impacts of Modular Grime on Technical Debt*, Master's Thesis, Bozeman, MT, 2014.

[62] Melissa R. Dale and Clemente Izurieta, *Impacts of design pattern decay on system quality*, Proceedings of the 8th acm/ieee international symposium on empirical software engineering and measurement, 2014.

[63] J. de Groot, A. Nugroho, T. Back, and J. Visser, *What is the value of your software?*, Managing Technical Debt (MTD), 2012 Third International Workshop on, June 2012, pp. 37 –44.

[64] George Digkas, Alexander N Chatzigeorgiou, Apostolos Ampatzoglou, and Paris C Avgeriou, *Can Clean New Code reduce Technical Debt Density*, IEEE Transactions on Software Engineering (2020), 1–1.

[65] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou, *How do developers fix issues and pay back technical debt in the Apache ecosystem?*, 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), March 2018, pp. 153–163.

[66] Jing Dong, Dushyant S. Lad, and Yajing Zhao, *Dp-miner: Design pattern discovery using matrix*, 14th annual ieee international conference and workshops on the engineering of computer-based systems (ecbs'07), 2007, pp. 371–380.

[67] Jing Dong, Sheng Yang, and Kang Zhang, *Visualizing Design Patterns in Their Applications and Compositions*, IEEE Transactions on Software Engineering **33** (July 2007), no. 7, 433–453.

[68] Charles W. Dunnett, *A Multiple Comparison Procedure for Comparing Several Treatments with a Control*, Journal of the American Statistical Association **50** (December 1955), no. 272, 1096–1121 (en).

[69] A.H. Eden, A. Yehudai, and J. Gil, *Precise specification and automatic application of design patterns*, Proceedings 12th ieee international conference automated software engineering, 1997, pp. 143–152.

[70] Ward Edwards and F.Hutton Barron, *Smarts and smarter: Improved simple methods for multiattribute utility measurement*, Organizational Behavior and Human Decision Processes **60** (1994), no. 3, 306–325.

[71] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, *Does code decay? Assessing the evidence from change management data*, IEEE Transactions on Software Engineering **27** (January 2001), no. 1, 1–12.

[72] Eric Eide, Alastair Reid, John Regehr, and Jay Lepreau, *Static and dynamic structure in design patterns*, Proceedings of the 24th international conference on software engineering, 2002, pp. 208–218.

[73] Davide Falessi, Michele A. Shaw, Forrest Shull, Kathleen Mullen, and Mark Stein Keymind, *Practical considerations, challenges, and requirements of tool-support for managing technical debt*, Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013, pp. 16–19.

[74] Davide Falessi and Alexander Voegele, *Validating and prioritizing quality rules for managing technical debt: An industrial case study*, 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), October 2015, pp. 41–48.

[75] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang, *Statistical power analyses using g* power 3.1: Tests for correlation and regression analyses*, Behavior research methods **41** (2009), no. 4, 1149–1160.

[76] Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, and Elisa Y. Nakagawa, *Correlating Pattern Grime and Quality Attributes*, IEEE Access **6** (2018), 23065–23078.

[77] Daniel Feitosa, Paris Avgeriou, Apostolos Ampatzoglou, and Elisa Yumi Nakagawa, *The evolution of design pattern grime: An industrial case study*, Product-focused software process improvement, 2017, pp. 165–181.

[78] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, *Design pattern mining enhanced by machine learning*, 21st ieee international conference on software maintenance (icsm'05), 2005, pp. 295–304.

[79] Rudolf Ferenc, Péter Hegedűs, and Tibor Gyimóthy, *Software product quality models* (Tom Mens, Alexander Serebrenik, and Anthony Cleve, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[80] Carlos Fernandez-Sanchez, Juan Garbajosa, and Agustin Yague, *A framework to aid in decision making for technical debt management*, 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), October 2015, pp. 69–76.

[81] Carlos Fernandez-Sanchez, Hector Humanes, Juan Garbajosa, and Jessica Diaz, *An Open Tool for Assisting in Technical Debt Management*, 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), August 2017, pp. 400–403.

[82] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, *JDeodorant: Identification and Removal of Feature Envy Bad Smells*, Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, October 2007, pp. 519 –520.

[83] F.A. Fontana, P. Braione, and M. Zanoni, *Automatic detection of bad smells in code: An experimental assessment*, Journal of Object Technology **11** (2012), no. 2.

[84] F.A. Fontana, V. Ferme, and S. Spinelli, *Investigating the impact of code smells debt on quality code evaluation*, Managing Technical Debt (MTD), 2012 Third International Workshop on, June 2012, pp. 15 –22.

[85] F.A. Fontana and M. Zanoni, *On Investigating Code Smells Correlations*, Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, March 2011, pp. 474 –475.

[86] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel Martenka, *Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains*, 2013 ieee international conference on software maintenance, 2013, pp. 260–269.

[87] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä, *Code smell detection: Towards a machine learning-based approach*, 2013 ieee international conference on software maintenance, 2013, pp. 396–399.

[88] Martin Fowler, Kent Beck, J Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the Design of Existing Programs*, Addison-Weseley, 1999.

[89] John Fox and Sanford Weisberg, *An R companion to applied regression*, Third, Sage, Thousand Oaks CA, 2019.

[90] R.B. France, Dae-Kyoo Kim, S. Ghosh, and Eunjee Song, *A UML-based pattern specification technique*, IEEE Transactions on Software Engineering **30** (March 2004), no. 3, 193–206 (en).

[91] Savio Freire, Nicolli Rios, Boris Perez, Camilo Castellanos, Dario Correal, Robert Ramac, Vladimir Mandic, Nebojsa Tausan, Gustavo Lopez, Alexia Pacheco, Davide Falessi, Manoel Mendonca, Clemente Izurieta, Carolyn Seaman, and Rodrigo Spinola, *How Experience Impacts Practitioners' Perception of Causes and Effects of Technical Debt*, 2021 IEEE/ACM 13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), May 2021, pp. 21–30.

[92] Savio Freire, Nicolli Rios, Boris Perez, Camilo Castellanos, Dario Correal, Robert Ramac, Vladimir Mandic, Nebojsa Tausan, Alexia Pacheco, Gustavo Lopez, Manoel Mendonca, Clemente Izurieta, Davide Falessi, Carolyn Seaman, and Rodrigo Spinola, *Pitfalls and Solutions for Technical Debt Management in Agile Software Projects*, IEEE Software **38** (November 2021), no. 6, 42–49.

[93] Savio Freire, Nicolli Rios, Boris Perez, Dario Torres, Manoel Mendonca, Clemente Izurieta, Carolyn Seaman, and Rodrigo Spinola, *How do Technical Debt Payment Practices Relate to the Effects of the Presence of Debt Items in Software Projects?*, 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), March 2021, pp. 605–609.

[94] Sávio Freire, Nicolli Rios, Boris Gutierrez, Darío Torres, Manoel Mendonça, Clemente Izurieta, Carolyn Seaman, and Rodrigo O. Spínola, *Surveying Software Practitioners on Technical Debt Payment Practices and Reasons for not Paying off Debt Items*, Proceedings of the Evaluation and Assessment in Software Engineering, April 2020, pp. 210–219 (en).

[95] Sávio Freire, Nicolli Rios, Manoel Mendonça, Davide Falessi, Carolyn Seaman, Clemente Izurieta, and Rodrigo O. Spínola, *Actions and impediments for technical debt prevention: results from a global family of industrial surveys*, Proceedings of the 35th Annual ACM Symposium on Applied Computing, March 2020, pp. 1548–1555 (en).

[96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*, Addison-Wesley, 1994.

[97] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, *Identifying Architectural Bad Smells*, Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on (March 2009), 255–258.

[98] Matt Gatrell, Steve Counsell, and Tracy Hall, *Design patterns and change proneness: A replication using proprietary c# software*, 2009 16th working conference on reverse engineering, 2009, pp. 160–164.

[99] O Gaudin, *Evaluate your technical debt with Sonar*, Sonar, Jun (2009).

[100] Marcela Genero, José Olivas, Mario Piattini, and Francisco Romero, *Using Metrics to Predict OO Information Systems Maintainability*, Advanced Information Systems Engineering, 2001, pp. 388–401.

[101] Isaac Griffith and Clemente Izurieta, *Design pattern decay: The case for class grime*, Proceedings of the 8th acm/ieee international symposium on empirical software engineering and measurement, 2014.

[102] Isaac Griffith, Clemente Izurieta, Hannane Taffahi, and David Claudio, *A simulation study of practical methods for technical debt management in agile software development*, Proceedings of the 2014 Winter Simulation Conference, December 2014, pp. 1014–1025.

[103] Isaac Griffith, Derek Reimanis, Clemente Izurieta, Zadia Codabux, Ajay Deo, and Byron Williams, *The correspondence between software quality models and technical debt estimation approaches*, 2014 sixth international workshop on managing technical debt, 2014, pp. 19–26.

[104] Isaac Griffith, Scott Wahl, and Clemente Izurieta, *Evolution of legacy system comprehensibility through automated refactoring*, Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering, 2011, pp. 35–42.

[105] ———, *TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility*, 24th international conference on computer applications in industry and engineering, November 2011.

[106] Michael Grottke, Rivalino Matias, and Kishor S. Trivedi, *The fundamentals of software aging*, 2008 ieee international conference on software reliability engineering workshops (issre wksp), 2008, pp. 1–6.

[107] Yuepu Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F.Q.B. da Silva, A. L M Santos, and C. Siebra, *Tracking technical debt – An exploratory case study*, Software Maintenance (ICSM), 2011 27th IEEE International Conference on (2011), 528–531.

[108] Yuepu Guo and Carolyn Seaman, *A portfolio approach to technical debt management*, Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 31–34.

[109] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Seaman, *Exploring the costs of technical debt management – a case study*, Empirical Software Engineering (November 2014) (en).

[110] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Raţiu Daniel, *Using concept analysis to detect co-change patterns*, Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, 2007, pp. 83–89.

[111] Tudor Gîrba, Stéphane Ducasse, Radu Marinescu, and Raţiu Daniel, *Identifying Entities That Change Together*, Ninth IEEE Workshop on Empirical Studies of Software Maintenance (WESS 2004), 2004.

[112] Tracy Hall, Min Zhang, David Bowes, and Yi Sun, *Some Code Smells Have a Significant but Small Effect on Faults*, ACM Transactions on Software Engineering and Methodology **23** (September 2014), no. 4, 1–39 (en).

[113] H.H. Hallal, E. Alikacem, W.P. Tunney, S. Boroday, and A. Petrenko, *Antipattern-based detection of deficiencies in Java multithreaded software*, Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on, September 2004, pp. 258 –267.

[114] Ilja Heitlager, Tobias Kuipers, and Joost Visser, *A practical model for measuring maintainability*, 6th international conference on the quality of information and communications technology (quatic 2007), 2007, pp. 30–39.

[115] ———, *A practical model for measuring maintainability*, 6th international conference on the quality of information and communications technology (quatic 2007), 2007, pp. 30–39.

[116] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, *Automatic design pattern detection*, 11th ieee international workshop on program comprehension, 2003., 2003, pp. 94–103.

[117] D. Heuzeroth, S. Mandel, and W. Lowe, *Generating design pattern detectors from pattern specifications*, 18th ieee international conference on automated software engineering, 2003. proceedings., 2003, pp. 245–248.

[118] Johannes Holvitie and Ville Leppanen, *DebtFlag: Technical debt management with a development environment integrated tool*, Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013, pp. 20–27.

[119] M. Hong, Tao Xie, and Fuqing Yang, *Jbooret: an automated tool to recover oo design and source models*, 25th annual international computer software and applications conference. compsac 2001, 2001, pp. 71–76.

[120] Hongbo Liu and Jiaxin Wang, *A new way to enumerate cycles in graph*, Advanced int'l conference on telecommunications and int'l conference on internet and web applications and services (aict-iciw'06), 2006Feb, pp. 57–57.

[121] David Hovemeyer and William Pugh, *Finding bugs is easy*, SIGPLAN Not. **39** (December 2004), no. 12, 92–106.

[122] Heyuan Huang, Shensheng Zhang, Jian Cao, and Yonghong Duan, *A practical pattern recovery approach based on both structural and behavioral analysis*, Journal of Systems and Software **75** (February 2005), no. 1-2, 69–87 (en).

[123] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton, *Software rejuvenation: analysis, module and applications*, Twenty-fifth international symposium on fault-tolerant computing. digest of papers, 1995, pp. 381–390.

[124] Simo Huopio, *A Quest for Indicators of Security Debt*, The Cyber Defense Review **5** (2020), no. 1, 169–184. Publisher: Army Cyber Institute.

[125] ISO, *ISO/IEC 9126-1:2001 Software Engineering – Product Quality – Part 1: Quality Model*, International Organization for Standardization, 2001.

[126] ______, *ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality REqurements and Evaluation (SQuaRE) – System and software quality models*, International Standards Organization, 2011.

[127] C. Izurieta and J.M. Bieman, *How Software Designs Decay: A Pilot Study of Pattern Evolution*, Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on, September 2007, pp. 449 –451.

[128] ______, *Testing Consequences of Grime Buildup in Object Oriented Design Patterns*, Software Testing, Verification, and Validation, 2008 1st International Conference on, April 2008, pp. 171 –179.

[129] C. Izurieta, I. Griffith, and C. Huvaere, *An industry perspective to comparing the sqale and quamoco software quality models*, 2017 acm/ieee international symposium on empirical software engineering and measurement (esem), 2017Nov, pp. 287–296.

[130] C. Izurieta, I. Griffith, D. Reimanis, and R. Luhr, *On the Uncertainty of Technical Debt Measurements*, Information Science and Applications (ICISA), 2013 International Conference on, 2013, pp. 1–4.

[131] C. Izurieta, A. Vetro, N. Zazworka, Yuanfang Cai, C. Seaman, and F. Shull, *Organizing the technical debt landscape*, Managing Technical Debt (MTD), 2012 Third International Workshop on, June 2012, pp. 23 –26.

[132] Clemente Izurieta, *Decay and grime buildup in evolving object oriented design patterns*, Ph.D. Thesis, 2009.

[133] Clemente Izurieta and JamesM. Bieman, *A multiple case study of design pattern decay, grime, and rot in evolving software systems*, Software Quality Journal (2012), 1–35 (English).

[134] Clemente Izurieta, Derek Reimanis, Isaac Griffith, and Travis Schanz, *Structural and behavioral taxonomies of design pattern grime*, 12th seminar on advanced techniques & tools for software evolution, July 2019.

[135] Sebastian Jancke, *Smell Detection in Context*, Ph.D. Thesis, 2010.

[136] Natalia Juristo and Ana M Moreno, *Basics of Software Engineering Experimentation*, Springer US, Boston, MA, 2001 (English).

[137] O. Kaczor, Y.-G. Gueheneuc, and S. Hamel, *Efficient identification of design patterns with bit-vector algorithm*, Conference on software maintenance and reengineering (csmr'06), 2006, pp. 10 pp.–184.

[138] Joshua Kerievsky, *Refactoring to patterns*, Pearson Deutschland GmbH, 2005.

[139] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, *Design Defects Detection and Correction by Example*, Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, June 2011, pp. 81 –90.

[140] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, *Design Defect Detection Rules Generation: A Music Metaphor*, Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, March 2011, pp. 241 –248.

[141] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer, *Search-Based Design Defects Detection by Example*, Fundamental Approaches to Software Engineering, 2011, pp. 401–415.

[142] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui, *Deviance from perfection is a better criterion than closeness to evil when identifying risky code*, Proceedings of the IEEE/ACM international conference on Automated software engineering, 2010, pp. 113–122.

[143] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, *An Exploratory Study of the Impact of Code Smells on Software Change-proneness*, Reverse Engineering, 2009. WCRE '09. 16th Working Conference on, October 2009, pp. 75 –84.

[144] F. Khomh, S. Vaucher, Y. G. Gueheneuc, and H. Sahraoui, *A Bayesian Approach for the Detection of Code and Design Smells*, Quality Software, 2009. QSIC '09. 9th International Conference on (August 2009), 305–314.

[145] Foutse Khomh, *Squad: Software quality understanding through the analysis of design*, 2009 16th working conference on reverse engineering, 2009, pp. 303–306.

[146] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol, *An exploratory study of the impact of antipatterns on class change- and fault-proneness*, Empirical Software Engineering **17** (June 2012), no. 3, 243–275 (en).

[147] D.-K. Kim and J. Whittle, *Generating uml models from domain patterns*, Third acis int'l conference on software engineering research, management and applications (sera'05), 2005, pp. 166–173.

[148] Dae-Kyoo Kim, *Evaluating conformance of uml models to design patterns*, 10th ieee international conference on engineering of complex computer systems (iceccs'05), 2005, pp. 30–31.

[149] Dae-Kyoo Kim and Charbel El Khawand, *An approach to precisely specifying the problem domain of design patterns*, Journal of Visual Languages & Computing **18** (December 2007), no. 6, 560–591 (en).

[150] Dae-Kyoo Kim, R. France, S. Ghosh, and Eunjee Song, *A role-based metamodeling approach to specifying design patterns*, Proceedings 27th annual international computer software and applications conference. compac 2003, 2003, pp. 452–457.

[151] Dae-Kyoo Kim, Robert France, and Sudipto Ghosh, *A UML-based language for specifying domain-specific patterns*, Journal of Visual Languages & Computing **15** (June 2004), no. 3-4, 265–289 (en).

[152] Dae-Kyoo Kim and Wuwei Shen, *Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies*, Software Quality Journal **16** (September 2008), no. 3, 329–359 (en).

[153] S.-K. Kim and D. Carrington, *Using integrated metamodeling to define oo design patterns with object-z and uml*, 11th asia-pacific software engineering conference, 2004, pp. 257–264.

[154] Tim Klinger, Peri Tarr, Patrick Wagstrom, and Clay Williams, *An enterprise perspective on technical debt*, Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 35–38.

[155] Makrina Viola Kosti, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Georgios Pallas, Ioannis Stamelos, and Lefteris Angelis, *Technical Debt Principal Assessment Through Structural Metrics*, 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), August 2017, pp. 329–333.

[156] C. Kramer and L. Prechelt, *Design recovery by automated search for structural design patterns in object-oriented software*, Proceedings of wcre '96: 4rd working conference on reverse engineering, 1996, pp. 208–215.

[157] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya, *Technical Debt: From Metaphor to Theory and Practice*, Software, IEEE **29** (December 2012), no. 6, 18 –21.

[158] J. Richard Landis and Gary G. Koch, *The measurement of observer agreement for categorical data*, Biometrics **33** (1977), no. 1, 159–174.

[159] Jason Lefever, Yuanfang Cai, Humberto Cervantes, Rick Kazman, and Hongzhou Fang, *On the Lack of Consensus Among Technical Debt Detection Tools*, 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), May 2021, pp. 121–130.

[160] J. Letouzey and M. Ilkiewicz, *Managing Technical Debt with the SQALE Method*, Software, IEEE **29** (December 2012), no. 6, 44 –51.

[161] J.-L. Letouzey, *The SQALE method for evaluating Technical Debt*, Managing Technical Debt (MTD), 2012 Third International Workshop on, June 2012, pp. 31 –36.

[162] Howard Levene, *Robust tests for equality of variances1*, Contributions to probability and statistics: Essays in honor of Harold Hotelling **2** (1960), 278–292.

[163] Wei Li and Sallie Henry, *Object-oriented metrics that predict maintainability*, Journal of Systems and Software **23** (November 1993), no. 2, 111–122 (en).

[164] Zengyang Li, Paris Avgeriou, and Peng Liang, *A systematic mapping study on technical debt and its management*, Journal of Systems and Software **101** (2015), 193–220.

[165] Hui Liu, Zhiyi Ma, Weizhong Shao, and Zhendong Niu, *Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort*, Software Engineering, IEEE Transactions on **38** (February 2012), no. 1, 220 –235.

[166] M.T. Llano and R. Pooley, *UML Specification and Correction of Object-Oriented Anti-patterns*, Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on, September 2009, pp. 39 –44.

[167] Mark Lorenz and Jeff Kidd, *Object-oriented software metrics: a practical guide*, Prentice Hall object-oriented series, PTR Prentice Hall, Englewood Cliffs, NJ, 1994.

[168] Lunjin Lu and Dae-Kyoo Kim, *Required behavior of sequence diagrams: Semantics and refinement*, 2011 16th ieee international conference on engineering of complex computer systems, 2011, pp. 127–136.

[169] ______, *Required behavior of sequence diagrams: Semantics and conformance*, ACM Transactions on Software Engineering and Methodology **23** (March 2014), no. 2, 1–28 (en).

[170] Bart Luijten and Joost Visser, *Faster defect resolution with higher technical quality of software*, Technical Report Series TUD-SERG-2010-006 (2010).

[171] Yixin Luo, A. Hoss, and D.L. Carver, *An ontological identification of relationships between anti-patterns and code smells*, Aerospace Conference, 2010 IEEE, March 2010, pp. 1 –10.

[172] Rim Mahouachi, Marouane Kessentini, and Khaled Ghedira, *A New Design Defects Classification: Marrying Detection and Correction*, Fundamental Approaches to Software Engineering, 2012, pp. 455–470.

[173] Somayeh Malakuti and Sergey Ostroumov, *The Quest for Introducing Technical Debt Management in a Large-Scale Industrial Company*, Software Architecture, 2020, pp. 296–311 (en). Series Title: Lecture Notes in Computer Science.

[174] Vladimir Mandic, Nebojsa Tausan, Robert Ramac, Savio Freire, Nicolli Rios, Boris Perez, Camilo Castellanos, Dario Correal, Alexia Pacheco, Gustavo Lopez, Clemente Izurieta, Davide Falessi, Carolyn Seaman, and Rodrigo Spinola, *Technical and Nontechnical Prioritization Schema for Technical Debt: Voice of TD-Experienced Practitioners*, IEEE Software **38** (November 2021), no. 6, 50–58.

[175] Vladimir Mandić, Nebojša Taušan, and Robert Ramač, *The prevalence of the technical debt concept in serbian it industry: Results of a national-wide survey*, Proceedings of the 3rd international conference on technical debt, 2020, pp. 77–86.

[176] Usman Mansoor, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb, *Code-smells detection using good and bad software design examples*, Technical report, Technical Report, 2013.

[177] M. Mantyla, J. Vanhanen, and C. Lassenius, *A taxonomy and an initial empirical study of bad smells in code*, Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, September 2003, pp. 381 –384.

[178] Salvatore T. March and Gerald F. Smith, *Design and natural science research on information technology*, Decision Support Systems **15** (1995), no. 4, 251–266.

[179] R. Marinescu, *Detection strategies: metrics-based rules for detecting design flaws*, Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, September 2004, pp. 350 –359.

[180] ———, *Assessing technical debt by identifying design flaws in software systems*, IBM Journal of Research and Development **56** (October 2012), no. 5, 9:1 –9:13.

[181] Radu Marinescu, *Assessing and Improving Object-Oriented Design* (2012).

[182] R.C. Martin, *Oo design quality metrics–an analysis of dependencies*, Proceedings of the workshop pragmatic and theoretical directions in object-oriented software metrics, October 1994.

[183] ———, *Agile software development: principles, patterns, and practices*, Prentice Hall PTR, 2003.

[184] Jabier Martinez, Nuria Quintano, Alejandra Ruiz, Izaskun Santamaria, Iker Martinez de Soria, and Jose Arias, *Security Debt: Characteristics, Product Life-Cycle Integration and Items*, 2021 IEEE/ACM International Conference on Technical Debt (TechDebt), May 2021, pp. 1–5.

[185] Antonio Martini and Jan Bosch, *The magnificent seven: towards a systematic estimation of technical debt interest*, Proceedings of the XP2017 Scientific Workshops, May 2017, pp. 1–5 (en).

[186] Antonio Martini, Jan Bosch, and Michel Chaudron, *Architecture technical debt: Understanding causes and a qualitative model*, 2014 40th euromicro conference on software engineering and advanced applications, 2014, pp. 85–92.

[187] ———, *Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study*, Information and Software Technology **67** (November 2015), 237–253 (en).

[188] Antonio Martini, Simon Vajda, Rajesh Vasa, Allan Jones, Mohamed Abdelrazek, John Grundy, and Jan Bosch, *Technical debt interest assessment: from issues to project*, Proceedings of the XP2017 Scientific Workshops, May 2017, pp. 1–6 (en).

[189] Mitin Mathur, *Java Smell Detector*, Ph.D. Thesis, 2011.

[190] T.J. McCabe, *A complexity measure*, IEEE Transactions on Software Engineering **SE-2** (1976Dec), no. 4, 308–320.

[191] Steve McConnell, *Managing Technical Debt*, Technical Report 1, Construx, 2008.

[192] T. Mens and T. Tourwe, *A declarative evolution framework for object-oriented design patterns*, Proceedings ieee international conference on software maintenance. icsm 2001, 2001, pp. 570–579.

[193] T. Miceli, H.A. Sahraoui, and R. Godin, *A metric based technique for design flaws detection and correction*, Automated Software Engineering, 1999. 14th IEEE International Conference on., October 1999, pp. 307 –310.

[194] P.F. Mihancea and R. Marinescu, *Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems*, Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on, March 2005, pp. 92 –101.

[195] T. Mikkonen, *Formalizing design patterns*, Proceedings of the 20th international conference on software engineering, 1998, pp. 115–124.

[196] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, *DECOR: A Method for the Specification and Detection of Code and Design Smells*, Software Engineering, IEEE Transactions on **36** (February 2010), no. 1, 20 –36.

[197] N. Moha, Y.-G. Gueheneuc, and P. Leduc, *Automatic Generation of Detection Algorithms for Design Defects*, Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on, September 2006, pp. 297 –300.

[198] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Françoise Le Meur, and Laurence Duchien, *A Domain Analysis to Specify Design Defects and Generate Detection Algorithms*, Fundamental Approaches to Software Engineering, 2008, pp. 276–291.

[199] Moha, Naouel, Huynh, Duc-loc, and Guéhéneuc Y-G, *A Taxonomy and a First Study of Design Pattern Defects*, IEEE International Workshop on Software Technology and Engineering Practice, 2005, pp. 225–229.

[200] M.J. Munro, *Product metrics for automatic identification of "bad smell" design problems in java source-code*, 11th ieee international software metrics symposium (metrics'05), 2005, pp. 15–15.

[201] Mika Mäntylä, *Bad Smells in Software – a Taxonomy and an Empirical Study*, Ph.D. Thesis, 2003.

[202] R.L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, *In Search of a Metric for Managing Architectural Technical Debt*, Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on, 2012, pp. 91–100.

[203] Ariadi Nugroho, Joost Visser, and Tobias Kuipers, *An empirical model of technical debt and interest*, Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 1–8.

[204] Felix Ocker, Matthias Seitz, Marius Oligschlager, Minjie Zou, and Birgit Vogel-Heuser, *Increasing Awareness for Potential Technical Debt in the Engineering of Production Systems*, 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), July 2019, pp. 478–484.

[205] M.C. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin, *Code decay analysis of legacy software through successive releases*, 1999 ieee aerospace conference. proceedings (cat. no.99th8403), 1999, pp. 69–81 vol.5.

[206] S.M. Olbrich, D.S. Cruzes, and D.I.K. Sjoberg, *Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems*, Software Maintenance (ICSM), 2010 IEEE International Conference on, September 2010, pp. 1 –10.

[207] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka, *The evolution and impact of code smells: A case study of two open source systems*, 2009 3rd international symposium on empirical software engineering and measurement, 2009, pp. 390–400.

[208] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum, *Maintainability defects detection and correction: a multi-objective approach*, Automated Software Engineering **20** (March 2013), no. 1, 47–79 (en).

[209] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk, *Detecting bad smells in source code using change history information*, 2013 28th ieee/acm international conference on automated software engineering (ase), 2013, pp. 268–278.

[210] David Lorge Parnas, *Software Aging*, Proceedings of the 16th International Conference on Software Engineering. ICSE'97 (May 1994), 279–287.

[211] Błażej Pietrzak and Bartosz Walter, *Leveraging Code Smell Detection with Inter-smell Relations*, Extreme Programming and Agile Processes in Software Engineering, 2006, pp. 75–84.

[212] I. Polasek, P. Liska, J. Kelemen, and J. Lang, *On extended Similarity Scoring and Bit-vector Algorithms for design smell detection*, Intelligent Engineering Systems (INES), 2012 IEEE 16th International Conference on, June 2012, pp. 115 –120.

[213] Boris Pérez, Juan Pablo Brito, Hernán Astudillo, Darío Correal, Nicolli Rios, Rodrigo Oliveira Spínola, Manoel Mendonça, and Carolyn Seaman, *Familiarity, causes and reactions of software practitioners to the presence of technical debt: A replicated study in the chilean software industry*, 2019 38th international conference of the chilean computer science society (sccc), 2019, pp. 1–7.

[214] Boris Pérez, Camilo Castellanos, Darío Correal, Nicolli Rios, Sávio Freire, Rodrigo Spínola, Carolyn Seaman, and Clemente Izurieta, *Technical debt payment and prevention through the lenses of software architects*, Information and Software Technology **140** (December 2021), 106692 (en).

[215] Narayan Ramasubbu and Chris F. Kemerer, *Towards a model for optimizing technical debt in software products*, Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013, pp. 51–54.

[216] Robert Ramač, Vladimir Mandić, Nebojša Taušan, Nicolli Rios, Manoel G. de Mendonca Neto, Carolyn Seaman, and Rodrigo Oliveira Spínola, *Common causes and effects of technical debt in serbian it: Insightd survey replication*, 2020 46th euromicro conference on software engineering and advanced applications (seaa), 2020, pp. 354–361.

[217] D. Rapu, S. Ducasse, T. Girba, and R. Marinescu, *Using history information to improve design flaws detection*, Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on, March 2004, pp. 223 –232.

[218] D. Ratiu, R. Marinescu, S. Ducasse, and T. Gırba, *Evolution-enriched detection of god classes*, Proc. of the 2nd CAVIS (2004), 3–7.

[219] Derek Reimanis and Clemente Izurieta, *Towards assessing the technical debt of undesired software behaviors in design patterns*, 2016 ieee 8th international workshop on managing technical debt (mtd), 2016, pp. 24–27.

[220] ______, *Behavioral Evolution of Design Patterns: Understanding Software Reuse Through the Evolution of Pattern Behavior*, Reuse in the Big Data Era, 2019, pp. 77–93 (en).

[221] Kalle Rindell and Johannes Holvitie, *Security Risk Assessment and Management as Technical Debt*, 2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), June 2019, pp. 1–8.

[222] Nicolli Rios, Savio Freire, Boris Perez, Camilo Castellanos, Dario Correal, Manoel Mendonca, Davide Falessi, Clemente Izurieta, Carolyn B. Seaman, and Rodrigo Oliveira Spinola, *On the Relationship Between Technical Debt Management and Process Models*, IEEE Software **38** (September 2021), no. 5, 56–64.

[223] Nicolli Rios, Leonardo Mendes, Cristina Cerdeiral, Ana Patrícia F. Magalhães, Boris Perez, Darío Correal, Hernán Astudillo, Carolyn Seaman, Clemente Izurieta, Gleison Santos, and Rodrigo Oliveira Spínola, *Hearing the voice of software practitioners on causes, effects, and practices to deal with documentation debt*, Requirements engineering: Foundation for software quality, 2020, pp. 55–70.

[224] Nicolli Rios, Manoel G Mendonça, Carolyn Seaman, and Rodrigo O Spínola, *Causes and effects of the presence of technical debt in agile software projects* (2019).

[225] Nicolli Rios, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola, *A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners*, Information and Software Technology **102** (October 2018), 117–145 (en).

[226] Nicolli Rios, Rodrigo Oliveira Spinola, Manoel Mendonca, and Carolyn Seaman, *Supporting Analysis of Technical Debt Causes and Effects with Cross-Company Probabilistic Cause-Effect Diagrams*, 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), May 2019, pp. 3–12.

[227] Nicolli Rios, Rodrigo Oliveira Spínola, Manoel Mendonça, and Carolyn Seaman, *The most common causes and effects of technical debt: first results from a global family of industrial surveys*, Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, October 2018, pp. 1–10 (en).

[228] Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh, *Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes*, 2012 19th working conference on reverse engineering, 2012, pp. 437–446.

[229] Naveen Roperia, *JSmell: A Bad Smell Detection Tool for Java Systems*, Ph.D. Thesis, 2009.

[230] Per Runeson (ed.), *Case study research in software engineering: guidelines and examples*, Wiley, Hoboken, N.J, 2012.

[231] Nyyti Saarimaki, Maria Teresa Baldassarre, Valentina Lenarduzzi, and Simone Romano, *On the Accuracy of SonarQube Technical Debt Remediation Time*, 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), August 2019, pp. 317–324.

[232] M. Salehie, Shimin Li, and L. Tahvildari, *A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws*, 2006.

[233] Jay Sappidi, Bill Curtis, and Alexandra Szynkarski, *The CRASH Report – 2011/12: Summary of Key Findings*, CAST Research Labs, 2012.

[234] Travis Schanz and Clemente Izurieta, *Object oriented design pattern decay: a taxonomy*, Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 2010, pp. 7:1–7:8.

[235] Klaus Schmid, *A formal approach to technical debt decision making*, Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, 2013, pp. 153–162.

[236] ______, *On the limits of the technical debt metaphor some guidance on going beyond*, Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013, pp. 63–66.

[237] ______, *Technical Debt – From Metaphor to Engineering Guidance: A Novel Approach based on Cost Estimation*, Technical Report 1/2013, SSE 1/13/E, Institute of Computer Science, University of Hildesheim, 2013 (English).

[238] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw, *Building empirical support for automated code smell detection*, Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 2010, pp. 8:1–8:10.

[239] Carolyn Seaman and Yuepu Guo, *Measuring and monitoring technical debt*, Advances in Computers **82** (2011), 25–46.

[240] Jochen Seemann and Jürgen Wolff von Gudenberg, *Pattern-based design recovery of Java software*, ACM SIGSOFT Software Engineering Notes **23** (November 1998), no. 6, 10–16 (en).

[241] Nija Shi and Ronald A. Olsson, *Reverse engineering of design patterns from java source code*, 21st ieee/acm international conference on automated software engineering (ase'06), 2006, pp. 123–134.

[242] Miltiadis G. Siavvas, Kyriakos C. Chatzidimitriou, and Andreas L. Symeonidis, *QATCH - An adaptive framework for software product quality assessment*, Expert Systems with Applications **86** (November 2017), 350–366 (en).

[243] Vallary Singh, Will Snipes, and Nicholas A. Kraft, *A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension*, 2014 sixth international workshop on managing technical debt, 2014, pp. 27–30.

[244] Dag I.K. Sjoberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dyba, *Quantifying the Effect of Code Smells on Maintenance Effort*, IEEE Transactions on Software Engineering **39** (August 2013), no. 8, 1144–1156.

[245] J.M. Smith and D. Stotts, *Spqr: flexible automated design pattern extraction from source code*, 18th ieee international conference on automated software engineering, 2003. proceedings., 2003, pp. 215–224.

[246] Mohamed Soliman, Paris Avgeriou, and Yikun Li, *Architectural design decisions that incur technical debt — An industrial case study*, Information and Software Technology **139** (November 2021), 106669 (en).

[247] N. Soundarajan and J.O. Hallstrom, *Responsibilities and rewards: specifying design patterns*, Proceedings. 26th international conference on software engineering, 2004, pp. 666–675.

[248] Armando Sousa, Lincoln Rocha, Ricardo Britto, Zhixiong Gong, and Feng Lyu, *Technical Debt in Large-Scale Distributed Projects: An Industrial Case Study*, 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), March 2021, pp. 590–594.

[249] Robert G. D. Steel, *A Multiple Comparison Sign Test: Treatments Versus Control*, Journal of the American Statistical Association **54** (December 1959), no. 288, 767.

[250] Marek G Stochel, Mariusz R Wawrowski, and Magdalena Rabiej, *Value-Based Technical Debt Model and Its Application*, ICSEA 2012, The Seventh International Conference on Software Engineering Advances, 2012, pp. 205–212.

[251] Shane Strasser, Colt Frederickson, Kevin Fenger, and Clemente Izurieta, *An automated software tool for validating design patterns*, Isca 24th international conference on computer applications in industry and engineering. caine, 2011.

[252] D. Streitferdt, C. Heller, and I. Philippow, *Searching design patterns in source code*, 29th annual international computer software and applications conference (compsac'05), 2005, pp. 33–34 Vol. 1.

[253] Peter Strečanský, Stanislav Chren, and Bruno Rossi, *Comparing maintainability index, SIG Method, and SQALE for technical debt identification*, Proceedings of the 35th Annual ACM Symposium on Applied Computing, March 2020, pp. 121–124 (en).

[254] Toufik Taibi and David Chek Ling Ngo, *Formal specification of design pattern combination using BPSL*, Information and Software Technology **45** (March 2003), no. 3, 157–170 (en).

[255] E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton, and J. Noble, *The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies*, Software Engineering Conference (APSEC), 2010 17th Asia Pacific, December 2010, pp. 336–345.

[256] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha, *Qualitas.class corpus: A compiled version of the qualitas corpus*, SIGSOFT Softw. Eng. Notes **38** (August 2013), no. 5, 1–4.

[257] Ted Theodoropoulos, Mark Hofberg, and Daniel Kern, *Technical debt from the stakeholder perspective*, Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 43–46.

[258] Edith Tom, A. Aurum, and Richard Vidgen, *An exploration of technical debt*, Journal of Systems and Software **0** (2013).

[259] P. Tonella and G. Antoniol, *Object oriented design pattern inference*, Proceedings ieee international conference on software maintenance - 1999 (icsm'99). 'software maintenance for business change' (cat. no.99cb36360), 1999, pp. 230–238.

[260] A. Trifu and R. Marinescu, *Diagnosing design problems in object oriented systems*, Reverse Engineering, 12th Working Conference on, November 2005, pp. 10 pp.

[261] K.S. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova, *Modeling and analysis of software aging and rejuvenation*, Proceedings 33rd annual simulation symposium (ss 2000), 2000, pp. 270–279.

[262] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, *JDeodorant: Identification and Removal of Type-Checking Bad Smells*, Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, April 2008, pp. 329 –331.

[263] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros Halkidis, *Design Pattern Detection Using Similarity Scoring*, IEEE Transactions on Software Engineering **32** (November 2006), no. 11, 896–909.

[264] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk, *When and Why Your Code Starts to Smell Bad*, 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, May 2015, pp. 403–414.

[265] Eva van Emden and Leon Moonen, *Java Quality Assurance by Detecting Code Smells*, Proceedings of the 9th Working Conference on Reverse Engineering, October 2002.

[266] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, *On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test*, Software Engineering, IEEE Transactions on **33** (December 2007), no. 12, 800 –817.

[267] Reinier Vis, Dennis Bijlsma, and Haiyun Xu, *Evaluation criteria trusted product maintainability*, SIG, 2021.

[268] ______, *SIG/TUVIT evaluation criteria trusted product maintainability: Guidance for producers*, SIG, 2021.

[269] Stefan Wagner, *Software Product Quality Control*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013 (en).

[270] Stefan Wagner, Klaus Lochmann, Lars Heinemann, Michael Kläs, Adam Trendowicz, Reinhold Plösch, Andreas Seidi, Andreas Goeb, and Jonathan Streit, *The quamoco product quality modelling and assessment approach*, 2012 34th international conference on software engineering (icse), 2012, pp. 1133–1142.

[271] Stephan Wagner, Lochmann Klaus, Sebastian Winter, Florian Deissenboeck, Elmar Juergens, Markus Herrmannsdoerfer, Lars Heinemann, Michael Kläs, Adam Trendowicz, Jens Heidrich, Reinhold Ploesch, Andreas Goeb, Christian Koemer, Korbinian Schoder, Jonathan Streit, and Christian Schubert, *The quamoco quality meta-model*, Institut für Informatik, Technische Universität München, 2016.

[272] Bartosz Walter and Błażej Pietrzak, *Multi-criteria Detection of Bad Smells in Code with UTA Method*, Extreme Programming and Agile Processes in Software Engineering **3556** (2005), 1159–1161.

[273] Wei Wang and Vassilios Tzerpos, *Design pattern detection in eiffel systems*, 12th working conference on reverse engineering (wcre'05), 2005, pp. 10 pp.–174.

[274] Lothar Wendehals, *Improving design pattern instance recognition by dynamic analysis*, Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA, 2003, pp. 29–32.

[275] Roel Wieringa, *Design science methodology for information systems and software engineering*, Springer, Berlin New York Dordrecht, 2014 (eng). OCLC: 931607131.

[276] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén, *Experimentation in Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012 (en).

[277] S. Wong, Yuanfang Cai, Miryung Kim, and M. Dalton, *Detecting software modularity violations*, Software Engineering (ICSE), 2011 33rd International Conference on, May 2011, pp. 411 –420.

[278] Lu Xiao, *Quantifying architectural debts*, Proceedings of the 2015 10th joint meeting on foundations of software engineering, 2015, pp. 1030–1033.

[279] Gueheneuc Y-G, H. Sahraoui, and F. Zaidi, *Fingerprinting design patterns*, 11th working conference on reverse engineering, 2004, pp. 172–181.

[280] Aiko Yamashita, *Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data*, Empirical Software Engineering **19** (August 2014), no. 4, 1111–1143 (en).

[281] Aiko Yamashita and Steve Counsell, *Code smells as system-level indicators of maintainability: An empirical study*, Journal of Systems and Software **86** (October 2013), no. 10, 2639–2653 (en).

[282] Aiko Yamashita and Leon Moonen, *Do code smells reflect important maintainability aspects?*, 2012 28th ieee international conference on software maintenance (icsm), 2012, pp. 306–315.

[283] ______, *Exploring the impact of inter-smell relations on software maintainability: An empirical study*, 2013 35th international conference on software engineering (icse), 2013, pp. 682–691.

[284] ______, *To what extent can maintenance problems be predicted by code smell detection? – An empirical study*, Information and Software Technology **55** (December 2013), no. 12, 2223–2242 (en).

[285] Robert K. Yin, *Case study research: design and methods*, 4th ed, Applied social research methods, Sage Publications, Los Angeles, Calif, 2009.

[286] Edward Yourdon and Larry L. Constantine, *Structured design: fundamentals of a discipline of computer program and systems design*, Prentice Hall, Englewood Cliffs, N.J, 1979.

[287] Nico Zazworka and Christopher Ackermann, *CodeVizard: a tool to aid the analysis of software evolution*, Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 2010, pp. 63:1–63:1.

[288] Nico Zazworka, Carolyn Seaman, and Forrest Shull, *Prioritizing design debt investment opportunities*, Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 39–42.

[289] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman, *Investigating the impact of design debt on software quality*, Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 17–23.

[290] Nico Zazworka, Antonio Vetro, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, and Forrest Shull, *Comparing Four Approaches for Technical Debt Identification*, Software Quality Journal (2012).

[291] Zhi-Xiang Zhang, Qing-Hua Li, and Ke-Rong Ben, *A new method for design pattern mining*, Proceedings of 2004 international conference on machine learning and cybernetics (ieee cat. no.04ex826), 2004, pp. 1755–1759 vol.3.

APPENDIX A

PGCL DEFINITIONS

A.1 (Object) Adapter

```
start_pattern: basic
start_type: ConcreteAdapter
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    start_method: Request
    public void [[name]]() {
        [[adaptee.name]].[[Adaptee.SpecificRequest.random]]();
    }
    end_method: Request

    [[methods]]
}
end_type: ConcreteAdapter
end_pattern: basic
```

A.2 Bridge

```
start_pattern: basic
start_type: Abstraction
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    start_method: Operation
    public void [[name]]() {
        [[imp.name]].[[Implementor.OperationImpl.name]]();
    }
    end_method: Operation
    [[methods]]
}
end_type: Abstraction

start_type: ConcreteImplementor
import java.util.*;

/**
[[ClassComment]]
```

```
 */
[[typedef]] {

    [[fields]]

    start_method: OperationImpl
    public void [[name]]() {
        System.out.println("Executing [[name]]...");
    }
    end_method: OperationImpl
    [[methods]]
}
end_type: ConcreteImplementor
end_pattern: basic
```

A.3 Chain of Responsibility

```
start_pattern: basic
start_type: Handler
/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: succ
    protected [[Handler.name]] [[succ.name]];
    end_field: succ

    [[fields]]

    start_method: HandleRequest
    public void [[name]]() {
        [[succ.name]].[[name]]();
    }
    end_method: HandleRequest
    [[methods]]
}
end_type: Handler
end_pattern: basic
```

A.4 Command

```
start_pattern: basic
start_type: ConcreteCommand
/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: r
    private [[Receiver.name]] [[r.name]];
    end_field: r

    [[fields]]

    start_method: Execute
    public void [[name]]() {
        [[r.name]].[[Receiver.Action.name]]();
    }
    end_method: Execute
    [[methods]]

    public [[InstName]]([[Receiver.name]] rcvr) {
        this.[[r.name]] = rcvr;
    }
}
end_type: ConcreteCommand

start_type: Command
/**
[[ClassComment]]
 */
[[typedef]] {


    [[fields]]

    [[methods]]

}
end_type: Command

start_type: AbstractCommand
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]
}
end_type: AbstractCommand

start_type: Client
/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: rcvr
    private [[Receiver.name]] [[rcvr.name]];
    end_field: rcvr

    [[fields]]

    public [[InstName]]() {
        [[rcvr.name]] = new [[Receiver.random]]();
        [[Command.name]] cmd = new [[ConcreteCommand.random]]([[rcvr.name]]);
    }

    [[methods]]
}
end_type: Client
end_pattern: basic
```

A.5 Composite

```
start_pattern: basic
start_type: ConcreteComposite
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: children
    List<[[Composite.root]]> [[name]] = new ArrayList<>();
    end_field: children
    [[fields]]

    start_method: Add
    public void [[name]]([[params]]) {
        if ([[param.c]] != null)
            [[children.name]].add([[param.c]]);
    }
    end_method: Add

    start_method: Remove
    public void [[name]]([[params]]) {
        if ([[param.c]] != null)
            [[children.name]].remove([[param.c]]);
    }
    end_method: Remove

    start_method: GetChild
    public [[type]] [[name]]([[params]]) {
        return [[children.name]].get([[param.index]]);
    }
    end_method: GetChild
    [[methods]]
}
end_type: ConcreteComposite
end_pattern: basic
```

A.6 Decorator

```
start_pattern: basic
start_type: ConcreteDecorator
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]
    start_method: Operation
    public void [[name]]() {
        super.[[name]]();
        [[AddedBehavior.name]]();
    }
    end_method: Operation

    public [[InstName]]([[Component.name]] component) {
        super(component);
    }
}
end_type: ConcreteDecorator

start_type: Decorator
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {
    start_field: absComp
    protected [[Component.name]] [[absComp.name]];
    end_field: absComp

    [[fields]]

    [[methods]]
    start_method: Operation
    public void [[name]]() {
        [[absComp.name]].[[name]]();
    }
    end_method: Operation

    public [[InstName]]([[Component.name]] component) {
        this.[[absComp.name]] = component;
    }
}
end_type: Decorator

start_type: AbstractDecorator
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]

    public [[InstName]]([[Component.name]] component) {
        super(component);
    }
}
end_type: AbstractDecorator
end_pattern: basic
```

A.7 Factory Method

```
start_pattern: basic
start_type: ConcreteCreator
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {
    [[fields]]

    start_method: FactoryMethod
    public [[Product.name]] [[name]]() {
        return new [[ConcreteProduct.random]]();
    }
    end_method: FactoryMethod
    [[methods]]
}
end_type: ConcreteCreator

start_type: ConcreteProduct
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]
}
end_type: ConcreteProduct
end_pattern: basic
```

A.8 Flyweight

```
start_pattern: basic
start_type: FlyweightFactory
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]
    start_field: flyweights
    List<[[Flyweight.name]]> [[flyweights.name]] = new ArrayList<>();
    end_field: flyweights

    [[methods]]
    start_method: GetFlyweight
    public [[Flyweight.name]] [[name]](int key) {
        if ([[flyweights.name]].size() > key && [[flyweights.name]].get(key)
   != null) {
            return [[flyweights.name]].get(key);
        } else {
            [[Flyweight.name]] temp = new [[ConcreteFlyweight.random]]();
            [[flyweights.name]].add(temp);
            return temp;
        }
    }
    end_method: GetFlyweight
}
end_type: FlyweightFactory
end_pattern: basic
```

A.9 Observer

```
start_pattern: basic
start_type: ConcreteObserver
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]
}
end_type: ConcreteObserver

start_type: Subject
import java.util.*;

/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: obs
    protected List<[[Observer.name]]> [[obs.name]] = new ArrayList<>();
    end_field: obs

    [[fields]]

    start_method: Attach
    public void [[name]]([[params]]) {
        if ([[param.obsv]] != null)
            [[obs.name]].add([[param.obsv]]);
    }
    end_method: Attach

    start_method: Detach
    public void [[name]]([[params]]) {
        if ([[param.obsv]] != null)
            [[obs.name]].remove([[param.obsv]]);
    }
    end_method: Detach
    start_method: Notify
    public void [[name]]() {
        for ([[Observer.name]] item : [[obs.name]])
            item.[[Observer.Update.name]]();
    }
    end_method: Notify
    [[methods]]
}
end_type: Subject

end_pattern: basic
```

A.10 Prototype

```
start_pattern: basic
start_type: Client
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]
    start_method: Operation
    public void [[name]]() {
        [[proto.name]].[[Clone.name]]();
    }
    end_method: Operation
}
end_type: Client

start_type: ConcretePrototype
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]
    start_method: Clone
    public [[Prototype.name]] [[name]]() {
        return new [[InstName]]();
    }
    end_method: Clone
}
end_type: ConcretePrototype
end_pattern: basic
```

A.11 Proxy

```
start_pattern: basic
start_type: Proxy
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    [[methods]]
    start_method: Request
    public void [[name]]() {
        [[prox.name]].[[name]]();
    }
    end_method: Request
}
end_type: Proxy
end_pattern: basic
```

A.12 Singleton

```
start_pattern: LazyInit
start_type: Singleton
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    protected [[InstName]]() {}

    [[methods]]
}
end_type: Singleton

start_type: ConcreteSingleton
/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: uniqueInstance
    private static [[Singleton.name]] [[uniqueInstance.name]];
    end_field: uniqueInstance
    [[fields]]
    private [[InstName]]() {
        super();
    }

    start_method: GetInstance
    public static [[Singleton.name]] [[name]]() {
        if ([[uniqueInstance.name]] == null)
            [[uniqueInstance.name]] = new [[InstName]]();
        return [[uniqueInstance.name]];
    }
    end_method: GetInstance

    [[methods]]
}
end_type: ConcreteSingleton
end_pattern: LazyInit
```

A.13 State

```
start_pattern: adjacencyList
start_type: Context
/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: currentState
    private [[State.root]] [[currentState.name]];
    end_field: currentState

    [[fields]]

    public [[InstName]]() {
      [[currentState.name]] = [[ConcreteState.random]].instance(this);
    }

    public void changeCurrentState([[State.root]] state) {
        this.[[currentState.name]] = state;
    }

    start_method: Request
    public void [[name]]() {
        [[currentState.name]].[[Handle.name]]();
    }
    end_method: Request
    [[methods]]
}
end_type: Context

start_type: ConcreteState
```

```
/**
[[ClassComment]]
 */
[[typedef]] {

    private static [[name]] instance;
    private [[Context.name]] context;
    [[fields]]

    private [[name]]([[Context.name]] ctx) {
        this.context = ctx;
    }

    public static [[InstName]] instance([[Context.name]] ctx) {
        if (instance == null) {
            instance = new [[InstName]](ctx);
        }
        return instance;
    }

    public void run() {}

    start_method: Handle
    /**
     *
     */
    @Override
    public void [[name]]() {
        context.changeCurrentState([[ConcreteState.random]].instance(context))
   ;
    }
    end_method: Handle

    [[methods]]
}
end_type: ConcreteState
end_pattern: adjacencyList
```

A.14 Strategy

```
start_pattern: basic
start_type: Context
/**
[[ClassComment]]
 */
[[typedef]] {

    start_field: currentStrategy
    private [[Strategy.root]] [[name]];
    end_field: currentStrategy
    [[fields]]

    start_method: ContextOperation
    public void [[name]]() {
        System.out.println("Operation");
    }
    end_method: ContextOperation
    [[methods]]
}
end_type: Context

start_type: ConcreteStrategy
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    start_method: Operation
    /**
     *
     */
    @Override
    public void [[name]]() {
    }
    end_method: Operation

    start_method: StrategyOp
    public void [[name]]() {
        System.out.println("StrategyOp");
    }
    end_method: StrategyOp

    [[methods]]
}
end_type: ConcreteStrategy

end_pattern: basic
```

A.15 Template Method

```
start_pattern: basic
start_type: Template
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]
    [[methods]]
    start_method: Operation
    public void [[name]]() {
        [[callsAll.PrimitiveOp]]
    }
    end_method: Operation
}
end_type: Template
end_pattern: basic
```

A.16 Visitor

```
start_pattern: basic
start_type: ConcreteElement
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    start_method: Accept
    public void [[name]]([[params]]) {
        [[param.vis]].[[Visitor.Visit.name]](this);
    }
    end_method: Accept
    [[methods]]
}
end_type: ConcreteElement

start_type: ConcreteVisitor
/**
[[ClassComment]]
 */
[[typedef]] {

    [[fields]]

    start_method: Visit
    public void [[name]]([[params]]) {
        [[param.elem]].[[ConcreteElement.Operation.name]]();
    }
    end_method: Visit
    [[methods]]
}
end_type: ConcreteVisitor
end_pattern: basic
```