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

by

Isaac Daniel Griffith

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

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

October, 2015

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF FIGURES

Figure Page

ABSTRACT

Technical debt is a financial metaphor describing the tradeoff 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, design pattern grime, modularity violations, or violations of good coding rules. Currently little is known about the relationships between these design disharmonies and current technical debt metrics, between design disharmonies and quality attributes (e.g., maintainability or security), and technical debt metrics and quality attributes.

The focus of this proposal is on evolution and effects of a type of design disharmony called design pattern grime. Design pattern grime is the accumulation of unnecessary or unrelated software artifacts within the classes of a design pattern instance. Since design patterns represent agreed upon methods to solve common problems and are based upon sound principles of good design, the decay of these patterns implies an evolution away from good design. This research will expand the knowledge base of design pattern grime through systematic and empirical evaluation of its effects on quality and technical debt.

Initially, we define an extended taxonomy of design pattern grime based on design principles in conjunction with metrics. Using this taxonomy as a guide we intend to experiment with open source software to evaluate the effect of design pattern grime on software product quality and technical debt. We also propose three case studies in the context of both open source and industry grade software to study the evolution of grime, the relationship between grime subtypes, and the effect of automated monitoring of technical debt in an industry setting.

This research proposes the following contributions: A formalized method for conducting controlled experiments through the automated injection of design disharmonies. An extended taxonomy of design pattern grime. An approach to automate the detection of design pattern grime. An evaluation of the relationships between design pattern grime types. Empirical confirmation of the existence of all known types of design pattern grime. Finally, an evaluation of the effects of grime on quality and technical debt.

INTRODUCTION

In 1992, Ward Cunningham coined the term *technical debt* to illuminate the need to refactor in a financial metaphor for the benefit of stakeholders [1]. Technical debt has since gained traction as a major concern for software engineers and stakeholders alike. Recently, CAST research labs, in an analysis of 745 applications comprising 365 million lines of code, found that on average there is $3.61 of technical debt per line of code [2]. The implication of this result, is that technical debt is a large factor in the long-term cost and sustainability of a software product.

Motivation

The primary goal of technical debt management research is to identify and address technical debt in a proactive way allowing forward progress in product development. Currently, several approaches to measure technical debt [3–11] have been developed, but the underlying connection to existing quality models is either unknown or not present [12]. Furthermore, even if a method to bridge the technical debt and quality gap is identified, neither technical debt measures nor quality models provide guidance towards remediation of the underlying issues, such as design disharmonies.

Design disharmonies, i.e., code smells [13], anti-patterns [14] and design pattern grime [15], form a portion of the technical debt landscape [16]. Although, these disharmonies indicate where a software system is in need of refactoring; little is known of the effects that they have on technical debt or software product quality. Connecting disharmonies with technical debt and quality models may provide a path towards optimization in technical debt management.

To address these issues, this research focuses on design patterns [17]. Empirical results show that design patterns are not immune from the negative side effects of software decay [15, 18–20]. Design patterns are micro-architectures within software systems and unlike the other software structures they can be formally specified using specification languages such as the role-based meta-modeling language (RBML) [21] in conjunction with the Object Constraint Language. As pattern realizations evolve, their structure and relationships tend to deviate from the intended specification. When these deviations introduce artifacts or relationships which are not functionally or structurally necessary, we call this *design pattern grime* [15], further referred to as *grime*. Since design patterns represent agreed upon methods to solve common problems and are based upon sound principles of good design, the decay of these patterns implies an evolution away from good design.

Currently, there are still several gaps in our knowledge of this phenomenon. The impact of grime on the quality of both software products and pattern realizations has only been subject to limited study [15, 19, 20, 22, 23], and has yet to encompass an entire model of quality. As mentioned, grime forms a part of the technical debt landscape, yet to date only the effects of modular grime on technical debt has been explored and only in a limited context [24]. The notion that different subtypes of grime can be interrelated or that subtypes of grime and other design disharmonies types can be related is another area of study still left untouched. Furthermore, the main bottle-neck hindering further investigation of grime is the manual detection process.

In summary, the overarching goal of this research is to evaluate grime and its effects on quality and technical debt by following a measurement driven approach that will further characterize its nature, taxonomy, and effects. Addressing these issues

will bring us closer to a method that allows actionable results from technical debt evaluations of a software product.

Organization

The rest of this proposal is organized as follows: Chapter 2 provides background on the current issues and concepts fundamental to the experiments and case studies conducted. Chapter 3 describes the problem statement, research objectives, summary of the proposed approach, and details the contributions of the proposed research. Chapter 4 extends the taxonomy of grime to include subtypes of class and organizational grime. Chapter 5 details the software injection framework. Chapter 6 describes the experiments and case studies and the underlying methods underpinning these studies. Chapter 7 presents the threats to validity of the experiments and case studies detailed in Chapter 6. Finally, chapter 8 details the time-line to complete the proposed work.

4

BACKGROUND AND RELATED WORK

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

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" [25]. Parnas [26] 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 [25, 27–30].

Design Disharmonies

Design disharmonies are a form of software decay which 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, etc.). They may also be categorized by the types of software artifacts affected, e.g. source code, unit tests, or databases. In the following subsections we discuss design defects affecting software systems at the statement, method, class, pattern, module, and system level.

Code Smells Fowler et al. [13] 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 [31] added 5 additional code smells and helped to further explain several of the original code smells, while providing several new corrective refactorings. Mäntylä [32] and Mäntylä et. al. [33] describe a taxonomy re-classifying the original 22 code smells based on how each affects a system.

Anti-Patterns Brown et al. [14] first identified anti-patterns, which are patterns of doing things incorrectly. Several subtypes of anti-patterns were identified including software development, software architecture, and software project management anti-patterns. They also define a language of anti-patterns, similar to design patterns [17], and what they call mini anti-patterns. Though Brown et. al. [14] compiled a listing of existing anti-patterns, prior evidence was provided by Riel [34].

Modularity Violations Modularity violations, introduced by Wong et. al. [35], are violations of architectural design principles concerning the coupling between different system modules. They propose an approach utilizing design matrix visualizations of the system architecture (focusing of modules). Based on these techniques, Schwanke et al. [36] investigated the use of modularity violations as an approach to evaluate architectural quality. Furthermore, they conducted an empirical enquiry of open source Java™ software systems. Their results showed that the fan-out metric is reliable in predicting locations of future faults. Reimanis et al. [37] later confirmed Schwanke et al.'s results through a replication study.

Design Pattern Disharmonies Initially, Moha et al. [38] 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* is when a design is missing a needed design pattern. *Deformed* patterns are those

which are not correctly implemented according to Gamma et al.'s [17] definition but which are not themselves erroneous. *Excess* is the over use 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 3 were found to be non-harmful deform defects. Furthermore, their research presented and evaluated multiple detection techniques including manual, semi-automatic, and automatic techniques based on a combination of detection strategies and constratint satisfaction techniques. Unfortunately, this taxonomy was not formally defined.

Izurieta and Bieman [18] presented another taxonomy of design pattern decay. Seminal work by Izurieta [15] 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 – 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.

Comparing to the Moha et al. taxonomy, grime relates most closely to the concept of deformed patterns, while rot most closely relates 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/or 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 [19].

The modular grime results led Schanz and Izurieta [39] 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 [19]. Based on this work Izurieta et al. [16] 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 certain lower level code issues that affect design patterns.

More recent work involving design pattern grime has been conducted by Dale and Izurieta [24] and Griffith and Izurieta [23]. Dale and Izurieta evaluated the effects of modular grime on technical debt. Their work show, through experimentation, that temporary modular grime types have the largest effect on technical debt. Griffith and Izurieta further developed the class grime taxonomy and showed, through experimentation, that each type of class grime negatively effects understandability of a pattern instance. Both of these studies utilized an early form of software injection to facilitate the experimental process. Specifically, Dale and Izurieta used a method which injects modular grime into Java™ bytecode [24]. Griffith and Izurieta used a method which modifies a model of the source code [23].

Another line of research into design pattern disharmonies has been conducted by Bouhours et al. [40–42]. They have studied what they term *spoiled patterns* [41].

Spoiled patterns are essentially the results of incomplete or failed instantiation of a design pattern either 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 [40]. 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 [40–42].

Design Disharmony Detection

The notion of design defect detection 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 [43–48], machine learning and artificial intelligence methods [49–55], and a combination of structural information and metrics [56–58].

One of the most widely extended methodologies is the *detection strategies* approach proposed by Marinescu [45]. A detection strategy is a filtering method which utilizes a combination of metric thresholds and set theory to identify probable locations of design disharmonies in code. Ratiu et. al. [59] 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. [60] and Gîrba et al. [61, 62] 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 [47] 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 [47].

To better understand the nature of code smells and to improve detection techniques Pietrzak and Walter [63] 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 [46] 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. [56–58] 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. [64]. 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 do 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 the process of developing the algorithms for detection. This notion of automation has progressed from the semi-automated to fully automated generation of detection algorithms. The first approach was proposed by Mihancea and Marinescu [65]. They used a genetic algorithm to tune the parameters for each filter in order 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 be developed. Kessentini et. al. have developed and evaluated numerous approaches to these problems [54,55,66–69]. 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 out perform the results of the original DECOR [58] rules, on DECOR's own training data.

Mahouachi et al. [67] extend the genetic programming approach by including both detection and correction (via refactoring) together in order to improve both steps simultaneously. Mansoor et al. [68] have also extended the genetic programming approach to utilize a multi-objective approach to increase both precision and recall of the generated rules. Other machine learning based approaches have been developed. Specifically, Khomh et al. [50, 70] have conducted research into the use of Bayesian belief networks to both specify and detect anti-patterns. Recently, Fontana et al. [71, 72] have detailed an approach to use machine learning techniques for code smell and anti-pattern 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 training data exists.

Given the fixation of detection approaches on the use of metrics, it is surprising that there is little empirical research into the feasibility of these approaches. To this end Schumacher et al. [73] conducted an empirical study of automated detection within an industry setting. Using the CodeVizard tool [74], 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 overall maintenance

effort required. Further research into the evaluation of automated detection was conducted by Fontana et al. [75]. This latter study compared four code smell detection tools across six versions of a single Java™ open source project. The results of this study show that the tools evaluated had a tendency to disagree, and that although these tools may prove useful they are far from adequate.

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 been focused specifically on the effects of code smells and anti-patterns on software maintainability. Early work was conducted by Olbrich et al. [76] 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 have a negative impact on maintainability. A larger study by Khomh et al. [77] 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. [78] conducted another longitudinal study across three open source systems. They found that, in the systems studied, the instances of god classes and brain classes studied exhibit less change and defects when normalized for size, contradicting previous results. Later, Kohmh et al. [79] 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. [80] showed that changes are more common in those classes affected by anti-patterns. Another study by Yamashita and Counsell [81]

found that code smells are significantly affected by size, making comparisons between systems of varying size impossible.

Unfortunately change- and fault-proneness are not comprehensive indicators of maintainability. Because of this, Yamashita and Moonen [82] 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 further develop the connection between code smells and maintainability [83]. 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. [84] 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 better understand how maintainability is affected [82, 83, 85].

Following this research, Yamashita and Moonen [86] evaluated the effects of inter-smell relations (previously identified and studied by Pietrzak and Walter [63] and Fontana and Zanoni [87]). 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 that this interaction affects maintainability. Furthermore, Yamashita and Counsell [81] 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. [88]. This study was conducted across the set of Java™ open source systems collected in the *Qualitas Corpus* [89]. 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 that in those systems with a high number of code smells there is a greater indication of deterioration in maintainability. 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. [90].

Bán and Ferenc [91] 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 [92]).

The majority of research has indicated that both code smells and anti-patterns affect quality by negatively impacting maintainability. Yet, only a single study has utilized a known quality model to conduct this evaluation [91]. 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 which will facilitate experimentation in order to provide estimation of effect size as well as causal analysis.

There is also an issue that current approaches are limited to the evaluation of only those items that can be detected by existing tools, thus limiting analysis to code smells and anti-patterns. This is indicative of the need for an approach which can formalize definitions of design disharmonies in a generalizable way. Finally, there is little evidence regarding the relationship between design disharmonies and any

other quality characteristics defined in the ISO/IEC 25010 specification [93] such as: *Functional Suitability*, *Reliability*, *Performance Efficiency*, *Usability*, *Security*, *Compatibility*, and *Portability*.

Technical Debt

Technical debt is a concept introduced by Ward Cunningham [1] 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.

Metaphor, Definition, and Properties

The notions surrounding technical debt until recently have been informal and under-specified. In lieu of this Tom et al. [94] 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 serves to satisfy short term needs. Incremental Debt is debt taken as several small steps but which 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 of how debt occurs, some researchers have furthered the understanding of the metaphor itself. Nugroho et al. [4] 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. [95] look at technical debt from the perspective of enterprise development and indicate that using financial tools, decision theory, stake-holder based quantification, and developing an understanding of unintentional debt are potential avenues of interest. Finally, Theodoropoulas et al. [96] view technical debt from the stakeholder perspective and provide a new definition based on the gap between technology infrastructure of an organization and its impact on quality.

More recent work has looked into the extent and practicality of the technical debt metaphor itself. Specifically, Schmid [97] [98] 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 objectively measure technical debt itself. This is due to the nature of the interest property associated with technical debt items. Since technical debt interest has a probability which indicates whether it may affect the system, we should instead focus not on measuring all technical debt (*potential technical debt*) but rather we should concern ourselves with the debt items that will have an impact (*effective technical debt*) on upcoming feature development or maintenance.

Technical Debt Management

Technical Debt Management comprises the actions of identifying, cataloging, and remediation of debt items. The current industry focus has been on identifying

and tracking debt as part of the working project backlog [99–101] or as part of a separate technical debt list [102–104]. 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 [105]. This influence acts as interest on the debt by increasing the amount of effort required to evolve the software [106].

Guo and Seaman [102–104] proposed a technical debt management framework (TDMF). Central to this framework is the Technical Debt List (TDL) which stores information pertaining to 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. [107] conducted a case study to evaluate the costs of using the TDMF. This study showed that after an initially high startup cost the cost of monitoring and remediation of debt reduces to a reasonable level. Holvitie and Leppänen [108] have further enhanced the TDMF with an approach called DebtFlag. The main purpose of the DebtFlag is to reduce information redundancy to provide more efficient debt propagation evaluation. This aids in more accurate estimation of debt impact, interest, and interest probability.

Schmid [97, 98, 109] 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. [110] approach this problem using a subsumption model of technical debt based on a modified Value Based Software Engineering [111] 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 [103], is used to evaluate each item provided the

best savings per release (similar to that of Schmid [97,98,109]). Foganholi et al. [112] have implemented the TDMF as a tool connected with SonarQube™ allowing for both automated and manual identification and tracking of technical debt.

Decision support approaches for debt acquisition have been less forthcoming than for debt repayment. Nevertheless, Falessi et al. [113] are exploring current open problems concerning this topic as well as the required decision support constructs needed to address the problem. Ramasabba and Kemerer [114] developed an optimization approach utilizing multiple projections of a single codebase to evaluate decisions regarding both debt acquisition and repayment. Griffith et al. [115] conducted a simulation study of TD management strategies. The results of 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. [116,117].

As the technical debt landscape has evolved the research community's focus has moved from identifying what is technical debt, to the underlying issues surrounding these items. Specifically Falessi and Voegele [118] 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 the quality of software. This study found that classes with high numbers of rule violations also tend to be defect-prone. Additionally, Tufano et al. [119] conducted a large 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 normally introduced at the creation of 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. [120] 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.

Impact and Consequences

The impact of technical debt on engineering effort, project cost, and project quality is of utmost concern. An initial empirical enquiry conducted by Zazworka et. al. [106] shows that technical debt has a negative impact on software quality. Furthermore, Zazworka et al. [121] investigated prioritizing debts using a cost/benefit analysis approach. This study shows that technical debt negatively affects the correctness and maintainability of a product. Recently, Griffith et al. [12] 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 [5, 6], SonarQube™ [11], or Marinescu's [122] approaches to measuring technical debt principal and quality attributes in the QMOOD quality model [123].

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 [16] of TD item types. Zazworka et al. [124] 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. [125] which has further defined the technical debt landscape. This expanded landscape has been divided into 13 subtypes of technical debt each consisting of multiple indicators. The original landscape as proposed by Izurieta et al. [16] is now encompassed within the subtype of *Design Debt*. Fontana, Ferme, and Spinelli [105] state that although code smells are important

components of the technical debt landscape, certain identified debt items may not actually constitute debt. Instead they indicate that domain knowledge must be used as a filter in order to identify these misnomers and to ensure that an accurate indication of technical debt is provided.

Measurement

Lastly, there must be a means to measure technical debt and its associated properties in a way that is both meaningful to developers and to stakeholders alike. Seminal work by, Brown et. al. [100] identified the technical debt metrics of: principal, interest, and interest probability. Subsequently, Nugroho et al. [4] 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 [101, 124] have been proposed to index the effect that design flaws (e.g., code smells and modularity violations) have on technical debt. For example, Marinescu [126] 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 which can be aggregated over the size of the system. In another approach Nord et al. [8] develop a strong foundation for measuring the architectural technical debt based on the notion of prudent, deliberate, and intentional debt.

Letouzey [10] 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. Recently, Curtis et al. [5] proposed methods to estimate the principal and interest as well as the size, cost, and type of technical debt. Given these various approaches for the quantification of technical debt and the wide range of differences in values, Izurieta et al. [127] 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.

Previous research focus has been on the measurement of technical debt principal, but more recent research has turned towards measuring interest [128–131]. This shift in direction is due to research indicating that large principal values do not convey understanding of the effect that technical debt will have on a project [131]. Falessi and Reichel [131] have developed a tool to extract historical and defect data from repository and bug-tracking information. This tool then interfaces with SonarQube™ to display TD interest data to influence developer decisions. Chatzigeorgiou et al. [129] are developing an optimization technique which attempts to optimize the time line for repayment based on historical data and metrics.

Architectural Debt

Recent research has shown that a majority of technical debt stems from issues with the underlying architecture or with architectural decisions [132], hence interest in architecture technical debt (ATD) has increased. Given this, Martini et al. [116, 117] have conducted an industrial case study at 5 companies in order to understand the underlying causes of ATD. Based on the underlying causes they have developed two models of ATD management: A crisis model describing reactionary recovery when accumulation reaches a critical point, and the feature-release model which incorporates constant accumulation with recovery after release. Furthermore, Xiao [133] has developed a method of quantifying architectural debts. Xiao defines architectural debts as groups or clusters of files which incur higher maintenance costs due to an underlying architectural degradation. Recently, Izurieta et al. [134] have

proposed the notion of *Model Driven Technical Debt*, which is technical debt incurred during the model-to-code mapping process in model driven design. The goal of this work is to classify MDTD in order to provide the ability to reduce TD in the modeling phase in order to reduce TD in the resulting implementation and thereby increase the quality of the software product.

Design Pattern Evolution

Design patterns were widely introduced to the software engineering community by Gamma et al. [17]. Design patterns are an abstraction of solutions, forged in experience, to commonly recurring design problems. These patterns are a type of micro-architecture which are subject to both evolutionary issues and design decay, yet 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 [135–138].

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 [139–147] 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 are 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 [21, 144, 147]. This meta-model extends the UML™ meta-model [148] which allows the instances to be visually described and constrained using the Object Constraint Language (OCL) [149]. 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 [150] initially proposed a method for evaluating the structural conformance of a pattern instance to the specification. This proposal was followed by Kim and Shen's [151, 152] divide-and-conquer approach. Based on this approach Strasser et al. [153] developed a tool to calculate a score for the conformance rating of a design pattern instance given its RBML specification. Recently, Lu and Kim [154, 155] have developed an approach to validate conformance of behavior and sequence diagrams of pattern instances. Another extension by Kim and Whittle [156] to generate models from existing patterns.

Research Gaps

The management of design pattern decay forms an important component, in the management of software aging and technical debt, and 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 [18–20, 22, 23].

- Technical Debt – Current research has looked into how grime plays a part in the technical debt landscape [16]. The effect of grime on the technical debt value of a software product and pattern instances has only been studied for modular grime [22].

- Relationships – The notion that different subtypes of grime can be interrelated or that subtypes of grime and design defects types can be related is another area of study still left untouched.

- Automation – The ability to detect grime is a manual and time-consuming process. In part, this is 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. Of these studies only a very small selection of systems have been studied. 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.

RESEARCH OBJECTIVES

The overarching goal of this research is to examine and expand our knowledge concerning the relationship between software product quality and technical debt. The focus of this research is narrowed to design pattern grime and its relationship to software product quality and technical debt using a measurement driven approach.

To guide the research towards a solution we employ the Goal Question Metric (GQM) method [157]. This method requires the problem to be divided into a set of research goals (RG). Each goal is then subdivided into a set of research questions (RQ). Finally, each question leads to the development of a set of metrics (M) which be used in experiments or case studies. The following is the GQM breakdown for the above problem statement:

RG1: Analyze design patterns to elaborate the complete taxonomy of class and organizational grime (see Chapter 4).

RQ1.1: What are the types of class grime?

RQ1.2: What are the types of organizational grime?

RG2: Analyze pattern instances for the purpose of detecting grime with respect to precision and recall, from the perspective of a software system, in the context of open source software projects.

RQ2.1: What is the precision [158] of the grime detection algorithms for each type of grime?

RQ2.2: What is the recall [158] of the grime detection algorithms for each type of grime?

M2.1: *Precision* – The number of true positives divided by the sum of the true positives and false positives.

M2.2: *Recall* – The number of true positives divided by the sum of the true positives and false negatives.

RG3: Analyze pattern instances for the purpose of tracking grime buildup with respect to the amount of grime from the perspective of several system versions in the context of open source software projects.

RQ3.1: How does grime change over time?

RQ3.2: Which type of grime is more likely to occur as pattern instances evolve?

RQ3.3: What relationships exist between pattern instance evolution and grime growth?

RQ3.4: Which patterns are more susceptible to grime accumulation?

RQ3.5: Which pattern families (groups of patterns based on common functionality or structure: i.e., in the 23 patterns defined in Gamma et al. [17] are divided into three groups: *Creational Patterns*, *Structural Patterns*, and *Behavioral Patterns*) are more susceptible to grime accumulation?

M3.1: *Grime Size* – The amount of grime accumulated within a pattern instance.

M3.2: *Grime Severity* – The grime size of a pattern instance normalized by the size of the pattern instance.

M3.3: *Grime Growth* – The rate of grime growth from initial identification up to the current version of the software.

M3.4: *Pattern Size* – The size of the pattern instance measured using the number of classes weighted by a normalized number of methods.

M3.5: *Grime Susceptibility* – A measure of the likelihood of a pattern instance to accumulate a type of grime given the pattern's type.

RG4: Analyze pattern instances for the purpose of identifying intra- and inter-relationships with respect to grime subtypes from the perspective of a grime taxonomy in the context of open source projects.

RQ4.1: What are the relationships between modular grime types?

RQ4.2: What are the relationships between class grime types?

RQ4.3: What are the relationships between organizational grime types?

RQ4.4: What are the relationships between modular, class, and organizational grime types?

M4.1: *Plain Support* – Grime type A supports grime type B, if the presence of A indicates a high likelihood of the presence of B [63].

M4.2: *Mutual Support* – Grime type A and B mutually support each other, if the presence of A indicates a high likelihood of the presence of B and the presence of B indicates a high likelihood of the presence of A [63].

M4.3: *Aggregate Support* – The generalization of plain support and rejection, in that there are two sets of grime types $\mathcal{A}$ and $\mathcal{B}$ where the set $\mathcal{A}$ is supporting grime types and set $\mathcal{B}$ is the rejecting grime types. The combination of the two sets indicates with high likelihood of the presence of grime types in a thrid set, $\mathcal{C}$ [63].

M4.4: *Transitive Support* – Grime type A transitively supports grime type C, if there is a relationship such as, grime type A supports grime type B which supports grime type C [63].

M4.5: *Rejection* – Grime type A rejects grime type B, if the presence of A indicates a very low likelihood of the presence of B [63].

M4.6: *Inclusion* – Grime type A includes grime type B, if A is a special case of B [63].

RG5: Analyze pattern instances afflicted with grime for the purpose of evaluation with respect to the Quamoco Quality Model, from the perspective of design pattern instances, in the context of open source software projects.

RQ5.1: How does each type of grime affect *functional suitability*, as measured by the Quamoco [159] quality model?

RQ5.2: How does each type of grime affect *maintainability*, as measured by the Quamoco [159] quality model?

RQ5.3: How does each type of grime affect *performance efficiency*, as measured by the Quamoco [159] quality model?

RQ5.4: How does each type of grime affect *reliability*, as measured by the Quamoco [159] quality model?

RQ5.5: How does each type of grime affect *security*, as measured by the Quamoco [159] quality model?

M5.1: *Functional Suitability* – the degree to which a software product provides functions that meet stated or implied needs when used under a specified set of conditions [93]. This will be measured using an implementation of the Quamoco [159] quality model.

M5.2: *Maintainability* – the degree of effectiveness and efficiency with which a software product can be modified by the intended maintainers [93]. This will be measured using an implementation of the Quamoco [159] quality model.

M5.3: *Performance Efficiency* – the degree of performance efficiency relative to the amount of resources used under a stated set of conditions [93]. This will be measured using an implementation of the Quamoco [159] quality model.

M5.4: *Reliability* – the degree to which a software product performs specified functions under specified conditions for a specified period of time [93]. This will be measured using an implementation of the Quamoco [159] quality model.

M5.5: *Security* – the degree to which a product or system protects information and data so that persons or other products or systems have the degreee of data access appropriate to their types and levels of authorization [93]. This will be measured using an implementation of the Quamoco [159] quality model.

RG6: Analyze pattern instances for the purpose of evaluating technical debt with respect to grime from the perspective of several system versions in the context of open source projects.

RQ6.1: What effect does each type of grime have on technical debt of an entire software system?

RQ6.2: How can we index the technical debt associated with the grime buildup in a pattern instance?

M6.1: *Technical Debt Principal* – The cost (in effort) associated with the current amount of refactoring required to bring a software system to a desired level of maintainability. This will be measured using several methods proposed in the literature [3–11].

Summary of the Approach

Initially we will begin by extendeding the taxonomies for class and organizational grime (see Chapter 4). Next, using the design pattern taxonomies define models of each type of grime. For each type of grime we will use these models to define: an injection strategy (see Chapter 5) and a detection strategy.

The injection strategies will be used in experiments (see Chapter 6 Section 6) to: 1.) evaluate the effect on quality attributes, 2.) evaluate the effect on technical debt, and 3.) evaluate the effectiveness of detection strategies. Furthermore, detection strategies will be used to conduct a series of case studies on open source software (see Chapter 6 Section 6) to: 1.) identify quality and technical debt relationships associated with design pattern grime, 2.) identify live instances of design pattern grime, and 3.) identify relationships between grime types. Finally, conduct an industry case study to validate the conclusions found on open-source software applies in industry software.

Research Contributions

Given the gaps identified in Chapter 2 Section 2 this research proposes the following contributions.

1. A formal benchmarking approach which injects disharmonies and pattern instances into software systems in order to provide both a "gold standard" to compare detection algorithms against as well as providing the basis upon

which experimentation can be conducted. Initial work in this area has started including the definition of a formal metamodel (see Chapter 5).

2. An expanded taxonomy of design pattern grime, which is pretty much complete (see Chapter 4).

3. An approach to automate the detection of design pattern grime, culminating in a tool which detects each type of grime. The tool is currently in the design phase.

4. A categorization of relationships between design pattern grime and design patterns types. Currently awaiting experimentation.

5. Empirical confirmation of the existence of the known subtypes of design pattern grime. Currently awaiting the development of the detection tool.

6. Analysis of the impact of grime on software product quality measures. The quality measurement tool has been developed, we are awaiting completion of the grime detection tool.

7. Analysis of the impact of grime on the technical debt value of a given software product. Currently awaiting both the development of the grime detection tool as well as the development of a technical debt measurment tool.

DESIGN PATTERN GRIME TAXONOMY

Introduction

This chapter describes the extensions to grime taxonomy by expanding the class and organizational leaves of the original grime taxonomy defined by Izurieta [160]. We first elaborate on the approach used to define the taxonomies. We then will define the extended taxonomies for class and organizational grime.

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 [39]. In reviewing their work as well as other publications defining disharmony types such as code smells [13] and modularity violations [35], we have developed an underlying process in order to further elaborate the class and organizational grime types, as follows:

1. Identify the software entities of concern, such as: classes, packages, or relationships.
2. Identify the design principles or practices that affect these entities, which have not already been elaborated upon by existing disharmonies.
3. Identify the measurable properties of these principles or their components to develop the levels of the taxonomy.
4. Select metrics which measure the identified properties.
5. Formally define 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.

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. This implies that class grime is essentially 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 [13].
- The Single Responsibility Principle (SRP) – a class should only have responsibility for a single part of the functionality of the software, and this responsibility should be fully encapsulated within the class [161].
- The Interface Segregation Principle (ISP) – no client should depend on those methods it does not use [161].
- High class cohesion – the responsibilities of the methods within a class should be highly related and support the responsibility of the class [162].

Each of these principles speaks to the cohesion of a class. Where a highly cohesive class is one in which it's member fields and methods are designed to work together to address a single major responsibility of the class. Using cohesion as the fundamental property we have divided class grime into eight specific subtypes as depicted in Figure 4.1. This division is further explained in the following sections.

Class Cohesion

Cohesion is used to describe how well constructed a class is [163]. The higher the cohesion of a class the closer aligned its internal components are towards a

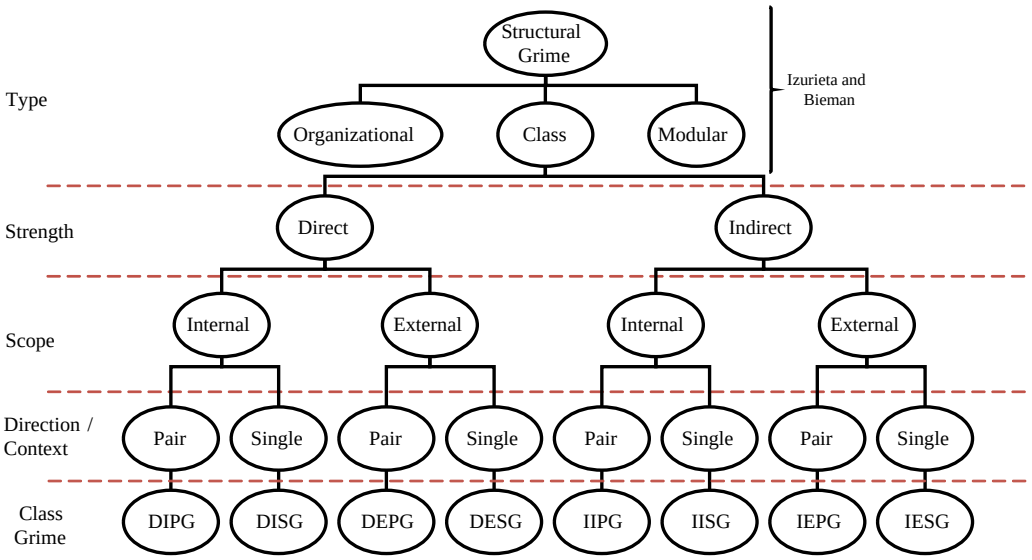

Figure 4.1: The extended class grime taxonomy.

common goal. In design pattern realizations, the classes should represent individual responsibilities of the pattern and if the specification is implemented correctly each class should have high cohesion, thus cohesion provides a basis to determine whether a design pattern realizations classes have been afflicted with class grime.

Strength Strength is indicated by the method in which attributes are locally accessed by a class methods. The method of access can be either direct (attributes are directly accessed by methods) or indirect (attribute access through the use of an accessor/mutator methods). Each of these can be seen in Figure 4.1, where the unbroken lines between attributes (rectangles) and methods (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, scope can either be internal or external. Internal refers to when an attribute of the class is accessed by a local method (or local method pair, depending on context) defined by the pattern specification. External refers to when an attribute is accessed by at least one local method (or local method pair) not defined by the pattern specification. In Figure 4.1, 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. This provides a means to distinguish between identification of attributes (internal) or methods (external) which are obscuring the pattern implementation, through a reduction in 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 [163]. 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) [164] 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) [165] metric which measures the cohesion of a class by looking at how individual methods use attributes, and it can handle both indirect and direct attribute use.

Class Grime Example

In Figure 4.2 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). This is a case of DISG due to the fact that there is a method, $m1$, allowed by the pattern specification that directly uses an attribute, $a1$, but no other method uses that attribute. This 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.

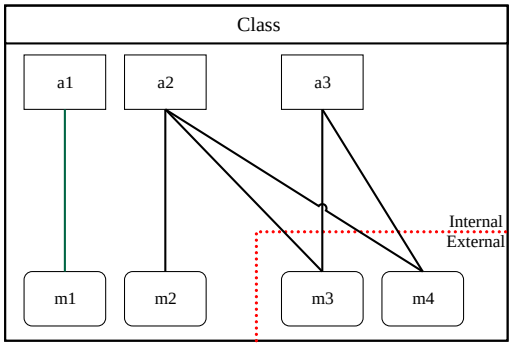

Figure 4.2: Example of DISG.

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 [161].

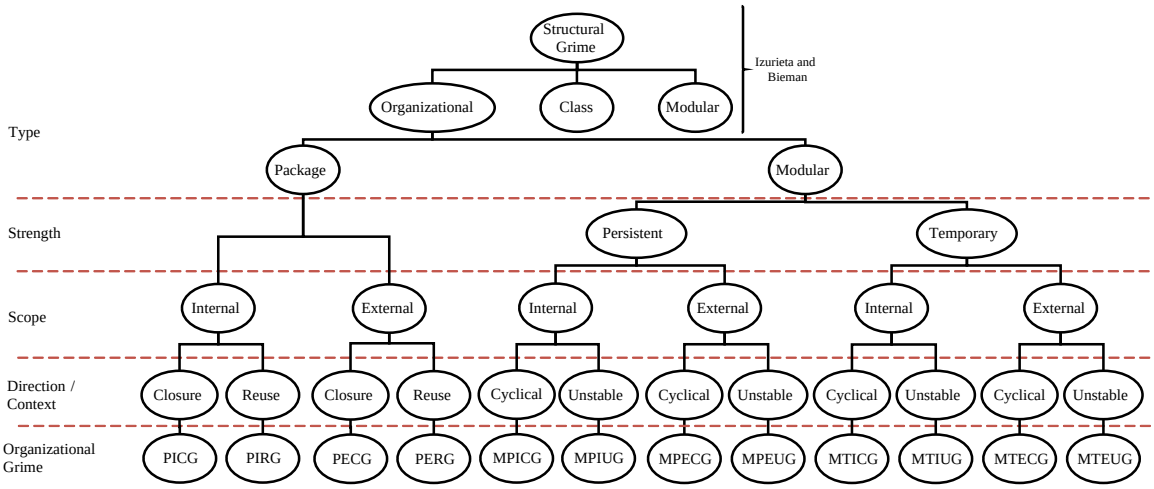

Figure 4.3: Organizational grime taxonomy.

- The Stable Dependencies Principle (SDP) – Depend in the direction of stability [161].
- The Stable Abstractions Principle (SAP) – Abstractness should increase with stability [161].
- The Common Closure Principle (CCP) – Classes in a package should be closed to the same kinds of changes [161].
- The Common Reuse Principle (CRP) – Classes in the same package should be reused together [161].

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 4.3. This division is further explained in the following sections.

Package Coupling

Package coupling is used to develop the *modular* subtype of organizational grime, as see in Figure 4.3. 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. 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 are relationships between packages that are caused by external classes interacting with pattern classes across packages. The final properties is the direction/context property. 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. Together these concepts will be used to form the modular branch of organizational grime.

Package Cohesion

Package cohesion is used to develop the *package* subtype of organizational grime, as seen in Figure 4.3. Here we consider only the scope and context properties. 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 it's scope is internal, otherwise it is external. 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

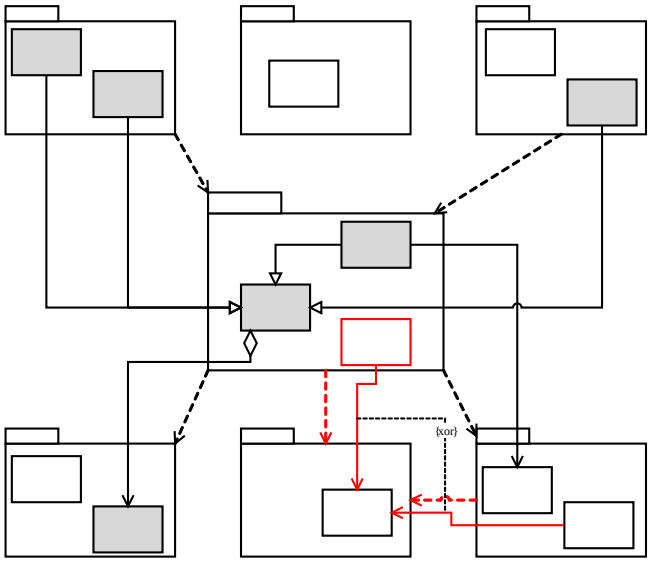

Figure 4.4: Example of PECG.

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. Together threse concepts will be used to form the package branch of organizational grime.

Organizational Grime Example

In Figure 4.4 we can see an example of Package External Closure Grime (PECG). In this diagram those classes which are a part of the pattern a marked as grey rectangles, classes not part of the pattern are marked as white rectangles, dependencies between packages are marked using a dashed line with an open arrow head pointing in the direction of the dependency, and 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 from the pattern) interacting with a non-pattern class but increasing the nubmer of packages reachable from pattern packages.

Conclusion

This chapter presented the enhanced design pattern grime taxonomy. Herein we detailed the exact methodology by which the taxonomies of class and organizational grime were developed. Furthermore, we connected these taxonomies to the underlying software engineering principles they are codifying as well as the metrics that will lead to the definition of their detection strategies. The development of these taxonomies along with the existing modular grime taxonomy will lead directly into the design of both injection and detection strategies for these different grime types.

DISHARMONY INJECTION

Introduction

Currently, design disharmony research is limited by manual verfication of disharmony detection techniques. This limitation is imposed by the necessity to identify live instances of any disharmony we wish to study. Such limitations have slowed the progress in evaluating the effects design disharmonies have on quality and technical debt. For those researchers who do not wish to perform these manual verfications, there are limited datasets that provide benchmarks for evaluating detection techniques. Unfortunately, these datasets are limited to a small subset of design disharmonies. In this chapter, we propose a framework to help eliminate these limitations and improve the capability for experimentation.

Disharmony Injection

The framework proposed is called *disharmony injection*. Basically, disharmony injection is a transformation which augments a software system to include the entities under study. Formally, we define disharmony injection as follows:

Definition: *Disharmony Injection.* Let $\mathcal{S}$ be a software system, $\mathcal{T}$ a transformation, $\mathcal{C}_{\mathcal{T}}$ a set of conditions on $\mathcal{T}$, and E the expected outcome. Disharmony injection modifies $\mathcal{S}$ using $\mathcal{T}$ to achieve the outcome E. This results in a modified system, $\mathcal{S}^*$, such that $\mathcal{S}^* = \mathcal{T}(\mathcal{S})$ if $\forall c \in \mathcal{C}_{\mathcal{T}}$, $c(\mathcal{S}) \wedge E(\mathcal{S}^*)$.

Injection Strategy Metamodel

This section describes the injection strategy concept. An injection strategy is a transformation or set of transformations designed to modify an existing system to include a specific entity or issue. Currently we are using a metamodeling approach to

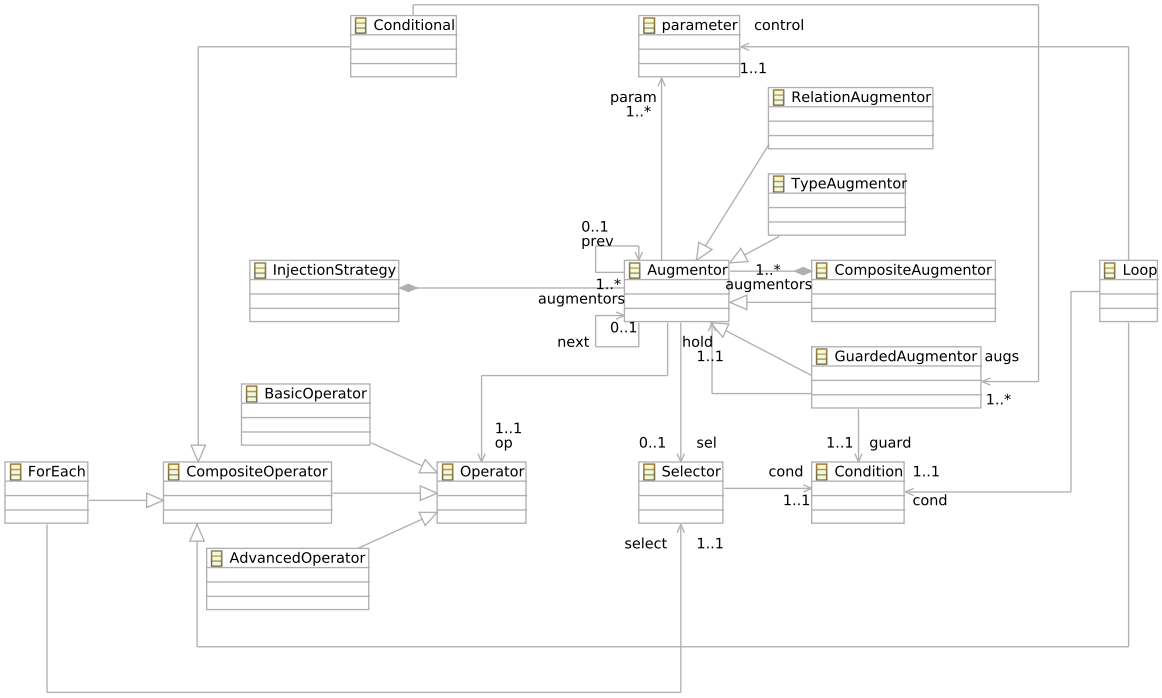

Figure 5.1: Disharmony Injection meta-model.

develop the injection strategy technology, as depicted in Figure 5.1. This metamodel will serve as the foundation for the development of a domain specific language for automating the injection process. An example use of this technolog is the injection of a design pattern instance into an existing software system. The following definitions describe the entities in this metamodel.

Definition: *Augmentor* – One of the three basic components of an Injection Strategy. Augmentors, perform the work of injecting new entities into the structure of the software. There are four types of augmentors: TypeAugmentors, RelationAugmentors, GuardedAugmentors, and CompositeAugmentors. Each Augmentor has a link to the next and previous augmentors in the Injection Strategy. Augmentors use

an associated operator to control their behavior. Furthermore, augmentors can be refined via parameters provided to elaborate the details of their processing. Finally, for augmentors which work with existing entities in the system, a selector can be provided to identify the item the augmentor works with.

Definition: *TypeAugmentor* – TypeAugmentors are designed to modify the types of a system or the contents of types in a system.

Definition: *RelationAugmentor* – An augmentor designed to modify the relations between types within a software system.

Definition: *CompositeAugmentor* – An augmentor designed to be composed of other augmetors in order enhance the capabilities of an injection strategy.

Definition: *GuardedAugmentor* – A composite augmentor which adds a condition or set of conditions that must be true in order to execute the contained augmentors.

Definition: *Selector* – A operand used by augmentors in order to specify the system entity which they will modify. A selector uses a set of conditional expressions to filter the system to find the specific entities to modify.

Definition: *Operator* – Represents the operator which controls what an augmentor does when modifying the system. There are three defined types of operators: Basic, Advanced, and Composite.

Definition: *BasicOperator* – Represents the basic augmentation operators. These include *Adding/Creating* entities in the system, *Deleting/Removing* entities from the

system, *Moving* entities from one place in the system to another, and *Changing* the properities associated with an entity (such as their name).

Definition: *CompositeOperator* – Represent a means to develop composite augmentation operations, or to control the flow of operations such as with conditional expressions or repeated operations using loops.

Definition: *Loop* – A composite operation which relies on a series of parameters to control processing. As a composite operation, it contains a set of operations associated with a given augmentator.

Definition: *ForEach* – A special type of loop which simply takes a set of entities to augment and applies the given set of augmentation operators to this set of system entities. The set is specified using a selector.

Definition: *Conditional* – A form of composite operator which allows the conditional processing of specified composite operators based on the evaluation of associated guard conditions.

Definition: *Condition* – A boolean logic condition or composite expression which evaluates to true or false.

Definition: *Parameter* – A parameter which augments the operator or augmentor to provide more capabilities in processing.

The Injection Process

Design defects and design patterns can both be injected into a system using the following approach, depicted in Figure 5.2. Initially, an Eclipse Modeling Framework

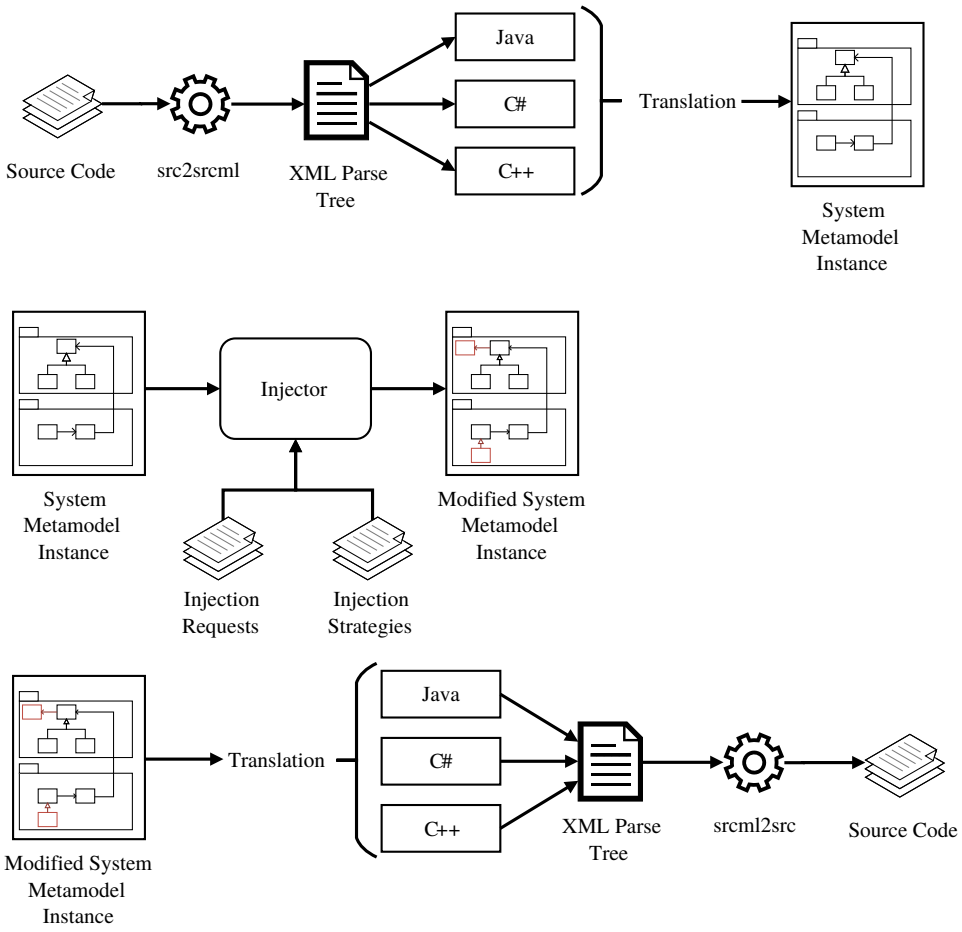

Figure 5.2: The disharmony injection process.

(EMF) model of the system must be extracted. This is performed as follows: First, we execute the srcML *src2srcml* tool to extract an XML representation of the parse tree for that system. We then use a language specific model builder to extract a model instance of the system from the XML, which uses a combination of XPath queries and DOM traversals. Once the model is extracted an injection strategy is applied and changes are noted. The record of changes is used to modify the exiting XML parse tree. The parse tree is modified using language specific transformations which ensure correctness of the modifications. Once the modifications to the parse tree are complete, we use the srcML *srcml2src* tool to generate the source code.

We note that this process is similar to that of Dale [22]. Dale used Java™ bytecode injection in order to inject modular grime into an existing system. The novelty of the process depicted here is in the introduction of artifacts such as code smells, antipatterns, design pattern grime, and design patterns using defined and validated injection strategies which control the injection process. This process also is based on the modification of source code using a model driven approach which is language independent. Finally, the ability to generate source code escapes the problem of simulation when dealing with this type of approach and it facilitates the ability to inspect the generated code to verify the production of these entities, a process that bytecode injection prohibits.

Application to Experimentation

Injection strategies can be devised to inject any number or type of software entities. When injection strategies are combined with proper experimental design and parameterization, then the various effects of design disharmonies on software systems can be easily evaluated. The use of injection allows for controlled creation of design disharmonies. Furthermore injection allows for the randomization in assignment and selection of treatment groups, thus providing a means to evaluate causal relationships. This eliminates the downsides of using live instances, such as manual identification and validation.

EMPIRICAL METHODS

Introduction

This chapter proposes the empirical body of work for the dissertation. Herein are the proposed experimental and case study designs that will be used to answer the research questions identified in Chapter 3. Each research question will be addressed through a series of experiments and case studies described in the following sections.

General Process

The general method proposed for both experiments and case studies is depicted in Figure 6.1 and proceeds as follows (as numbered in the figure). First, Using the research goals as a guide an experimental design or case study design is selected. Next, we will conduct a sample size analysis, using R statistical software, to determine the sample size and/or number of replications necessary to achieve the desired statistical power for the analysis methods selected. Simultaneously, we select from either a set of clean design pattern instances (marked "Instance Space" in Figure 6.1) or software projects (marked "Project Space" in Figure 6.1) (see Section 6). Once selected, each design pattern instance or software project is assigned in a manner according to the design selected in step 1. Upon completing the assignment phase the experiment or case study protocol is executed (see Section 6 for more details). Next, data analysis, using R statistical software, is conducted as described in the "Analysis Procedure" subsections of each experiment and case study description. Simultaneously, a power analysis is conducted to verify that the type I error rate is minimized. Finally, the results of the analysis procedures are reported.

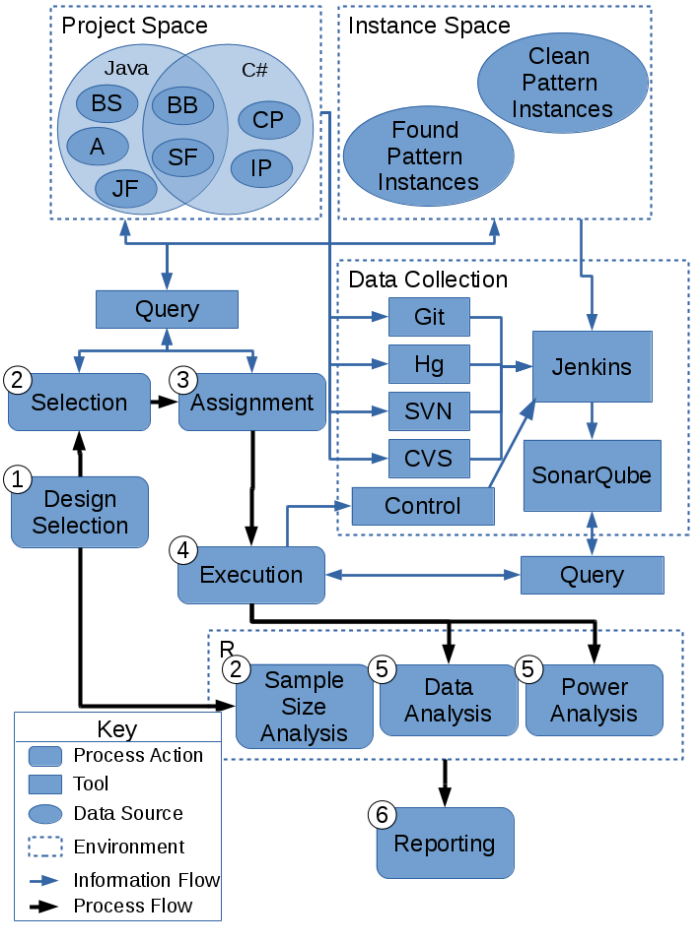

Figure 6.1: The general experimental process.

Datasets

There are two sources of data for this research. The first source is a set of design pattern instances which are generated and known to be free of disharmonies. The second is a collection of design pattern instances extracted, using the Percerons Client tool [166], from open source projects hosted at the following project hosting sites: SourceForge (SF) [1], JavaForge (JF)[2], Apache Software Foundation (A)[3], BitBucket

[1] http://www.sourceforge.net
[2] http://www.javaforge.net
[3] http://www.apache.com

(BB)[4], CodePlex (CP)[5], and BeanStalk (BS)[6] of from our Industry Partners (IP). The other data set is composed of projects selected from the aforementioned hosting sites. In the case of software projects, each project's repository URI must first be is extracted from it's host site. The repository URI is then used, along with other project meta-data, to construct a build job in Jenkins CI, a software project build automation tool used here to automate protocol execution.

Execution

Jenkins will be used to automatically download projects from hosted repositories, execute the build process, and call SonarQube™ to perform the required measurements. SonarQube™ uses a collection of plugins to analyze software quality and collect required measures. Currently, several plugins are in development which will provide the Quamoco [159] quality analysis, design pattern grime analysis, and technical debt measures.

The experimental and case study protocols will be executed using a tool designed to control Jenkins CI build jobs. This control program will determine the run order of each study, will extract and collect the data once the SonarQube™ analysis is complete and will execute developed R scripts to analyze the data. Furthermore, in experiments where disharmony injection is required, this control program will also be used to execute the injection strategies.

[4] `http://www.bitbucket.com`
[5] `http://www.codeplex.com`
[6] `http://www.beanstalk.com`

Experiments

Grime Detection

Purpose The purpose of these experiments is to evaluate detection strategies for each grime type, using the metrics precision and recall in order to answer RQ 2.1 and 2.2. Our goal is to compare, at the detection strategy level, which detection strategies (manually defined or automatically generated) have the better precision and recall. In these experiments we intend to test the following hypotheses:

$H_{1,0}$**:** There is no difference in the recall of the automated and manual detection strategies.

$H_{1,1}$**:** The recall of the automated detection strategies is greater than that of the manual detection strategies for at least one grime type gt_i.

$H_{2,0}$**:** There is no difference in the precision of the automated and manual detection strategies for grime type gt_i.

$H_{2,1}$**:** The precision of the automated detection strategies is greater than that of the manual detection strategies for at least one grime type gt_i.

Experimental Design A split-plot design with a randomized complete block design in the whole plots has been selected to compare manually defined detection strategies against those designed using a genetic programming approach. The selected design will consider the different detection strategies design types (manual or genetic programming) to be the whole plots. Nested within these is a randomized complete block design. The inner design will block grime injection types by pattern type. That is, for each of the 23 design patterns defined in Gamma et al. [17], a randomly selected instance of that pattern will be randomly assigned to a grime injection strategy for

each type of grime. The comparison will be based on the precision and recall of algorithms for each grime subtype.

Data Collection Procedure As described in Chapter 2, Kessentini et al. [54,55,66] have developed a method to generate detection strategies using genetic programming techniques. This experiment will use a similar method to generate design pattern grime detection strategies. The process for validating and comparing these strategies with manually designed ones is as follows:

1. generate a dataset of clean design pattern implementations for each type of design pattern defined in Gamma et al. [17].
2. For each grime type, for each pattern type:
   (a) generate a dataset of 100 possible instances of design pattern grime.
   (b) generate the detection strategies attempting to maximize the precision and recall.
3. Utilize 10-fold cross-validation to validate the generated detection strategies and to prevent over-fitting the data.
4. Test the results of validation on a separate generated dataset.

Analysis Procedure In order to test the first two sets of hypotheses we will conduct an ANOVA [167], assuming the assumptions are met, at the 0.05 α threshold. If the assumptions cannot be met we will attempt to transform the data if appropriate or resort to the use of appropriate non-parametric methods. The remaining hypotheses will be tested using a Tukey's HSD [168] multiple comparison procedure, at the grime subtype level.

Grime Effects on Quality

Purpose The purpose of these experiments is to evaluate the effects that design pattern grime has on the quality of a design pattern, in order to answer RQ 5.1–5.5. The following hypotheses (for each quality attribute, qa_i, of Functional Suitability, Maintainability, Reliability, Security, and Performance Efficiency) will be tested:

$H_{1,0}$**:** There is no change in mean qa_i due to grime.

$H_{2,0}$**:** There is no difference in mean qa_i between class and organizational grime.

$H_{3,0}$**:** There is no difference in mean qa_i between class and modular grime

$H_{4,0}$**:** There is no difference in mean qa_i between modular and organizational grime

$H_{5,0}$**:** There is no difference in the change in qa_i between indirect and direct class grime types.

$H_{6,0}$**:** There is no difference in the change in qa_i between internal and external class grime types.

$H_{7,0}$**:** There is no difference in the change in qa_i between single and pair class grime types.

$H_{8,0}$**:** There is no difference in the change in qa_i between persistent and temporary modular grime types.

$H_{9,0}$**:** There is no difference in the change in qa_i between internal and external modular grime types.

$H_{10,0}$**:** There is no difference in the change in qa_i between afferent and efferent modular grime types.

$H_{11,0}$**:** There is no difference in the change in qa_i between package and modular organizational grime types.

$H_{12,0}$**:** There is no difference in the change in qa_i between internal and external organizational grime types.

$H_{13,0}$**:** There is no difference in the change in qa_i between closure and reuse organizational grime types.

$H_{14,0}$**:** There is no difference in the change in qa_i between cyclical and unstable organizational grime types.

Experimental Design The selected experimental design will be a randomized complete block design. Here the blocking variable is the design pattern type and the treatments are the subtypes of class, modular, or organizational grime to be injected. The response variables measured are the following five quality attributes measured using an implementation of the Quamoco quality model: *Functional Suitability*, *Maintainability*, *Reliability*, *Performance Efficiency*, and *Security*.

Data Collection Procedure Using the general process this experiment will utilize pattern instances in both Java™ and C# languages. For each pattern type, n instances will be selected randomly for both Java™and C#, where n is the number of grime subtypes. The instance will then be randomly assigned to a grime subtype for injection.

Analysis Procedure Once the experimental control program has completed and all reported results are collected the analysis will be commenced. A MANOVA [169] analysis will be used to evaluate $H_{1,0}$. The remaining hypotheses will be evaluated using a combination of contrasts and multiple comparison methods for multivariate data.

Grime Effects on Technical Debt

Purpose The purpose of these experiments is to evaluate the effects that design pattern grime has on the technical debt associated with a design pattern, in order to answer question RQ 6.1. The following hypotheses are to be tested:

$H_{1,0}$**:** There is no change in mean technical debt due to grime.

$H_{2,0}$**:** There is no difference in mean technical debt between class and organizational grime.

$H_{3,0}$**:** There is no difference in mean technical debt between class and modular grime

$H_{4,0}$**:** There is no difference in mean technical debt between modular and organizational grime

$H_{5,0}$**:** There is no difference in the change in technical debt between indirect and direct class grime types.

$H_{6,0}$**:** There is no difference in the change in technical debt between internal and external class grime types.

$H_{7,0}$**:** There is no difference in the change in technical debt between single and pair class grime types.

$H_{8,0}$**:** There is no difference in the change in technical debt between persistent and temporary modular grime types.

$H_{9,0}$**:** There is no difference in the change in technical debt between internal and external modular grime types.

$H_{10,0}$**:** There is no difference in the change in technical debt between afferent and efferent modular grime types.

$H_{11,0}$**:** There is no difference in the change in technical debt between package and modular organizational grime types.

$H_{12,0}$**:** There is no difference in the change in technical debt between internal and external organizational grime types.

$H_{13,0}$**:** There is no difference in the change in technical debt between closure and reuse organizational grime types.

$H_{14,0}$**:** There is no difference in the change in technical debt between cyclical and unstable organizational grime types.

Experimental Design The selected experimental design will be a randomized complete block design. Here the blocking variable is the design pattern type and the treatments are the subtypes of class, modular, or organizational grime to be injected. The response variables measured are the following five measures of technical debt principal: the SQALE method [9, 10], the CAST method [5, 6], Nugroho et al's [4] method, Nord et al.'s [8] method, and Chin et al.'s [7] method.

Data Collection Procedure Using the general process this experiment will utilize pattern instances in both Java™ and C# languages. For each pattern type, n instances will be selected randomly for both Java™and C#, where n is the number of grime subtypes. The instance will then be randomly assigned to a grime subtype for injection.

Analysis Procedure Once the experimental control program has completed and all reported results are collected the analysis will be commenced. A MANOVA [169] analysis will be used to evaluate $H_{1,0}$. The remaining hypotheses will be evaluated using a combination of contrasts and multiple comparison methods for multivariate data.

Case Studies

Grime Buildup

Purpose The purpose of this case study is to understand how grime develops or evolves in patterns in order to answer research questions RQ 3.1–3.5 described in Chapter 3. Understanding grime evolution will lead to improved detection strategies, improved classification, and an improved understanding of the importance of these disharmony types. We intend to test the following hypotheses:

$H_{1,0}$**:** Across versions a design pattern instances grime buildup remains constant.

$H_{2,0}$**:** For any given design pattern type, the likelihood of any specific type of grime occurring is the same.

$H_{3,0}$**:** There is no relationship between design pattern growth and grime build up.

$H_{4,0}$**:** The likelihood of developing grime is the same for all design pattern types.

$H_{5,0}$**:** The likelihood of developing grime is the same for all design pattern families.

Study Design The subjects of this study are the design pattern instances within the selected systems. System selection criteria is a combination of size (as measured using lines of code) and language (either Java™ or C#). Furthermore, this case study will only consider open source systems. Finally, each pattern instance selected for study must exist across multiple versions of the containing software.

Data Collection Procedure Using the general process this experiment will utilize both Java™ and C# open source projects. Pattern instances will then be extracted using the Percerons client tool, as stated in Section 6.

Analysis Procedure The first hypothesis, $H_{1,0}$, will be tested using an ARIMA analysis [170] to determine if the mean grime buildup remains the same regardless of the change in version, while controlling for change in software size. The remaining hypotheses will be tested using the following procedure. First, selected software systems will be scanned for existing design pattern instance. The found pattern instances will be grouped first by pattern type then by pattern family. Grime detection strategies will be executed across all design pattern instances. Grime counts are then measured for both pattern types and families. Next, an empirical distribution will be fitted to the grime count data in order to find the likelihood of grime development in patterns and pattern families. An ANOVA [167] analysis will be conducted to evaluate hypotheses $H_{2,0}$ and $H_{3,0}$. Based on the results from the test for $H_{2,0}$ further contrast analyses will be conducted to evaluate hypotheses $H_{4,0}$ and $H_{5,0}$.

Grime Relationships

Purpose The purpose of this case study is to identify relationships between different grime types in order to answer research questions RQ 4.1–4.4 described in Chapter 3. The identification of these relationships will help better define detection strategies as well as to better classify these disharmony types. In this case study We will test the following hypotheses:

$H_{1,0}$: There is no positive relationship between any pair of different grime subtypes (plain support).

$H_{2,0}$: There is no negative relationship between any pair of different grime subtypes (rejection).

$H_{3,0}$**:** There is no positive relationship between a pair (A, B) and a pair (B, A) of different grime subtypes (mutual support).

Study Design The subjects of this study are the design pattern instances within the selected systems. System selection criteria is a combination of size (as measured using lines of code) and language (either Java™ or C#). Furthermore, this case study will only consider open source systems.

Data Collection Procedure Using the general process this experiment will utilize both Java™ and C# open source projects. Pattern instances will then be extracted using the Percerons client tool, as stated above in Section 6.

Analysis Procedure Plain support, mutual support and rejection relationships will be evaluated using Kendall's τ [171] non-parametric measure of association between all pairs of grime subtypes. Transitive support will be evaluated by analysis of the results from the plain support and mutual support findings. Similarly, aggregate support will be evaluated by based on a combination of clustering analysis (initially investigating k-means, hierarchical, and spectral clustering methods) and indications of plain support, mutual support and rejection findings. Finally, inclusion will be based on a combination of the review of the taxonomy definitions in conjunction with the findings from the other relationships.

Industry Case Study

Purpose Each of the previous experiments and case studies involve the evaluation of generated or open source software. Furthermore, the data collection procedure will produce a process along with a dashboard allowing for continuous quality and technical debt monitoring. It is imperative to understand the utility, within the context of the software development lifecycle, that this process and its associated

dashboard provides in an industry setting. Given this, we are currently working with an industry partner developing a new architecture combining several related software projects into a single software product line. One of the goals of this new product line is to have a high level of quality [93] while meeting security requirements based on the Risk Management Framework (RMF) specification [172]. Therefore, we will be conducting a longitudinal case study of several of their projects, to evaluate if the developers (without intervention) are using this process and its associated dashboard to influence decisions concerning the evolution of the software projects. The goal of this study is to compare the differences in the historical quality and security evolution prior to using the process, with the evolution of the project after implementation of the feedback process. This case study will evaluate the following hypotheses:

$H_{1,0}$**:** There is no trend in each quality subcharacteristic.

$H_{2,0}$**:** There is no trend in each RMF subcharacteristic.

$H_{3,0}$**:** The mean values for quality subcharacteristics remained unchanged over time.

$H_{4,0}$**:** The mean values for RMF security subcharacteristics remained unchanged over time.

Study Design The software of concern at our industry partner is currently being migrated from a series of similar products to a product-line architecture solution. The goal is to centralize each of the similar products under a single database, implementation language, and common domain model. Thus, our case selection becomes the different major components of this new architecture.

Data Collection Procedure Using the general process this experiment will utilize C# projects provided by an industry partner. Pattern instances will then be extracted using the Percerons client tool [166], as stated above in Section 6.

Analysis Procedure For each time period, the hypotheses will be tested as follows. For both, $H_{1,0}$ and $H_{2,0}$ will be tested using a generalized least squares approach to provide the 95% confidence interval for the trends. For both $H_{3,0}$ and $H_{4,0}$ will be evaluated by first fitting an ARIMA model [170] and then evaluating the Given the evaluation the time period results will then be compared to evaluate the effect of adding a quality/RMF automated feedback process into the software development life-cycle.

THREATS TO VALIDITY

This chapter discusses the potential threats to the validity of the experiments and case studies detailed in Chapter 6. Specifically, we focus on threats to conclusion validity, internal validity, construct validity, content validity, external validity, and reliability [173] [174] [175] [176] [177].

Conclusion Validity

Conclusion validity is concerned with establishing statistical significance between the independent variables and the dependent variables. In order to ensure that there are no threats to conclusion validity we will be taking the following measures. In the experiments, we will verify that the assumptions can be met in order to perform the necessary statistical tests, in the case that the assumptions are violated and transformations are inappropriate or impossible, we will utilize the most powerful non-parametric approaches available. As a part of the experimental planning we will be conducting a replication analysis to determine the number of replications necessary to ensure that the type II error is controlled. We will also be conducting a power analysis at the completion of the experiments in order to validate that the power of the tests used was within expectations.

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. In the case of the experiments, there is no threat to internal validity due to the experiments being fully controlled. On the other hand, the case studies are not controlled and thus, will have issues relating to internal validity, as to be expected with case studies.

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. The use of any quality model to measure quality attributes opens a threat to construct validity due to nature of quality itself. Yet, the Quamoco quality model has already been validated by quality experts [159], and so does not pose a threat to the validity of our studies. Several measures have been proposed to measure technical debt principal and interest. In the experiments and case studies only technical debt principal is measured using the most well known measures. Thus, as we do not cover all the properties of technical debt with our measures there is a threat to construct validity. This threat is mitigated by restricting our conclusions to only the concept of technical debt principal.

Content Validity

Content validity is concerned with how well the selected measures cover the content domain. The experiments and case studies concerning quality cover the major characteristics of quality based on the ISO 25010 quality model using the Quamoco implementation, thus for these studies there are no threats to content validity. For those experiments and case studies concerning technical debt, technical debt is measured using the five proposed methods of measurement of technical debt principal, thus, there is no threat to content validity in these experiments either.

External Validity

External validity is concerned with the ability to generalize the results of a study. The experiments are conducted using existing open source systems implemented in

C#, Java™ , and C++. The use of these three languages side steps a potential threat due to only looking at a single language but does not escape the fact that they are languages all representative of a single paradigm, namely object-oriented and a specific type of object-orientation. Thus, we cannot generalize outside of object-oriented implementations. Since the experiments are to be conducted only on open source systems, there is a threat to the validity of these results, but we are mitigating this threat by conducting verification case studies on both open source and industry software. Thus, we will be able to validate that the results hold outside of open source systems.

Reliability

Reliability is concerned with the dependence between specific researchers and the data and analysis. Reliability is specifically of concern for the case studies we are going to conduct. In the case study designs we have selected to gather quantitative data using an automation framework and measurement tools. Thus, the data collected is not reliant on any researcher nor are the values collected left to interpretation. In all studies proposed the analyses will be conducted using scripts written for the R statitical processing software. The version of R used and the scripts create will be made available to all researchers. Hence, there should be no threats to the reliability of the studies.

TIMELINE

This chapter outlines the high-level goals and a work schedule to provide assessment of progress on the dissertation. As a part of the this we also present a list of current publications and future publications as a part of a publication plan. We conclude this chapter with a description of the proposed contributions of this research.

Work Plan

1. Revise proposal as starting point for dissertation, incorporating feedback from the comprehensive examination.

   Assessment: Evaluation and approval by Committee Chair.

2. Finish development of Disharmony Injection Framework.

   Assessment: Associated software made available for download.

3. Finish implementation of detection strategy framework and detection strategies for design pattern grime.

   Assessment: Publication of associated paper and distribution to committee members.

4. Complete the download and extraction of open source projects which will form the basis for the dataset used during the experiments.

5. Conduct the remaining experiments and cases studies.

   Assessment: Publication of associated paper and distribution to committee members.

6. Finish writing the dissertation. **Assessment:** Dissertation evaluated by advisor and deemed ready for committee review.

7. Present and defend the dissertation in Fall 2016.

Publication Plan

Current Publications

- "Preemptive Management of Model Driven Technical Debt for Improving Software Quality." [134]
- "A Simulation Study of Practical Methods for Technical Debt Management in Agile Software Development." [115]
- "The Correspondence between Software Quality Models and Technical Debt Estimation Approaches." [12]
- "Design Pattern Decay: The Case for Class Grime." [23]
- "Design Pattern Decay: An Extended Taxonomy and Empirical Study of Grime and its Impact on Design Pattern Evolution." [178]
- "On the Uncertainty of Technical Debt Measurements." [127]
- "Development and Application of a Simulation Environment (NEO) for Integrating Empirical and Computational Investigations of System-Level Complexity." [179]
- "Evolution of Legacy System Comprehensibility through Automated Refactoring." [180]
- "TrueRefactor: An Automated Refactoring Tool to Improve Legacy System and Application Comprehensiblity." [181]

Planned Publications

- "Design Pattern Decay: The Case of Organizational Grime"
- "The Evolution of Design Pattern Grime"
- "Automated Detection of Design Pattern Decay"

REFERENCES CITED

[1] Ward Cunningham. The WyCash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992. doi:10.1145/157710.157715.

[2] Jay Sappidi, Bill Curtis, Alexandra Szynkarski. The CRASH Report 2011/12: Summary of Key Findings. Technical Report, CAST Research Labs, 2012.

[3] J. de Groot, A. Nugroho, T. Back, J. Visser. What is the value of your software? *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 37 –44, June 2012. doi:10.1109/MTD.2012.6225998.

[4] Ariadi Nugroho, Joost Visser, Tobias Kuipers. An empirical model of technical debt and interest. *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, pages 1–8, Waikiki, Honolulu, HI, USA, 2011. ACM. doi:10.1145/1985362.1985364.

[5] B. Curtis, J. Sappidi, A. Szynkarski. Estimating the size, cost, and types of technical debt. *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 49 –53, June 2012. doi:10.1109/MTD.2012.6226000.

[6] B. Curtis, J. Sappidi, A. Szynkarski. Estimating the principal of an application's technical debt. *Software, IEEE*, 29(6):34 –42, December 2012. doi:10.1109/MS.2012.156.

[7] S. Chin, E. Huddleston, W. Bodwell, I. Gat. The economics of technical debt. *Cutter IT Journal*, 23(10):11–15, 2010.

[8] R.L. Nord, I. Ozkaya, P. Kruchten, 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*, pages 91–100, 2012. doi:10.1109/WICSA-ECSA.212.17.

[9] J. Letouzey, M. Ilkiewicz. Managing technical debt with the SQALE method. *Software, IEEE*, 29(6):44 –51, December 2012. doi:10.1109/MS.2012.129.

[10] J.-L. Letouzey. The SQALE method for evaluating technical debt. *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 31 –36, June 2012. doi:10.1109/MTD.2012.6225997.

[11] O Gaudin. Evaluate your technical debt with sonar. *Sonar, Jun*, 2009.

[12] Isaac Griffith, Derek Reimanis, Clemente Izurieta, Zadia Codabux, Ajay Deo, Byron Williams. The correspondence between software quality models and technical debt estimation approaches. pages 19–26. IEEE, September 2014. doi:10.1109/MTD.2014.13.

[13] Martin Fowler, Kent Beck, J Brant, William Opdyke, Don Roberts. *Refactoring: Improving the Design of Existing Programs*. Addison-Weseley, 1999.

[14] W.H. Brown, R.C. Malveau, T.J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998.

[15] Clemente Izurieta. *Decay and grime buildup in evolving object oriented design patterns*. Doctoral Dissertation, Colorado State University, 2009.

[16] C. Izurieta, A. Vetro, N. Zazworka, Yuanfang Cai, C. Seaman, F. Shull. Organizing the technical debt landscape. *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 23 –26, June 2012. doi:10.1109/MTD.2012.6225995.

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addison-Wesley, 1994.

[18] C. Izurieta, 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*, pages 449 –451, September 2007. doi:10.1109/ESEM.2007.55.

[19] C. Izurieta, J.M. Bieman. Testing consequences of grime buildup in object oriented design patterns. *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 171 –179, April/n 2008. doi:10.1109/ICST.2008.27.

[20] Clemente Izurieta, JamesM. Bieman. A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, pages 1–35, 2012. doi:10.1007/s11219-012-9175-x.

[21] R.B. France, Dae-Kyoo Kim, S. Ghosh, Eunjee Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, March 2004. doi:10.1109/TSE.2004.1271174.

[22] Melissa R. Dale, Clemente Izurieta. Impacts of design pattern decay on system quality. pages 1–4. ACM Press, 2014. doi:10.1145/2652524.2652560.

[23] Isaac Griffith, Clemente Izurieta. Design pattern decay: the case for class grime. pages 1–4. ACM Press, 2014. doi:10.1145/2652524.2652570.

[24] Melissa Dale. *Impacts of Modular Grime on Technical Debt.* Doctoral Dissertation, Montana State University, Bozeman, MT, 2014.

[25] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, January 2001. doi:10.1109/32.895984.

[26] David Lorge Parnas. Software aging. *Proceedings of the 16th International Conference on Software Engineering. ICSE'97*, pages 279–287, May 1994.

[27] Michael Grottke, Rivalino Matias, Kishor S. Trivedi. The fundamentals of software aging. pages 1–6. IEEE, November 2008. doi:10.1109/ISSREW.2008.5355512.

[28] Y. Huang, C. Kintala, N. Kolettis, N.D. Fulton. Software rejuvenation: analysis, module and applications. pages 381–390. IEEE Comput. Soc. Press, 1995. doi:10.1109/FTCS.1995.466961.

[29] K.S. Trivedi, K. Vaidyanathan, K. Goseva-Popstojanova. Modeling and analysis of software aging and rejuvenation. pages 270–279. IEEE Comput. Soc, 2000. doi:10.1109/SIMSYM.2000.844925.

[30] M.C. Ohlsson, A. von Mayrhauser, B. McGuire, C. Wohlin. Code decay analysis of legacy software through successive releases. pages 69–81 vol.5. IEEE, 1999. doi:10.1109/AERO.1999.790190.

[31] Joshua Kerievsky. *Refactoring to patterns.* Pearson Deutschland GmbH, 2005.

[32] Mika Mntyl. *Bad Smells in Software a Taxonomy and an Empirical Study.* Doctoral Dissertation, Helsinki University of Technology, 2003.

[33] M. Mantyla, J. Vanhanen, C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 381 – 384, September 2003. doi:10.1109/ICSM.2003.1235447.

[34] A.J. Riel. *Object-oriented design heuristics.* Addison-Wesley Longman Publishing Co., Inc., 1996.

[35] S. Wong, Yuanfang Cai, Miryung Kim, M. Dalton. Detecting software modularity violations. *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 411 –420, May 2011. doi:10.1145/1985793.1985850.

[36] Robert Schwanke, Lu Xiao, Yuangfang Cai. Measuring architecture quality by structure plus history analysis. 2013.

[37] Derek Reimanis, Clemente Izurieta, Rachael Luhr, Lu Xiao, Yuanfang Cai, Gabe Rudy. A replication case study to measure the architectural quality of a commercial system. pages 1–8. ACM Press, 2014. doi:10.1145/2652524.2652581.

[38] Moha, Naouel, Huynh, Duc-loc, Gueheneuc Y-G. A Taxonomy and a First Study of Design Pattern Defects. *IEEE International Workshop on Software Technology and Engineering Practice*, pages 225–229, Budapest, Hungary, 2005. IEEE Computer Society.

[39] Travis Schanz, Clemente Izurieta. Object oriented design pattern decay: a taxonomy. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 7:1–7:8, Bolzano-Bozen, Italy, 2010. ACM. doi:10.1145/1852786.1852796.

[40] Cédric Bouhours, Hervé Leblanc, Christian Percebois. Bad smells in design and design patterns. *Journal of Object Technology*, 8(3):43–63, 2009.

[41] Cdric Bouhours, Herv Leblanc, Christian Percebois. Sharing bad practices in design to improve the use of patterns. pages 1–24. ACM Press, 2010. doi:10.1145/2493288.2493310.

[42] Cdric Bouhours, Herv Leblanc, Christian Percebois. Spoiled patterns: how to extend the GoF. *Software Quality Journal*, August/n 2014. doi:10.1007/s11219-014-9249-z.

[43] T. Miceli, H.A. Sahraoui, R. Godin. A metric based technique for design flaws detection and correction. *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 307 –310, October 1999. doi:10.1109/ASE.1999.802337.

[44] Eva van Emden, Leon Moonen. Java quality assurance by detecting code smells. *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, October 2002.

[45] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350 – 359, September 2004. doi:10.1109/ICSM.2004.1357820.

[46] Bartosz Walter, Baej Pietrzak. Multi-criteria detection of bad smells in code with UTA method. *Extreme Programming and Agile Processes in Software Engineering*, 3556:1159–1161, 2005. doi:10.1007/11499053_18.

[47] M.J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. pages 15–15. IEEE, 2005. doi:10.1109/METRICS.2005.38.

[48] Ral Marticorena, Carlos Lpez, Yania Crespo. *Extending a Taxonomy of Bad Code Smells with Metrics.*

[49] M. Salehie, Shimin Li, L. Tahvildari. *A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws.* 2006.

[50] F. Khomh, S. Vaucher, Y. G. Gueheneuc, H. Sahraoui. A bayesian approach for the detection of code and design smells. *Quality Software, 2009. QSIC '09. 9th International Conference on*, pages 305–314, August/n 2009. doi:10.1109/QSIC.2009.47.

[51] Foutse Khomh. SQUAD: Software quality understanding through the analysis of design. pages 303–306. IEEE, 2009. doi:10.1109/WCRE.2009.22.

[52] I. Polasek, P. Liska, J. Kelemen, J. Lang. On extended similarity scoring and bit-vector algorithms for design smell detection. *Intelligent Engineering Systems (INES), 2012 IEEE 16th International Conference on*, pages 115 –120, June 2012. doi:10.1109/INES.2012.6249814.

[53] Ral Marticorena, Carlos Lpez, Yania Crespo. *Parallel Inheritance Hierarchy: Detection from a Static View of the System.*

[54] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni. Design defects detection and correction by example. *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 81 –90, June 2011. doi:10.1109/ICPC.2011.22.

[55] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, Manuel Wimmer. Search-based design defects detection by example. Dimitra Giannakopoulou, Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, Volume 6603 series *Lecture Notes in Computer Science*, pages 401–415. Springer Berlin / Heidelberg, 2011.

[56] N. Moha, Y.-G. Gueheneuc, P. Leduc. Automatic generation of detection algorithms for design defects. *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 297 –300, September 2006. doi:10.1109/ASE.2006.22.

[57] Naouel Moha, Yann-Gal Guhneuc, Anne-Franoise Le Meur, Laurence Duchien. A domain analysis to specify design defects and generate detection algorithms. Jos Fiadeiro, Paola Inverardi, editors, *Fundamental Approaches to Software*

Engineering, Volume 4961 series *Lecture Notes in Computer Science*, pages 276–291. Springer Berlin / Heidelberg, 2008.

[58] N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20 –36, February 2010. doi:10.1109/TSE.2009.50.

[59] D. Ratiu, S. Ducasse, T. Girba, R. Marinescu. Using history information to improve design flaws detection. *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 223 – 232, March 2004. doi:10.1109/CSMR.2004.1281423.

[60] D. Ratiu, R. Marinescu, S. Ducasse, T. Grba. Evolution-enriched detection of god classes. *Proc. of the 2nd CAVIS*, pages 3–7, 2004.

[61] Tudor Grba, Stphane Ducasse, Radu Marinescu, Ra\ctiu Daniel. Identifying entities that change together. *Ninth IEEE Workshop on Empirical Studies of Software Maintenance (WESS 2004)*. IEEE, 2004.

[62] Tudor Grba, Stphane Ducasse, Adrian Kuhn, Radu Marinescu, Ra\ctiu 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*, IWPSE '07, pages 83–89, Dubrovnik, Croatia, 2007. ACM. doi:10.1145/1294948.1294970.

[63] Baej Pietrzak, Bartosz Walter. Leveraging code smell detection with inter-smell relations. Pekka Abrahamsson, Michele Marchesi, Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, Volume 4044 series *Lecture Notes in Computer Science*, pages 75–84. Springer Berlin / Heidelberg, 2006.

[64] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, Denys Poshyvanyk. Detecting bad smells in source code using change history information. pages 268–278. IEEE, November 2013. doi:10.1109/ASE.2013.6693086.

[65] P.F. Mihancea, 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*, pages 92 – 101, March 2005. doi:10.1109/CSMR.2005.53.

[66] M. Kessentini, H. Sahraoui, M. Boukadoum, M. Wimmer. Design defect detection rules generation: A music metaphor. *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 241 –248, March 2011. doi:10.1109/CSMR.2011.30.

[67] Rim Mahouachi, Marouane Kessentini, Khaled Ghedira. A new design defects classification: Marrying detection and correction. Juan de Lara, Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, Volume 7212 series *Lecture Notes in Computer Science*, pages 455–470. Springer Berlin / Heidelberg, 2012.

[68] Usman Mansoor, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb. Code-smells detection using good and bad software design examples. Technical Report, Technical report, Technical Report, 2013.

[69] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, March 2013. doi:10.1007/s10515-011-0098-8.

[70] Foutse Khomh, Stephane Vaucher, Yann-Gal Guhneuc, Houari Sahraoui. BDTEX: A GQM-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, April/n 2011. doi:10.1016/j.jss.2010.11.921.

[71] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, Mika V. Mantyla. Code smell detection: Towards a machine learning-based approach. pages 396–399. IEEE, September 2013. doi:10.1109/ICSM.2013.56.

[72] Francesca Arcelli Fontana, Mika V. Mntyl, Marco Zanoni, Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, June 2015. doi:10.1007/s10664-015-9378-4.

[73] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, Michele Shaw. Building empirical support for automated code smell detection. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 8:1–8:10, Bolzano-Bozen, Italy, 2010. ACM. doi:10.1145/1852786.1852797.

[74] Nico Zazworka, 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*, ESEM '10, pages 63:1–63:1, Bolzano-Bozen, Italy, 2010. ACM. doi:10.1145/1852786.1852865.

[75] F.A. Fontana, P. Braione, M. Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2), 2012.

[76] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. pages 390–400. IEEE, October 2009. doi:10.1109/ESEM.2009.5314231.

[77] F. Khomh, M. Di Penta, 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*, pages 75 –84, October 2009. doi:10.1109/WCRE.2009.28.

[78] S.M. Olbrich, D.S. Cruzes, 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*, pages 1 –10, September 2010. doi:10.1109/ICSM.2010.5609564.

[79] Foutse Khomh, Massimiliano Di Penta, Yann-Gal Guhneuc, Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275, June 2012. doi:10.1007/s10664-011-9171-y.

[80] Daniele Romano, Paulius Raila, Martin Pinzger, Foutse Khomh. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. pages 437–446. IEEE, October 2012. doi:10.1109/WCRE.2012.53.

[81] Aiko Yamashita, Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *Journal of Systems and Software*, 86(10):2639–2653, October 2013. doi:10.1016/j.jss.2013.05.007.

[82] Aiko Yamashita, Leon Moonen. Do code smells reflect important maintainability aspects? pages 306–315. IEEE, September 2012. doi:10.1109/ICSM.2012.6405287.

[83] Aiko Yamashita, Leon Moonen. To what extent can maintenance problems be predicted by code smell detection? an empirical study. *Information and Software Technology*, 55(12):2223–2242, December 2013. doi:10.1016/j.infsof.2013.08.002.

[84] Dag I.K. Sjoberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, Tore Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, August/n 2013. doi:10.1109/TSE.2012.89.

[85] Aiko Yamashita. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering*, 19(4):1111–1143, August/n 2014. doi:10.1007/s10664-013-9250-3.

[86] Aiko Yamashita, Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. pages 682–691. IEEE, May 2013. doi:10.1109/ICSE.2013.6606614.

[87] F.A. Fontana, M. Zanoni. On investigating code smells correlations. *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 474 –475, March 2011. doi:10.1109/ICSTW.2011.14.

[88] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, Pawel Martenka. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. pages 260–269. IEEE, September 2013. doi:10.1109/ICSM.2013.37.

[89] E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton, J. Noble. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336 –345, December 2010. doi:10.1109/APSEC.2010.46.

[90] Tracy Hall, Min Zhang, David Bowes, Yi Sun. Some Code Smells Have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–39, September 2014. doi:10.1145/2629648.

[91] Dnes Bn, Rudolf Ferenc. Recognizing Antipatterns and Analyzing Their Effects on Software Maintainability. Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, Carmelo Torre, Jorge Gustavo Rocha, Maria Irene Falco, David Taniar, Bernady O. Apduhan, Osvaldo Gervasi, editors, *Computational Science and Its Applications ICCSA 2014*, Volume 8583, pages 337–352. Springer International Publishing, Cham, 2014.

[92] Tibor Bakota, Peter Hegedus, Peter Kortvelyesi, Rudolf Ferenc, Tibor Gyimothy. A probabilistic software quality model. pages 243–252. IEEE, September 2011. doi:10.1109/ICSM.2011.6080791.

[93] ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality REqurements and Evaluation (SQuaRE) – System and software quality models, 2011.

[94] Edith Tom, Aybke Aurum, Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, (0):–, 2013. doi:10.1016/j.jss.2012.12.052.

[95] Tim Klinger, Peri Tarr, Patrick Wagstrom, Clay Williams. An enterprise perspective on technical debt. *Proceedings of the 2nd Workshop on Managing*

Technical Debt, MTD '11, pages 35–38, Waikiki, Honolulu, HI, USA, 2011. ACM. doi:10.1145/1985362.1985371.

[96] Ted Theodoropoulos, Mark Hofberg, Daniel Kern. Technical debt from the stakeholder perspective. *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, pages 43–46, Waikiki, Honolulu, HI, USA, 2011. ACM. doi:10.1145/1985362.1985373.

[97] Klaus Schmid. On the limits of the technical debt metaphor some guidance on going beyond. *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 63–66, 2013. doi:10.1109/MTD.2013.6608681.

[98] Klaus Schmid. A formal approach to technical debt decision making. *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, QoSA '13, pages 153–162, New York, NY, USA, 2013. ACM. doi:10.1145/2465478.2465492.

[99] Steve McConnell. Managing technical debt. Best Practices White Paper 1, Construx, 2008.

[100] 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, Nico Zazworka. Managing technical debt in software-reliant systems. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 47–52, Santa Fe, New Mexico, USA, 2010. ACM. doi:10.1145/1882362.1882373.

[101] Philippe Kruchten, Robert L. Nord, Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *Software, IEEE*, 29(6):18 –21, December 2012. doi:10.1109/MS.2012.167.

[102] Yuepu Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F.Q.B. da Silva, A. L M Santos, C. Siebra. Tracking technical debt an exploratory case study. *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531, 2011. doi:10.1109/ICSM.2011.6080824.

[103] Carolyn Seaman, Yuepu Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011.

[104] Yuepu Guo, Carolyn Seaman. A portfolio approach to technical debt management. *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, pages 31–34, Waikiki, Honolulu, HI, USA, 2011. ACM. doi:10.1145/1985362.1985370.

[105] F.A. Fontana, V. Ferme, S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 15 –22, June 2012. doi:10.1109/MTD.2012.6225993.

[106] Nico Zazworka, Michele A. Shaw, Forrest Shull, Carolyn Seaman. Investigating the impact of design debt on software quality. *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, pages 17–23, Waikiki, Honolulu, HI, USA, 2011. ACM. doi:10.1145/1985362.1985366.

[107] Yuepu Guo, Rodrigo Oliveira Spnola, Carolyn Seaman. Exploring the costs of technical debt management a case study. *Empirical Software Engineering*, November 2014. doi:10.1007/s10664-014-9351-7.

[108] Johannes Holvitie, Ville Leppanen. DebtFlag: Technical debt management with a development environment integrated tool. *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 20–27, 2013. doi:10.1109/MTD.2013.6608674.

[109] Klaus Schmid. Technical debt – from metaphor to engineering guidance: A novel approach based on cost estimation. Informatikberichte 1/2013, SSE 1/13/E, Institute of Computer Science, University of Hildesheim, 2013.

[110] Marek G Stochel, Mariusz R Wawrowski, Magdalena Rabiej. Value-based technical debt model and its application. *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, pages 205–212, 2012.

[111] Barry Boehm, Li Guo Huang. Value-based software engineering: A case study. *Computer*, 36(3):33–41, 2003.

[112] Lucas Borante Foganholi, Rogrio Eduardo Garcia, Danilo Medeiros Eler, Ronaldo Celso Messias Correia, Celso Olivete Junior. Supporting technical debt cataloging with TD-Tracker tool.

[113] Davide Falessi, Michele A. Shaw, Forrest Shull, Kathleen Mullen, Mark Stein Keymind. Practical considerations, challenges, and requirements of tool-support for managing technical debt. *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 16–19, 2013. doi:10.1109/MTD.2013.6608673.

[114] Narayan Ramasubbu, Chris F. Kemerer. Towards a model for optimizing technical debt in software products. *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 51–54, 2013. doi:10.1109/MTD.2013.6608679.

[115] Isaac Griffith, Clemente Izurieta, Hannane Taffahi, David Claudio. A simulation study of practical methods for technical debt management in agile software development. *Proceedings of the 2014 Winter Simulation Conference*, pages 1014–1025, Savannah, GA, USA, December 2014. IEEE.

[116] Antonio Martini, Jan Bosch, Michel Chaudron. Architecture Technical Debt: Understanding Causes and a Qualitative Model. pages 85–92. IEEE, August/n 2014. doi:10.1109/SEAA.2014.65.

[117] Antonio Martini, Jan Bosch, Michel Chaudron. Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, 67:237–253, November 2015. doi:10.1016/j.infsof.2015.07.005.

[118] Davide Falessi, Alexander Voegele. Validating and Prioritizing Quality Rules for Managing Techncial Debt: An Industrial Case Study. *MTD 2015-UNDER REVISION*, 2015.

[119] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad.

[120] Md Abdullah Al Mamun, Christian Berger, Jorgen Hansson. Explicating, Understanding, and Managing Technical Debt from Self-Driving Miniature Car Projects. pages 11–18. IEEE, September 2014. doi:10.1109/MTD.2014.15.

[121] Nico Zazworka, Carolyn Seaman, Forrest Shull. Prioritizing design debt investment opportunities. *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, pages 39–42, Waikiki, Honolulu, HI, USA, 2011. ACM. doi:10.1145/1985362.1985372.

[122] R. Marinescu. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9:1 –9:13, October 2012. doi:10.1147/JRD.2012.2204512.

[123] J. Bansiya, C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, January 2002. doi:10.1109/32.979986.

[124] Nico Zazworka, Antonio Vetro, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, Forrest Shull. Comparing four approaches for technical debt identification. *Software Quality Journal*, 2012.

[125] Nicolli S.R. Alves, Leilane F. Ribeiro, Vivyane Caires, Thiago S. Mendes, Rodrigo O. Spinola. Towards an Ontology of Terms on Technical Debt. pages 1–7. IEEE, September 2014. doi:10.1109/MTD.2014.9.

[126] Radu Marinescu. Assessing and improving object-oriented design. 2012.

[127] C. Izurieta, I. Griffith, D. Reimanis, R. Luhr. On the uncertainty of technical debt measurements. *Information Science and Applications (ICISA), 2013 International Conference on*, pages 1–4, 2013. doi:10.1109/ICISA.2013.6579461.

[128] Vallary Singh, Will Snipes, Nicholas A. Kraft. A Framework for Estimating Interest on Technical Debt by Monitoring Developer Activity Related to Code Comprehension. pages 27–30. IEEE, September 2014. doi:10.1109/MTD.2014.16.

[129] Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Areti Ampatzoglou, Theodoros Amanatidis. Estimating the Breaking Point for Technical Debt.

[130] Areti Ampatzoglou, Apostolos Ampatzoglou, Paris Avgeriou, Alexander Chatzigeorgiou. Establishing a framework for managing interest in technical debt.

[131] Davide Falessi, Andreas Reichel. Towards an Open-Source Tool for Measuring and Visualizing the Interest of Technical Debt.

[132] Johannes Holvitie, Ville Leppanen, Sami Hyrynsalmi. Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey. pages 35–42. IEEE, September 2014. doi:10.1109/MTD.2014.8.

[133] Lu Xiao. Quantifying architectural debts. pages 1030–1033. ACM Press, 2015. doi:10.1145/2786805.2803194.

[134] C. Izurieta, G. Rojas, I. Griffith. Preemptive Management of Model Driven Technical Debt for Improving Software Quality. *Proceedings of the 11th International ACM SigSoft Conference on the Quality of Software Qrchitectures*, Montreal, Canada, May 2015.

[135] T. Mens, T. Tourwe. A declarative evolution framework for object-oriented design patterns. pages 570–579. IEEE Comput. Soc, 2001. doi:10.1109/ICSM.2001.972774.

[136] J.M. Bieman, G. Straw, H. Wang, P.W. Munger, R.T. Alexander. Design patterns and change proneness: an examination of five evolving systems. pages 40–49. IEEE Comput. Soc, 2003. doi:10.1109/METRIC.2003.1232454.

[137] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, Massimiliano Di Penta. An empirical study on the evolution of design patterns. page 385. ACM Press, 2007. doi:10.1145/1287624.1287680.

[138] Matt Gatrell, Steve Counsell, Tracy Hall. Design patterns and change proneness: A replication using proprietary c# software. pages 160–164. IEEE, 2009. doi:10.1109/WCRE.2009.31.

[139] A.H. Eden, A. Yehudai, J. Gil. Precise specification and automatic application of design patterns. pages 143–152. IEEE Comput. Soc, 1997. doi:10.1109/ASE.1997.632834.

[140] T. Mikkonen. Formalizing design patterns. pages 115–124. IEEE Comput. Soc, 1998. doi:10.1109/ICSE.1998.671108.

[141] Eric Eide, Alastair Reid, John Regehr, Jay Lepreau. Static and dynamic structure in design patterns. page 208. ACM Press, 2002. doi:10.1145/581339.581367.

[142] Toufik Taibi, David Chek Ling Ngo. Formal specification of design pattern combination using BPSL. *Information and Software Technology*, 45(3):157–170, March 2003. doi:10.1016/S0950-5849(02)000195-7.

[143] J.M. Smith, D. Stotts. SPQR: flexible automated design pattern extraction from source code. pages 215–224. IEEE Comput. Soc, 2003. doi:10.1109/ASE.2003.1240309.

[144] Dae-Kyoo Kim, R. France, S. Ghosh, Eunjee Song. A role-based metamodeling approach to specifying design patterns. pages 452–457. IEEE Comput. Soc, 2003. doi:10.1109/CMPSAC.2003.1245379.

[145] N. Soundarajan, J.O. Hallstrom. Responsibilities and rewards: specifying design patterns. pages 666–675. IEEE Comput. Soc, 2004. doi:10.1109/ICSE.2004.1317488.

[146] Soon-Kyeong Kim, D. Carrington. Using integrated metamodeling to define OO design patterns with object-z and UML. pages 257–264. IEEE, 2004. doi:10.1109/APSEC.2004.108.

[147] Dae-Kyoo Kim, Robert France, Sudipto Ghosh. A UML-based language for specifying domain-specific patterns. *Journal of Visual Languages & Computing*, 15(3-4):265–289, June 2004. doi:10.1016/j.jvlc.2004.01.004.

[148] OMG Unified Modeling Language™ (OMG UML), Infrastructure, August/n 2011.

[149] Object Constraint Language, February 2014.

[150] Dae-Kyoo Kim. Evaluating conformance of UML models to design patterns. pages 30–31. IEEE, 2005. doi:10.1109/ICECCS.2005.38.

[151] Dae-Kyoo Kim, Wuwei Shen. An approach to evaluating structural pattern conformance of UML models. page 1404. ACM Press, 2007. doi:10.1145/1244002.1244305.

[152] Dae-Kyoo Kim, Wuwei Shen. Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies. *Software Quality Journal*, 16(3):329–359, September 2008. doi:10.1007/s11219-008-9048-5.

[153] Shane Strasser, Colt Frederickson, Kevin Fenger, Clemente Izurieta. An automated software tool for validating design patterns. *ISCA 24th International Conference on Computer Applications in Industry and Engineering. CAINE*, Volume 11, 2011.

[154] Lunjin Lu, Dae-Kyoo Kim. Required behavior of sequence diagrams: Semantics and refinement. pages 127–136. IEEE, April/n 2011. doi:10.1109/ICECCS.2011.20.

[155] Lunjin Lu, Dae-Kyoo Kim. Required behavior of sequence diagrams: Semantics and conformance. *ACM Transactions on Software Engineering and Methodology*, 23(2):1–28, March 2014. doi:10.1145/2523108.

[156] D.-K. Kim, J. Whittle. Generating UML models from domain patterns. pages 166–173. IEEE, 2005. doi:10.1109/SERA.2005.44.

[157] Victor R. Basili. Software modeling and measurement: The goal/question/-metric paradigm. Technical Report, University of Maryland at College Park, College Park, MD, USA, 1992.

[158] Ethem Alpaydin. *Introduction to machine learning*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, edition 2nd ed, 2010.

[159] Stefan Wagner, Klaus Lochmann, Lars Heinemann, Michael Klas, Adam Trendowicz, Reinhold Plosch, Andreas Seidi, Andreas Goeb, Jonathan Streit. The Quamoco product quality modelling and assessment approach. pages 1133–1142. IEEE, June 2012. doi:10.1109/ICSE.2012.6227106.

[160] Clemente Izurieta, James Adviser-Bieman. *Decay and grime buildup in evolving object oriented design patterns*. Doctoral Dissertation, Colorado State University, 2009.

[161] R.C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[162] Edward Yourdon, Larry L. Constantine. *Structured design: fundamentals of a discipline of computer program and systems design*. Prentice Hall, Englewood Cliffs, N.J, 1979.

[163] L.C. Briand, J.W. Daly, J.K. Wust. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998. 10.1023/A:1009783721306.

[164] James M. Bieman, Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. *ACM SIGSOFT Software Engineering Notes*, 20(SI):259–262, August/n 1995. doi:10.1145/223427.211856.

[165] L.C. Briand, S. Morasca, V.R. Basili. Measuring and assessing maintainability at the end of high level design. pages 88–87. IEEE Comput. Soc. Press, 1993. doi:10.1109/ICSM.1993.366952.

[166] Apostolos Ampatzoglou, Olia Michou, Ioannis Stamelos. Building and mining a repository of design pattern instances: Practical and research benefits. *Entertainment Computing*, 4(2):131–142, April/n 2013. doi:10.1016/j.entcom.2012.10.002.

[167] Ronald Aylmer Fisher. *Statistical methods for research workers*. Oliver and Boyd, Edinburgh, edition 14th ed., revised and enlarged, 1970.

[168] John W. Tukey. Comparing Individual Means in the Analysis of Variance. *Biometrics*, 5(2):99, June 1949. doi:10.2307/3001913.

[169] James Stevens. *Applied multivariate statistics for the social sciences*. Routledge, New York, edition 5th ed, 2009.

[170] Jonathan D. Cryer, Kung-sik Chan. *Time series analysis: with applications in R*. Springer texts in statistics. Springer, New York, edition 2nd ed, 2008.

[171] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81, June 1938. doi:10.2307/2332226.

[172] Joint Task Force Transformation Initiative. Security and Privacy Controls for Federal Information Systems and Organizations. Technical Report NIST SP 800-53r4, National Institute of Standards and Technology, April/n 2013.

[173] D. Campbell, T. D. Cook. *Quasi-experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, 1979.

[174] D. Campbell, J. Stanley. *Experimental and Quasi-experimental Designs for Research*. Rand-McNally, 1963.

[175] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, Anders Wessln. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[176] Robert K. Yin. *Case study research: design and methods*. Number v. 5 series Applied social research methods. Sage Publications, Los Angeles, Calif, edition 4th ed, 2009.

[177] Per Runeson, editor. *Case study research in software engineering: guidelines and examples*. Wiley, Hoboken, N.J, 2012.

[178] Isaac Griffith, Clemente Izurieta. Design pattern decay: An extended taxonomy and empirical study of grime and its impact on design pattern evolution. 2013.

[179] C. Izurieta, G. Poole, R.A. Payn, I. Griffith, R. Nix, A. Helton, E. Bernhardt, A.J. Burgin. Development and application of a simulation environment (NEO) for integrating empirical and computational investigations of system-level complexity. *Information Science and Applications (ICISA), 2012 International Conference on*, pages 1 –6, May 2012. doi:10.1109/ICISA.2012.6220928.

[180] Isaac Griffith, Scott Wahl, Clemente Izurieta. Evolution of legacy system comprehensibility through automated refactoring. *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, MALETS '11, pages 35–42, Lawrence, Kansas, United States, 2011. ACM. doi:10.1145/2070821.2070826.

[181] I. Griffith, S. Wahl, C. Izurieta. TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility. *24 International Conference on Computer Applications in Industry and Engineering*, CAINE '11, Honolulu, HI, USA, November 2011. ISCA.