Design Pattern Decay: An Extended Taxonomy and Empirical Study of Grime and its Impact on Design Pattern Evolution

Isaac Griffith and Clemente Izurieta (Advisor)
Department of Computer Science
Montana State University
Bozeman, MT 59717-3880
1+(406) 994-4780
isaac.griffith@msu.montana.edu
clemente.izurieta@cs.montana.edu

ABSTRACT

Design patterns are well known solutions to common problems and are extensively utilized in software development. Yet, little empirical work has been conducted to evaluate or validate the consequences that poor design decisions have on pattern realizations.

This paper describes a research program to further the understanding of design pattern evolution. Specifically, we focus on design pattern decay by studying how grime, a decisively negative consequence of software evolution occurs. The research proposed herein furthers the exploration of design pattern decay by providing empirical evidence of grime buildup, a new grime taxonomy, and the consequences exhibited through decreased adaptability and maintainability in actual realizations of patterns in code. These notions will be supported through the development of semi-automated grime detection and refactoring research tools that will also link to existing forms of design decay such as code smells, anti-patterns, and modularity violations. An extension of this research focuses on the exploration of these notions inlying coupled pattern realizations.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design — Design Concepts. *Object-oriented design methods;* D.2.11 [**Software Engineering**]: Software Architectures — *Patterns*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement — *Enhancement, Maintainability, Maintenance measurement.*

General Terms

Measurement, Design, Experimentation.

Keywords

Software Architectures, Object Oriented Design Patterns, Software Evolution, Software Decay, Technical Debt.

1. INTRODUCTION

It is well known that design patterns have become a widely used technique in software development. However, empirical results show that design patterns are not immune from the negative side effects of software decay [23][24][25][26]. The decay of pattern realizations involves the obfuscation of the pattern's original intent. That is, as a pattern realization evolves, its structure and behavior tends to deviate from its original intent. 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.

A specific type of design pattern decay is design pattern grime [24] (further referred to as "grime"). Grime refers to the accumulation of structural and behavioral artifacts that cause a deviation from the intended design of a pattern [24]. Although some work has been done to understand grime, there are still several gaps in our knowledge of this phenomenon. The overarching goal of this research is to explore grime and its relationship to other types of design defects by following a measurement driven approach that will further characterize its nature, taxonomy, and consequences on adaptability and maintainability. Methods to mitigate the potential negative impact it has on the architecture of software products will be investigated through the development of research tools in support of semi-automated detection and refactoring strategies.

This paper is organized as follows: Section 2 describes issues with the current research to be addressed at IDoESE. Section 3 describes background and related work. Section 4 describes the objectives and hypotheses that will be explored by this research. Section 5 provides the underlying research methods that will be followed. Section 6 indicates potential threats to the validity of the study. Finally, section 7 concludes this paper with a summary of the current progress of the research program and immediate plans for future work.

2. CURRENT RESEARCH ISSUES

This dissertation was started in September of 2011 and we seek advice on its goals and merits; however we would like to receive the most advice on the following issues:

- Is there an efficient (partially or fully automated) way to validate detection of design patterns? This is currently a manual task.
- Assuming that we utilize existing design pattern detection, design defect detection (code smells [14], anti-patterns [6], and modularity violations [48]), and UML conformance checking tools, how can we account for the error in measurement inherent in the various tools?
- We are currently planning on developing custom tools. Would it be better to rely upon existing, though limited in functionality, tools? For example, several tools developed to detect code smells and anti-patterns only find a handful of the known types.

3. BACKGROUND AND RELATED WORK

3.1 Design Pattern Evolution

Design patterns were widely introduced by Gamma et al. [17], and represent abstract definitions of agreed upon design solutions to commonly recurring problems in software development.

Design patterns represent a form of micro-architecture within a software project, and thus, are also subject to evolutionary disharmonies. However, said disharmonies can be more readily studied, but few empirical studies of design pattern evolution and decay exist in the literature.

In order to study design pattern realizations a means to specify a pattern is necessary. Various design pattern languages have been proposed [15][4][7], and although some are visual and some textual, their intent is similar –a higher level of representational abstraction. Yet, although the representational aspects may be well understood, the specification of realizations that are conformant to their intended design remains a hard problem.

3.2 Software Decay and Aging

Software decay is a specific form of software evolution. When a system has evolved such that it becomes "harder to change than it should be" [13], then it is said to have suffered from decay. Another phenomenon, identified by Parnas [39] that complements software decay is *software aging*. It describes how changes in a software system's environment can reduce the overall value of software.

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 [21][22][47][13][38]. Specific types of software decay have been identified, such as the code smell identified by Fowler et al. and the anti-pattern identified by Brown et al. These and other types of software decay and aging recently have fallen under the umbrella of *technical debt* (see section 3.2.2).

3.2.1 Design Pattern Decay

Izurieta and Bieman [24] identified a new type of software decay known as design pattern decay. Design pattern decay has been primarily studied by Izurieta and Bieman [24][25][26], Izurieta and Schanz [42] and Izurieta [23]. Initially, Izurieta [23] identified two distinct categories of design pattern decay: *design pattern grime* and *design pattern rot*. Of these two types, empirical studies have only confirmed the existence of grime. Grime was initially subdivided into three disjoint categories: *class grime*, *modular grime*, and *organizational grime*. This initial taxonomy is depicted in the top portion of Figure. 1.

Seminal work by Izurieta [23] found that pattern realizations tend to accumulate artifacts that obscure the intended use of patterns. Empirical studies further showed that, of the three types of grime, modular grime was the most significant [25]. Based on this, Schanz and Izurieta [42] further expanded a taxonomy of modular grime by subdividing the original classification into six disjoint types of grime. They conducted a series of empirical studies across open source systems in order to validate the existence of these types of grime. This extended taxonomy (lower portion of Figure 2) was based on properties of class coupling identified by Briand et al. [5]. Modular grime was hierarchically subdivided by the strength, scope, and direction of coupling [42].

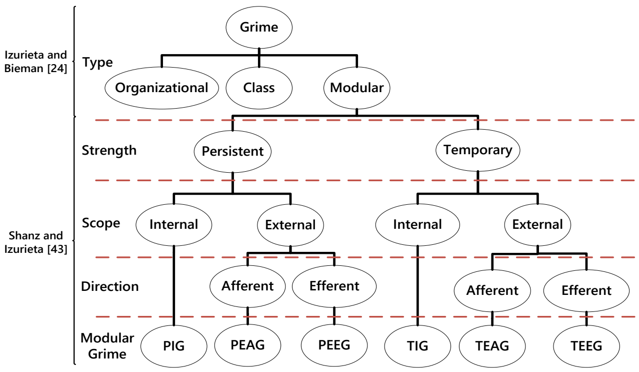

Figure 1. Extended grime taxonomy, which focuses on modular grime.

Further empirical studies on grime have shown implications in the area of testing [25]. Based on this work Izurieta et al. [28] 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.

3.2.2 Design Pattern Coupling

Design pattern coupling was initially studied by McNatt and Bieman [33]. Their goal was to identify how often pattern realization coupling occurs and to investigate the benefits of design pattern coupling. They found that pattern couplings frequently occur, and that typical pattern couplings were of the intersection type (based on use dependencies between pattern realizations).

Bieman and Wang [3] examined the effects of pattern coupling. They found that couplings tend to introduce dependencies which can increase the fault proneness and lower the adaptability of the end product, and that couplings based on association (persistent couplings) are more likely to be change prone and exhibit higher coupling (leading to a reduced modification stability).

3.2.3 Technical Debt

Technical debt is a metaphor originally described by Cunningham [11] as a way of explaining the need to restructure software through a financial metaphor. Technical debt was later mentioned by both Fowler et al. [14] as an argument towards the benefits of refactoring as a means of reducing technical debt. Today, the agile community views the management of known and unknown technical debt items as first class objects that once identified, should be tracked (over their lifetime) as a part of a combined backlog [18]. For a deeper exploration of recent research, we refer the reader to a comprehensive multi-vocal literature review by Tom et al. [46].

3.3 Current Research Gaps

The management of design pattern decay forms an important component, in the management of software aging and technical debt, which warrants further research. The following is a list of research gaps that have been identified in this area:

- *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. Furthermore, the increasing use of modularity frameworks, such as OSGi[1], [29] indicates that these types of grime will become more prevalent.
- *Pattern Coupling* – Patterns are often used in conjunction with other patterns [33][3]. The exploration of couplings between patterns, and the effects of these couplings on grime requires further investigation.
- *Quality* – The impact of grime on the quality of both software products and pattern realizations has only been subject to limited study [25][23][26].

[1] http://www.osgi.org

- *Technical Debt* – Current research has looked into how grime plays a part in the technical debt landscape [49]. The effect of grime on the technical debt value of a software product and pattern realizations has yet to be studied.
- *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.

3.4 Proposed Contributions

In this dissertation we propose the following contributions to address the gaps:

- Comprehensive literature review covering design pattern grime and design pattern defects and their relationship to quality, other types of design defects, and software quality.
- Using Marinescu's notion of Detection Strategies [32] coupled with design pattern detection we plan to automate the detection of grime.
- Using automated techniques to extract information over several versions of selected software projects to explore how grime develops in singular and coupled pattern realizations.
- Development of a completed taxonomy of grime, including a refined taxonomy of class and organizational grime with enhancements to include design pattern coupling.
- Evaluating a large collection of software (with multiple versions and from multiple languages) to confirm the existence of grime.
- Exploring methods to calculate the technical debt, pattern adaptability, refactoring difficulty, and refactoring complexity of grime.
- Defining and developing semi-automated and automated methods to refactor patterns in the face of grime accumulation.
- Development of a defect injection method to study the effects of design defects on software in a controlled manner.
- Connection between design pattern grime, technical debt, and software quality.

4. OBJECTIVES

This section identifies the research objectives and underlying hypotheses. In order to further develop the objectives into testable hypotheses and identify a set of research questions, we followed the Goal Question Metric (GQM) method [2]. Each object is identified by a research goal (RG) followed by a set of research questions (RQ).

4.1 Research Objectives

RG1 – Analyze pattern realizations 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.

RQ1.1 – What is the precision [1] of the grime detection algorithms for each type of grime?

RQ1.2 – What is the recall [1] of the grime detection algorithms for each type of grime?

RQ1.3 – What is the average number of grime instances per grime type discovered across all identified pattern realizations?

RQ1.4 – What is the average number of grime instances per type per system?

RG2 – Analyze pattern realizations for the purpose of tracking grime buildup with respect to the amount of grime from the perspective of several versions in the context of open source software projects.

RQ2.1 – How does grime change over time?

RQ2.2 – Which type of grime is more likely to occur as pattern realizations evolve?

RQ2.3 – What relationships exist between pattern realization evolution and grime growth?

RG3 – Analyze pattern realizations affected by grime for the purpose of identifying when refactoring is needed from the perspective of several system versions in the context of open source software projects.

RQ3.1 – What refactoring combinations are needed to remove each type of grime?

RQ3.2 – When is refactoring needed to return a pattern realization back to the original pattern?

RQ3.3 – When is refactoring needed to change a pattern realization to a different pattern (refactoring to other patterns)?

RQ3.4 – At what point does a pattern realization no longer resemble the intended design pattern?

RQ3.5 – At what point is refactoring away from patterns necessary?

RG4 – Analyze pattern realizations for the purpose of evaluation with respect to susceptibility to grime buildup from the perspective of grime in the context of open source software projects.

RQ4.1 – Which patterns or pattern families are more susceptible to grime accumulation?

RQ4.2 – Which patterns or pattern families need to be watched more closely by development teams?

RG5 – Analyze pattern realizations for the purpose of identifying intra- and inter-relationships with respect to pattern grime from the perspective of a grime taxonomy and other defect types (i.e., code smells, anti-patterns, modularity violations, etc.) in the context of open source projects.

RQ5.1 – What are the relationships between the different types of grime?

RQ5.2 – What relationships exist between grime and code smells?

RQ5.3 – What relationships exist between grime and anti-patterns?

RQ5.4 – What relationships exist between grime and modularity violations?

RG6 – Analyze pattern realizations 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 calculate the technical debt associated with the grime buildup in a pattern realization?

RG7 – Analyze pattern realizations for the purpose of evaluation with respect to grime form the perspective of design pattern couplings in open source software.

RQ7.1 – What is the precision and recall of pattern realization coupling detection algorithms?

RQ7.2 – What is the precision and recall of pattern coupling grime detection algorithms?

RQ7.3 – Which pattern combinations, as identified by Gamma et al. [17], are more susceptible to grime buildup?

RQ7.4 – Which type of grime is more likely to build up in coupled design patterns?

RG8 – Analyze pattern realizations afflicted with grime for the purpose of evaluation with respect to maintainability and adaptability from the perspective of design pattern realizations in the context of open source software projects.

RQ8.1 – What is the effect of each type of grime on the *adaptability* of a pattern realization?

RQ8.2 – What is the effect of each type of grime on the *maintainability* of a pattern realization?

4.2 Important Metrics

We have identified several metrics that will be used for the various experiments described in section 5.

M1. *Grime Count* (GC) – The total amount of grime accumulated within a pattern realization. This metric will be used in RQ1.1—RQ1.4, RQ2.1, RQ2.2, RQ4.1, RQ4.2, RQ6.1, RQ6.2, RQ7.1 and RQ7.2.

M2. *Refactoring Count* (RCT) – A count of the number of refactorings needed to remove grime buildup within a pattern realization. This metric will be used in RQ3.1—RQ3.5, RQ 6.1 and RQ6.2.

M3. *Refactoring Difficulty* (RD) – A measure of the difficulty of refactoring a design defect. This metric will be used in RQ3.1—RQ3.5, RQ6.1 and RQ6.2.

M4. *Grime Susceptibility* (GS) – A measure of the probability that a pattern realization will accumulate grime. By grouping pattern realizations, in a yet to be determined fashion, and counting the amount of grime, we can identify which members of the group are more susceptible to grime. This metric will be used in RQ4.1, RQ4.2, and RQ7.3.

M5. *Pattern Coupling Count* (PCC) – A measure of the number of distinct pattern realizations that another pattern realization is coupled to. This metric will be used in RQ7.1—RQ7.4.

M6. *Technical Debt* (TD) – a measure of the cost required to remove all technical debt principle in a particular software system. This metric will be used in RQ6.1 and RQ6.2

M7. *Grime Growth* (GG) – A measure of the change in grime from a specific pattern realization over the evolution of the system. This metric will be used in RQ2.1—RQ2.3, RQ6.1, RQ6.2, RQ7.1—RQ7.4, RQ8.1 and RQ8.2.

M8. *Pattern Adaptability* (PA) – A measure of the ability of a pattern realization to adapt to future requirements. It will be measured using the surrogate metric: Architecture Adaptability Index (AAI) [44]. This metric, AAI, will be used in RQ8.1 and RQ8.2.

M9. *Pattern Maintainability* (PM) – A measure of the maintainability of a design pattern. We will use the following surrogate metrics for maintainability as indicated by Li and Henry [31]: From the Chidamber and Kemerer [9] metrics suite we will use Depth of Inheritance Tree (DIT), Number of Children (NOC), Lack of Cohesion in Object Methods (LCOM), Weighted Methods per Class (WMC) and Response For Class (RFC). From Li and Henry [31] we will use Data Abstraction Coupling (DAC), Message Passing Coupling (MPC) and Number of Methods (NOM). These metrics will be used in RQ8.1 and RQ8.2.

M10. *Grime Severity* (GS) – A measure of the severity of an instance of a specific form of grime. These metrics will be used in RQ6.1, RQ6.2, RQ8.1 and RQ8.2.

4.3 Working Hypotheses

H1. Accurate detection of grime is achievable with automated detection strategies.

H2. There exists evidence to support the accumulation of modular, class, and organizational grime in open source systems

H3. There exists a point when the grime buildup within a pattern realization indicates that the pattern realization must be refactored to its original intent, to another more applicable pattern, or away from patterns in general and towards alternate good designs.

H4. Pattern susceptibility to each type of grime is related to the type of pattern (behavioral, creational, or structural) or structural meta-pattern [40] a pattern realization belongs to.

H5. Relationships and Grime

H5.1. Grime is distinguishable through the different types of relationships that exist between grime types.

H5.2. Grime is distinguishable through the different types of relationships that exist between grime types and other non-pattern related design defects.

H6. Grime and Technical Debt:

H6.1. Grime has a negative effect on the technical debt of a software system as a whole.

H6.2. Grime has a negative effect on the technical debt of a pattern realization.

H7. Patterns tend to be used together and this use makes these coupled pattern realizations more susceptible to grime buildup.

H8. Each type of grime has a significant negative impact on the maintainability and adaptability of pattern realizations.

5. METHODS

This section describes the underlying research methods used to further understand grime and its effects.

5.1 Data Collection

The data collection process and framework is depicted in Figure 2. Depending on the study approach (controlled experiment or case study) we can begin in one of two ways. Case studies will be carried out in selected groups of open source projects that are representative of different programming languages. In a controlled experiment, we utilize a model injector (a tool designed to introduce software artifacts into a model of the software) to both create a model and inject the grime (or other design defects) entities under study. Since the model is a representation of the source code, once the model is available, product measures are extracted from the model and both the model and metrics are placed into the database. For the case studies, we also simultaneously collect pattern realization information using design pattern detection methods, which are then manually verified. Upon verification, the pattern realization information is stored in the database.

The design information now stored in the database is used to collect additional data. Simultaneously, we extract design defect information (such as code smells, anti-patterns, and modularity violations), identify and extract grime information, and identify coupled pattern realizations. Once all this information is collected, it is stored in the database. In the final step, design information is used to extract response measures that are in turn used in the analysis phase. The following subsections provide greater detail on the process and tools which makeup the data collection framework.

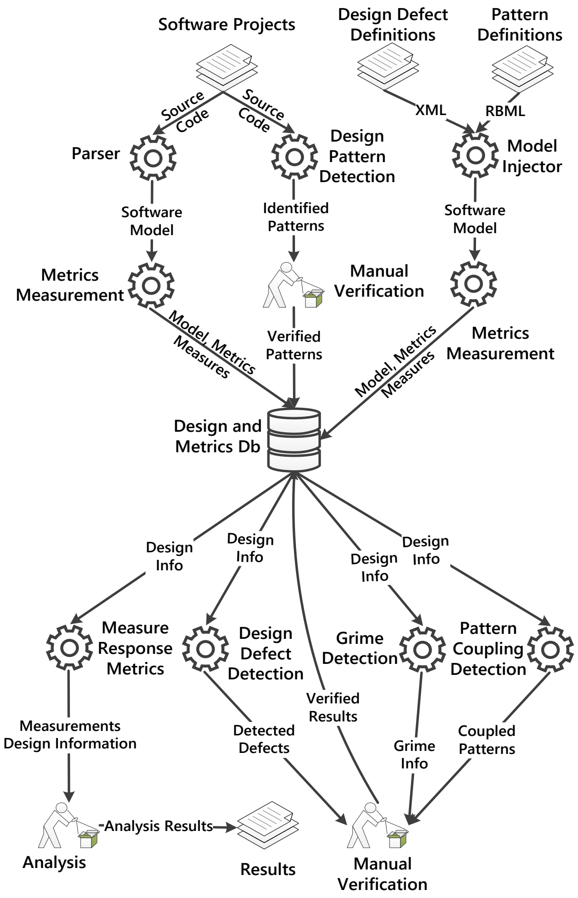

Figure 2. Depiction of the data collection process and framework.

5.1.1 Design Patterns Studied

The proposed study will encompass pattern realizations, which are either injected or found through detection methods, as defined by Gamma et al. [17].

5.1.2 Software Studied

The proposed study evaluates the software systems composed of several open source systems. We intend to develop a collection of open source software developed in the Java^{TM}, C++, C#, and Ruby programming languages using the method developed by Tempero et al. [45].

Every available software version in all available systems contained in the corpus will be evaluated. The selection of systems to be included in the evaluation depends on the research goal being addressed. Some goals require a single system and others a complete evolutionary analysis of multiple versions and possibly multiple systems.

5.1.3 Design Pattern Detection

In order to evaluate grime, we must first detect pattern realizations in their setting of use. In order to reduce the amount of work required to manually identify and validate pattern realizations, we are investigating design pattern detection algorithms that will be directly embedded into the framework.

5.1.4 Parser and Metrics Measurement

In order to detect both grime and other design defects we have constructed a parser using the SableCC framework [16]. We have also implemented several metrics [37][35][34][32][30] that have been shown to detect different types of grime as well as various other defects (i.e., code smells, anti-patterns, and modularity violations). Once the structural representations and metrics have been collected, they are stored in the database.

5.1.5 Design Pattern Grime Detection

There exists no automated method for detecting pattern grime within a software system. We are exploring the combination of RBML-UML conformance checking algorithms [43] and algorithms based on Marinescu's detection strategies [32] to develop an automated tool for identifying grime instances. The output of this tool will identify grime instances associated with a known pattern realization. Identified grime instances will be stored in the database. It should be noted that if automated or semi-automated methods are not fruitful in identifying grime, manual detection can still be utilized without hindering the overall process but at a significant cost in time.

5.1.6 Design Defect Detection

Utilizing the same underlying technology as the grime detector, we will include detection strategies for code smells, anti-patterns, and other design defects. These detection strategies are based on the work of Marinescu [32], Lanza and Marinescu [30], Moha et al. [34][35][36], Munro [37], and others. These detection strategies utilize the metrics and structural information stored in the database. The goal is to extract and store in the database detected design defects associated with a pattern realizations and its surrounding classes.

5.1.7 Design Pattern Coupling Detection

We will analyze the structural coupling between pattern realizations. These pattern couplings are detected by analysis of shared architecture between pattern realizations in a given component of software. The information for this process is extracted from the database and analyzed by a form of pair-wise comparison between realizations.

5.1.8 Software Model and Model Injection

We will utilize a meta-model of a software project that facilitates information extraction from a common data structure. The model uses a graph that combines the semantics of UML[2] class and sequence diagrams with that of call-graphs. This model can either be extracted from an abstract syntax tree provided by a parser or can be generated programmatically. The model can be used directly or design defects (such as grime, code smells, etc.) and pattern realizations can be injected into the model based on formal descriptions (provided via XML[3] and RBML [15] descriptions).

5.2 Research Approach

The research approach selected depends on the research question, but it can be an experiment or case study.

Initially we will conduct a systematic literature review to support the development of the grime taxonomy as well as to investigate existing approaches towards automating detection of design defects

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

[3] http://www.w3.org/TR/2006/REC-xml11-20060816/

such as code smells. This literature review will not only investigate academic sources but industry sources as well.

RQ1.1 and RQ1.2 will be conducted as an experiment to evaluate the detection strategies used to identify grime in pattern realizations. The experiment will be conducted by executing detection algorithms across generated and grime injected pattern realizations. We will then calculate the precision and recall of each algorithm. We will compare these results against pure random assignment in a 10-fold cross validation [1] approach. The data will be analyzed using a student's t-test for paired data [41]. RQ1.3 and RQ1.4 will be investigated by using these tested detection algorithms to identify grime in the collected software projects.

RQ2.1—RQ2.3 will be evaluated through the use of a multiple case study design by observing the evolution of grime in pattern realizations across the different versions of each software project under study. We will look for statistically significant correlations between time and changes in different types of grime.

RQ3.1—RQ3.5 will be evaluated using a controlled experiment and a multiple case study. The controlled experiment will evaluate refactoring strategy effectiveness, until we have identified refactoring strategies to remove each type of grime (RQ3.1). We will also vary the amount of injected grime to determine the effect on PA, RD, and RC (RQ3.2—RQ3.5). We are looking to develop a statistical model (mathematical model which describes the behavior of an object under study using random variables [41], in this case grime and refactoring), evaluated using the ANOVA method [41], that can assess alternatives and provide decision support before refactorings are performed. Once, the statistical models have been evaluated we will use them to evaluate pattern realizations found within the available software projects.

RQ4.1 and RQ4.2 will be evaluated using a multiple longitudinal case study approach across all versions of available software. For each type of pattern (creational, structural, behavioral or meta-pattern type) we will extract grime affected pattern realizations and evaluate the pattern's susceptibility to grime accumulation. We will evaluate results using a one-way ANOVA model.

RQ5.1—RQ5.4 will be evaluated using a controlled experiment and a multiple case study approach across all versions of available software. The controlled experiment will be used to validate the detection algorithms (similar to the approach used in RQ1.1 and RQ1.2). The case study approach will collect information regarding code smells, anti-patterns, modularity violations, and grime with corresponding source code locations within the software. We will then use the collected instances of grime and design defects to determine if any relationships exist between the different types of grime, and to determine if any relationships exist between types of grime and the other design defects. Once identified, we will evaluate if there is statistical evidence that suggests whether defect type relationships are meaningful.

RQ6.1 will be conducted as a controlled experiment. We will generate a system with pattern realizations injected with grime. Prior to and after each injection we will measure the technical debt associated with the system. We can then identify the effect that each type of grime has on technical debt. Along with the evaluation of the impact on TD we will also consider the evaluation of grime severity and its relationship to TD. The analysis of the effect of each type of grime on technical debt will be conducted through the use of an ANOVA model and pair-wise comparison. We will then conduct the multiple case study described in RQ6.2 to validate the statistical models.

RQ7.1 and RQ7.2 will be evaluated using a controlled experiment. The analysis is similar to RQ1.1 and RQ1.2. RQ7.3 and RQ7.4 and will be evaluated using a multiple case study that cross cuts the available software projects. The case study will investigate grime buildup in pattern couplings and identify coupled patterns that appear more susceptible to grime. The goal is to develop statistical models, evaluated using ANOVA, of how coupling affects non-coupled grime and grime susceptibility to identify which pattern coupling-based grime types, patterns are more susceptible to.

RQ8.1 and RQ8.2 will be evaluated using a controlled experiment and a multiple case study approach. For each pattern realization the PA, PM, and GG metrics will be measured before and after grime injection. As grime is introduced the change in PA and PM will be observed. We will then evaluate the change in PA and PM versus GG for statistically significant correlations. In the case studies we will be tracking grime to see if the relationship holds in actual software systems. We also expect to explore how grime and grime severity affects portions of the Quamoco[12] quality model by specifically focusing on maintainability and adaptability as measured by PM and PA.

6. THREATS TO VALIDITY

There are several threats to the validity of the proposed study, which are based on the classification scheme of Cook, Campbell and Day [10] and of Campbell et al. [8]. We have identified threats to internal and external validity, and seek advice to minimize construct validity, before we setup the experiments.

There is a potential threat to internal validity due to other design defects overlapping with grime. In order to control for this we conduct experiments to determine relationships between other design defects and grime. Similarly, a potential threat can occur due to overlap between specific types of grime. Although considered to be disjoint within each type (modular, class, and organizational) there may be overlap between types. To control for this we will conduct experiments to evaluate relationships between types of grime.

Given that these studies focus on open source software, there are two threats to external validity. The first is the exclusive use of open source software. We can only control for this threat by including studies on commercial software, but this is left to future study.

The second threat to the external validity is the total number of patterns and pattern realizations studied. In previous studies on grime have focused on a small number and types of design patterns. Here we will use a large number of realizations of each type of pattern and we are looking into the entire catalog of the Gang of Four patterns, which alleviates the issue that only a small subset of well-known patterns is likely to have grime. Because of the threats we cannot generalize beyond those systems studied.

7. CONCLUSIONS

In this paper we described a set of empirical studies to further explore and understand the phenomenon of design pattern grime. We intend to not only explore the accumulation of grime via pattern realization evolution but to also explore the intra-relations of grime types as well as the inter-relations between grime types and other design defects, which will help further the understanding of how grime affects technical debt.

The current status of this research is in the framework development phase. Based on our previous work we have developed the underlying metrics, model, and parsing tools [20] and we have conducted initial research into the development of automated refactoring techniques [19]. We have also begun work focusing on the uncertainty of technical debt and other measures [27]. Presently, we are beginning work towards extracting design pattern realizations and pattern coupling information from the collection of open source projects and completing the grime taxonomy.

8. REFERENCES

[1] E. Alpaydin. 2010. *Introduction to Machine Learning* (2nd. ed.). The MIT Press, Cambridge, MA.

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

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

[4] J. Bosch. 1998. Design patterns as language constructs. *J. Object-Oriented Programming*. 11, 2, 18-32.

[5] L. C. Briand, J. Daly, and J. Wust. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*. 25, 1 (Jan.-Feb. 1999), 91-121. DOI= http://dx.doi.org/10.1109/32.748920.

[6] W. H. Brown, R. C. Malveau, H. W. McCornnick III, and T. J. Mowbray. 1998. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY.

[7] F. Buschmann, K. Henney, and D. Schimdt. 2007. *Pattern-oriented Software Architecture,* Vol. 5: *On Patterns and Pattern Language*. John Wiley & Sons, Inc., New York, NY.

[8] D. T. Campbell, J. C. Stanley, and N. L. Gage. 1963. *Experimental and Quasi-experimental Designs for Research*. Houghton Mifflin, Boston, MA.

[9] S. R. Chidamber and C. F. Kemerer. 1991. Towards a metrics suite for object oriented design. *SIGPLAN Not.* 26, 11 (November 1991), 197-211. DOI= http://doi.acm.org/10.1145/118014.117970.

[10] T. D. Cook, D. T. Campbell, and A. Day. 1979. *Quasi-experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin, Boston, MA.

[11] W. Cunningham. 1992. The WyCash portfolio management system. *SIGPLAN OOPS Mess.* 4, 2 (December 1992), 29-30. DOI= http://doi.acm.org/10.1145/157710.157715

[12] F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner. 2011. The quamoco tool chain for quality modeling and assessment. In *Proceedings of the 33rd International Conference on Software Engineering* (Honolulu, HI, USA, May 21-28). ICSE'11, 1007-1009. DOI=http://doi.acm.org/10.1145/1985793.1985977.

[13] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Software Engineering*. 27, 1 (Jan. 2001), 1-12. DOI=http://dx.doi.org/10.1109/32.895984.

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

[15] R. B. France, D. K. Kim, E. Song, and S. Ghosh. 2002. *Patterns as Precise Characterizations of Designs*. Technical Report. Colorado State University.

[16] E. Gagnon and L. Hendren. 1998. SableCC, an object-oriented compiler framework. In *Proceedings of the Technology of Object-Oriented Languages* (Santa Barbara, CA, USA, August 03-07, 1998). TOOLS USA'98. 140-154. DOI= http://dx.doi.org/10.1109/TOOLS.1998.711009.

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

[18] Gat and J. D. Heintz. 2011. From assessment to reduction: how cutter consortium helps rein in millions of dollars in

[19] Griffith, S. Wahl, and C. Izurieta. 2011. Evolution of legacy system comprehensibility through automated refactoring. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering* (Lawrence, Kansas, USA, November 12, 2011). MALETS '11. ACM, New York, NY, USA, 35-42. DOI= http://doi.acm.org/10.1145/2070821.2070826.

[20] Griffith, S. Wahl, and C. Izurieta. 2011. TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility. In *Proceedings of the of ISCA 24th International Conference on Computer Applications in Industry and Engineering* (Honolulu, HI, USA, November 16-18). CAINE'11.

[21] M. Grottke, R. Matias, and K. Trivedi. 2008. The fundamentals of software aging. In *Proceedings of the IEEE International Conference on Software Reliability Engineering Workshops* (Seattle, WA, USA, November 11-14, 2008). ISSRE Wksp 2008. 1-6. DOI= http://dx.doi.org/10.1109/ISSREW.2008.5355512.

[22] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. 1995. Software rejuvenation: analysis, module and applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing* (Pasadena, CA, USA, June 27-30, 1995). FTCS-25. 381-390. DOI= http://dx.doi.org/10.1109/FTCS.1995.466961.

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

[24] C. Izurieta and J. Bieman. 2007. How software designs decay: A pilot study of pattern evolution. In *Proceedings of the First Symposium on Empirical Software Engineering and Measurement* (Madrid, Spain, September 20-21, 2007). ESEM 2007. 449-451. DOI= http://dx.doi.org/10.1109/ESEM.2007.55.

[25] C. Izurieta and J. Bieman. 2008. Testing consequences of grime buildup in object oriented design patterns. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation* (Lillehammer, Norway, April 09-11, 2008). ICST 2008, 171-179. DOI= http://dx.doi.org/10.1109/ICST.2008.27.

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

[27] C. Izurieta, I. Griffith, D. Reimanis, R. Luhr. 2013. On the uncertainty of technical debt measurements. In *Proceedings of the 4th International Conference on Information Science and Applications* (Pattaya, Thailand, June 24-26, 2013).

ICISA 2013. IEEE Computer Society Press, Los Alamitos, CA, USA.

[28] C. Izurieta, A. Vetro, N. Zazworka, Y. Cai, C. Seaman, and F. Shull. Organizing the technical debt landscape. In *Proceedings of the Third International Workshop on Managing Technical Debt* (Zurich, Sweden, June 5, 2012). MTD'12. 23-26. DOI= http://dx.doi.org/10.1109/MTD.2012.6225995.

[29] K. Knoernschild. 2012. Java Application Architecture: Modularity Patterns with Examples Using OSGi. Pearson Education, Inc., Boston, MA.

[30] M. Lanza and R. Marinescu. 2006. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer-Verlag Berlin Heidelberg, Berlin, Germany.

[31] W. Li and S. Henry. 1993. Object-oriented metrics that predict maintainability. *J. Syst. and Sftwr*. 23, 2, 111-122.

[32] R. Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance* (Chicago, IL, USA, September 11-17, 2004). ICSM 2004. 350-359. DOI= http://dx.doi.org/10.1109/ICSM.2004.1357820.

[33] W. McNatt and J. Bieman. 2001. Coupling of design patterns: common practices and their benefits. In *Proceedings of the 25th Annual International Computer Software and Applications Conference* (Chicago, IL, USA, October 08-12, 2001). COMPSAC 2001. 574-579. DOI= http://dx.doi.org/10.1109/CMPSAC.2001.960670.

[34] N. Moha, Y. Guéhéneuc, L. Duchien, and A. F. Le Meur. 2009. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Engineering*. 36, 1, (Jan.-Feb. 2010), 20-36. DOI= http://dx.doi.org/10.1109/TSE.2009.50.

[35] N. Moha, Y. Guéhéneuc, and P. Leduc. 2006. Automatic generation of detection algorithms for design defects. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering* (Tokyo, Japan, September 18-26, 2006). ASE '06. 297-300. DOI= http://dx.doi.org/10.1109/ASE.2006.22.

[36] N. Moha, Y. Guéhéneuc, A. F. Le Meur, and L. Duchien. 2008. A domain analysis to specify design defects and generate detection algorithms. In *Fundamental Approaches to Software Engineering Lecture Notes in Computer Science*. 4961. 276-291. Springer Berlin / Heidelberg, Berlin, Germany. DOI= http://dx.doi.org/10.1007/978-3-540-78743-3_20.

[37] M. Munro. 2005. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Proceedings of the 11th IEEE International Symposium on Software Metrics* (Como, Italy, September 19-22, 2005). METRICS 2005. 15-15. DOI= http://dx.doi.org/10.1109/METRICS.2005.38.

[38] M. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin. 1999. Code decay analysis of legacy software through successive releases. In *Proceedings of the Aerospace Conference* (Aspen, CO, USA, March 7, 1999). AERO 1999. 5, 69-81. IEEE. DOI= http://dx.doi.org/10.1109/AERO.1999.794163.

[39] D. L. Parnas. 1994. Software aging. In *Proceedings of the 16th International Conference on Software Engineering* (Sorrento, Italy May 16-21, 1994). ICSE '94. 279-287. IEEE Computer Society Press, Los Alamitos, CA, USA.

[40] D. Posnett, C. Bird, and P. Dévanbu. 2011. An empirical study on the influence of pattern roles on change-proness. *J. Empirical Software Engineering*. 16, 3 (Jun. 2011), 396-423. Springer US. DOI= http://dx.doi.org/10.1007/s10664-010-9148-2.

[41] F. Ramsey and D. Schafer. 2002. *The Statistical Sleuth: A Course in Methods of Data Analysis* (2nd. ed.). Cengage Learning, Independence, KY.

[42] T. Schanz and C. Izurieta 2010. Object oriented design pattern decay: a taxonomy. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano-Bozen, Italy, September 16-17, 2010). ESEM '10. 7, 1-8. ACM, New York, NY, USA. DOI= http://doi.acm.org/10.1145/1852786.1852796.

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

[44] N. Subramanian and L. Chung. 2001. Metrics for software adaptability. In *Proceedings of Software Quality Management Conference* (Leicestershire, United Kingdom, April 18-20, 2001). SQM 2001.

[45] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The qualitas corpus: A curated collection of java code for empirical studies. In *Proceedings of the 17th Asia Pacific Software Engineering Conference* (Sydney, Australia, November 30 – December 3, 2010). APSEC 2010. 336-345. DOI= http://dx.doi.org/10.1109/APSEC.2010.46.

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

[47] K. Trivedi, K. Vaidyanathan, and K. Goseva-Popstojanova. 2000. Modeling and analysis of software aging and rejuvenation. In *Proceedings of the 33rd Annual Simulation Symposium* (Washington, DC, USA, April 16-20, 2000). SS 2000. 270-279. DOI= http://dx.doi.org/10.1109/SIMSYM.2000.844925.

[48] S. Wong, Y. Cai, M. Kim, and M. Dalton. 2011. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering* (Honolulu, HI, USA, May 21-28). ICSE'11, 411-420. DOI= http://doi.acm.org/10.1145/1985793.1985850.

[49] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull. 2013. Comparing four approaches for technical debt identification. Software Quality Journal. (Apr. 2013). 1-24. Springer US. DOI= http://dx.doi.org/10.1007/s11219-013-9200-8.