Behavioral Evolution of Design Patterns: Understanding Software Reuse Through the Evolution of Pattern Behavior

Derek Reimanis(✉) and Clemente Izurieta

Montana State University, Bozeman, MT 59718, USA
derek.reimanis@msu.montana.edu, clemente.izurieta@montana.edu

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

Keywords: Software evolution · Software quality assurance · Design patterns · Software reuse

1 Introduction

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

The deployment of complex products with multiple components does not come without its drawbacks, however. The expectation that multi-component complex systems are delivered on time and within budget, require the adoption of robust processes to accommodate all phases of the product's software

X. Peng et al. (Eds.): ICSR 2019, LNCS 11602, pp. 77–93, 2019.
https://doi.org/10.1007/978-3-030-22888-0_6

life-cycle. One such process is software quality assurance (QA); which seeks to measure and monitor all aspects of software quality over the entire lifetime of a software solution. Software design represents the vision of a software solution, considering current and potential future requirements. Designs must be flexible enough to accommodate change, facilitate extensibility, and promote the ease of interchangeable and reusable software components, while still maintaining a high level of quality. A common strategy employed to assist with this balance is the use of design patterns, which can act as reusable design-level and knowledge-share software components among developers [14].

Design patterns embody recurring and reusable solutions to common problems encountered in the software development process [7]. Design patterns capture experience reuse and represent decisions that are made in the design phase of a software life-cycle. They have the properties of being reusable, maintainable, and easy to extend in future versions. The choice to utilize design patterns in a project comes with the understanding of an important assumption– specifically that the initial implementation of a pattern instance may take longer than a non-pattern implementation, but future revisions and maintenance efforts will be faster and therefore cheaper if a pattern is present. This assumption holds true in a theoretical sense, yet is controversial in a practical sense. Historically, design pattern realizations have been found to deviate or drift from their initial and pure intent, thus eliminating many of the beneficial qualities the pattern offers in the first place. Such a deviation may occur if a new developer is unfamiliar with a code-base, or if pressure from management to ship a product requires 'quick-and-dirty' extensions of the pattern. The existence and extent of such deviations are not fully explored; for example, it is not known whether the presence of such a deviation within a design pattern provides more harm to a software product than choosing not to utilize a design pattern in the first place.

1.1 Research Problem

With the understanding that design patterns offer reusable solutions to common problems in software development [7], the importance of verifying correctness of design patterns is crucial. A design pattern instance that deviates from its specification loses many of its reusable qualities, meaning the pattern instance can no longer be applied as a reuse mechanism. Previous research efforts have both explored the existence and measured the effects of design pattern deviations only from a structural perspective. The structural perspective of a design pattern refers to the class members of the pattern, including the operations and attributes of the pattern's classes, as well as the relationships between class members. This research has found that such deviations do exist within a design pattern's evolution, and that these deviations have a negative effect on software quality. However, the structural perspective is one of many perspectives into a design pattern. Another perspective used to understand design patterns is the behavioral perspective, or the events that occur as a design pattern instance is operating at program run-time, which are not visible from a structural perspective. A behavioral perspective offers additional insights into a design pattern and

its evolution, thus refining existing scientific models and taxonomies [8,19] that capture design pattern evolution.

1.2 Research Objective

The goal of this research is to expand the body of knowledge surrounding software reuse, as it pertains to design pattern evolution, from a behavioral perspective. Three specific activities aligned with our overarching goal are identified. First, the identification of design pattern deviations from a behavioral perspective. Second, the characterization of behavioral deviations into a structured organizational scheme, a taxonomy. Third, the evaluation of the effects of behavioral deviations on existing structural models as well as software quality. Meeting these objectives will complement existing structural approaches, and provide software stakeholders with more advanced techniques and tools to monitor software quality, so that important decisions surrounding a software product, specifically reuse of components, can be made with increased certainty.

1.3 Contributions

The contributions of this work are threefold:

- A taxonomy that captures behavioral grime in design pattern instances.
- Evaluation of the relationships between structural grime and behavioral grime.
- Evaluation of the relationships between behavioral grime and pattern quality.

2 Background and Related Work

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

2.1 Design Pattern Formalization

Design patterns can be formally specified using a combination of the Role-Based Meta-Modeling Language (RBML) [13] and the Object Constraint Language (OCL) [21]. RBML specializes the Unified Modeling Language (UML) [18] metamodel and captures key elements of a design pattern, based on specific roles that participants in that design pattern may take. A design pattern specification consists of two sub-specifications, the Structural Pattern Specification (SPS) and the Interaction Pattern Specification (IPS) [13]. An SPS characterizes the structural elements of a pattern, including the class members, attributes, operation signatures, and relationships. An IPS characterizes the behavioral elements of a pattern, referring to the flow of information that occurs as a design pattern is in

operation, at program run-time. SPSes are analogous to UML class diagrams, whereas IPSes are analogous to UML sequence diagrams. Both SPSes and IPSes exist at a meta-level that describes design specifications, which is referred to as the M2 level [6]. A given pattern instance, as implemented in a software project, exists at the design level, which is referred to as the M1 level. The process of checking conformance for a pattern instance entails mapping the pattern's members that exist at its M1 level implementation to its corresponding pattern roles, captured with an SPS and IPS, at its respective M2 level pattern definition.

2.2 Design Pattern Decay

Software applications are used everyday, yet they do not 'wear out' over extended use periods in the classical sense, as physical objects would. Instead, software is subject to a different type of wear, related to the maintenance of the underlying design and code. Over time, many factors such as unforeseen changing requirements, developer turnover, legacy code dependencies, and others, will contribute to the degradation of software quality. This phenomenon is captured by the terms *software decay* and *code decay*. Software and code are deemed *decayed* if they are *harder to change than they should be* [3]. A specific form of software decay is *design pattern decay*. Design pattern decay refers to the addition of undesired elements or loss of desired elements in a design pattern pattern instance, over the lifetime of the design pattern [9,10]. Design pattern decay is considered a sub-domain of design decay, which is analogous to code decay with the exception that the decay occurs in the design level of a software project instead of the code level. Design pattern decay consists of two categories; *design pattern grime* and *design pattern rot* [9]. Design pattern grime, hereafter referred to as *grime*, is defined as the build-up of unintended artifacts over the lifetime of a design pattern instance. These artifacts do not contribute to the pattern's intended role in the overall software project, detracting away many of the beneficial qualities the pattern would otherwise provide. Previous work has shown that the presence of grime is associated with decreases in testability and adaptability, as well as the presence of anti-patterns [11]. Additionally, recent work has shown that the presence of grime is related to the depreciation of system correctness, system performance, and system security [4]. Furthermore, Feitosa et al. has found that grime has a tendency to accumulate linearly, suggesting the quality of a pattern worsens as the grime of that pattern increases [5]. Design pattern rot, hereafter referred to as *rot*, is defined as the removal of key elements of the pattern such that the pattern no longer retains its core elements. A pattern that has succumbed to rot no longer identifies as such; instances of rot in software projects has eluded researchers because of the difficulty in identifying it.

3 Research Approach

In an effort to expand on software reuse, as it pertains to design pattern evolution from a behavioral perspective, the strategy employed in this research has three-steps; first, the identification or detection of unintended behavioral items, as they

appear in the code of design pattern instances. Second, the characterization of unintended behavioral items into categories that simplify the remediation effort. Third, the measurement of severity of unintended behavioral items so that remediation efforts can be prioritized.

3.1 GQM

We use Basili's Goal-Question-Metric (GQM) approach [1] as a guide for this research. The GQM approach provides an outline of high-level research goals (RG) supplemented with questions (RQ) and metrics (M) that guide the research. The GQM for this research is listed below:

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

RQ1 How does the behavior of a design pattern instance deviate from the expected behavior of that pattern type?
RQ2 Is there evidence to suggest that behavioral grime is present in pattern instances of a single pattern type?
RQ3 Is there evidence to suggest that behavioral grime is present in pattern instances across different pattern types?
RQ4 To what extent can a pattern instance have both structural and behavioral grime?
RQ5 What is the relationship between structural and behavioral grime?

RG2: *Quantify* the impact of behavioral grime *for the purpose* of capturing the effect of behavioral grime on patterns *with respect to* proper pattern behavior as defined by the design pattern's specification *from the perspective of* the software system *in the context of* design patterns in open source software systems.

RQ6 What is the relationship between behavioral grime and design pattern quality, in terms of pattern integrity and pattern instability?
RQ7 Is the size of a design pattern instance related to the amount of behavioral grime in that pattern instance?

Metrics: Several metrics are outlined that aid in answering the questions, which are described in Table 1. Formulations for each metric are given, with respect to a pattern instance P.

3.2 Study Design

The study design for this research is depicted in Fig. 1. To begin, we selected several software projects to study according to the selection process presented in the paragraph below. From these software projects, we identified design pattern instances using the design pattern detection tool developed by Tsantalis

Table 1. Description of the metrics used in this study

Metric name	Description
Structural Conformance (**M1**)	The percentage of structural roles in P that conform to at least one structural role from P's SPS
Behavioral Conformance (**M2**)	The percentage of behavioral roles in P that conform to at least one behavioral role from P's IPS
Structural Grime Count (**M3**)	A count of the number of unique instances of structural pattern grime in P
Behavioral Grime Count (**M4**)	A count of the number of unique instances of behavioral pattern grime in P
Pattern Integrity (**M5**)	$\frac{M1+M2}{2}$
Pattern Instability (**M6**)	Adopted from Martin's Instability metric (I) [16], the efferent coupling of P divided by the sum of the efferent coupling of P and the afferent coupling of P $\frac{C_e(P)}{C_e(P)+C_a(P)}$
Pattern Size (**M7**)	Adopted from Li and Henry's Size2 metric (*size2*) [15], the sum of attributes and methods across all classes in P

et al. in [20]. We chose this tool because it is based on strong theory and provides evidence of little to no false positives in practice. Additionally, we used the tool SrcML [2] to assist in the source code parsing process. We chose this tool because it offers a translation from language-specific source code to standard format XML, meaning this process becomes language-agnostic. Following XML generation, we reverse-engineered the UML class and sequence diagrams of the entire software project. Once we had reverse-engineered the UML class and sequence diagrams, we generated a UML representation of the design pattern by combining the design pattern's detection with the corresponding UML diagrams. Next, we subjected each design pattern instance to a process of coalescence. The process of pattern coalescence involves identifying members of the design pattern not captured by the design pattern detection tool. Such members may be subclasses, super-classes, or pattern-methods within a pattern class that the design pattern detection tool may have missed. Following coalescence, we extracted the evolution of each pattern instance by tracking and connecting contributing roles of patterns across software versions. Once pattern instance evolutions were generated, we entered the evaluation stage wherein we evaluated pattern conformance, pattern grime, and pattern quality/size for each version (pattern instance) in the pattern instance evolution.

The process of selecting experimental units, or software projects, is as follows. In an effort to increase generalizability of results, we chose to analyze five projects in total. To ensure relevancy, projects were selected based on their popularity ranking on the online code repository GitHub[1]. Specifically, we ranked

[1] www.github.com.

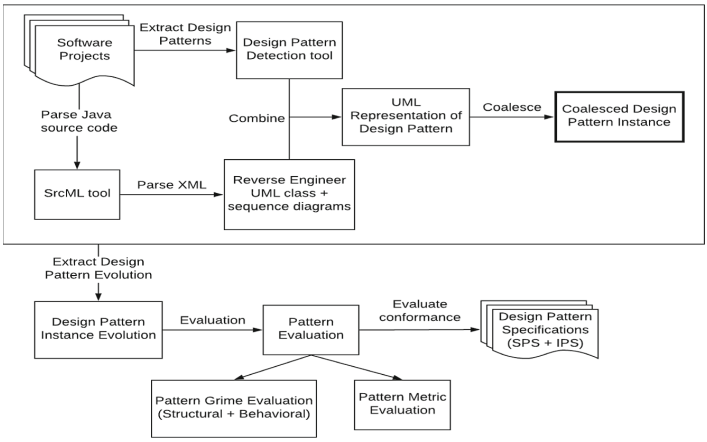

Fig. 1. Summary of study design. Design pattern instances are extracted from software projects, and the associated UML is reverse-engineered from source code. The evolution of each pattern instance is generated, and evaluations for conformance, grime, and metrics are found across each pattern instance evolution.

Table 2. Demographics of the projects under analysis

Project name	Domain	Releases (total releases)	Release dates
Selenium	Testing framework	3.0–3.141.59 (20)	Oct 2016–Nov 2018
RxJava	Asynchronous streaming	2.0–2.2.7 (20)	Oct 2016–Feb 2019
guava	Java libraries	9.0–27.1 (20)	Apr 2011–Mar 2019
spring-boot	Java packaging framework	1.0–2.1.3 (20)	Apr 2014–Feb 2019
Hystrix	Fault tolerance library	1.0.2–1.5.18 (20)	Nov 2012–Nov 2018

all projects according to their 'number of stars', which is synonymous with a favorite or bookmark, and selected the first five projects such that each project had at least 2,000 commits, 75 releases, and 100 unique contributors. In most cases, all projects had significantly more than the minimum required filters; for example the selenium project features 23,550 commits, 116 releases, and 424 contributors. From each project, we selected the 20 most recent minor releases evenly divided between most recent minor release and most recent major release, under the assumption that the project follows traditional notation for release numbers, which is: [major.minor.bug_fix]. If a project did not have at least 20 minor releases in major release window, we selected minor releases from the next-major release. We utilized this process to generate an even spread of data points between the most recent release and the last major release. The outcome from this project selection process is presented in Table 2, along with the release numbers and respective release dates.

Due to the exploratory nature of our study, we chose to focus our analysis on four pattern types; the Singleton pattern from the 'Creational' category [7], the Object-Adapter pattern from the 'Structural' category [7], and the State and Template Method patterns from the 'Behavioral' category [7]. Our initial intuition is that patterns in the behavioral category may be more prone to behavioral deviations, so we selected two pattern types from that category. Additionally, these four pattern types provided us the largest sample size of detected pattern instances; many projects featured zero pattern instances of certain types, such as the Visitor or Observer pattern. The count of pattern instance evolutions for each pattern type and across each project under analysis is shown in Table 3. Note this is a count of pattern instance evolutions, not pattern instances; the difference being pattern instance evolutions track a single pattern instance across multiple versions, while pattern instances refer to a single pattern instance at a single software version.

Table 3. Count of pattern instance evolutions for each of the projects under analysis

Project name	Singleton evolutions	Object-Adapter evolutions	State evolutions	Template method evolutions
Selenium	9	21	28	9
RxJava	5	27	124	11
guava	44	9	34	103
spring-boot	13	4	10	17
Hystrix	14	0	5	5
Total	85	61	201	145

4 Results

Behavioral evaluations of pattern grime have, to the best our knowledge, not been explored in the literature. This allows us to make use of exploratory techniques when reviewing our findings. We thus, utilize correlation analyses and linear regression approaches to identify potential relationships between variables, and will reserve causative analysis techniques for future experiments when research hypotheses are identified.

RQ1: To answer this research question, which is concerned with identifying how the behavior of a design pattern instance can deviate from the expected behavior of that pattern type, we performed an *in-vitro* experiment [12]. Specifically, we created an instance of the Observer pattern that perfectly aligns to its SPS and IPS. Such an instance might be impractical in the real-world, yet would mark a starting/calibration point for experiments. We injected code into

this Observer pattern instance that constitutes modular structural grime, as presented by Schanz and Izurieta [19]. Modular structural grime is concerned with the relationships that pattern members may have with either other pattern members, or non-pattern members. Therefore, modular structural grime provides a constraint on all possible pattern behaviors. In other words, a given behavior, whether between pattern members or non-pattern members, cannot exist unless the two members share a structural relationship. To each injected modular grime instance, we applied the behavioral deviations as presented by Reimanis and Izurieta [17]. Specifically, these deviations are 'Improper Order of Sequences', in which expected behaviors occur in an incorrect order, and 'Excessive Actions' in which excessive actions hamper the run-time expectations of a pattern. For this work, we chose to focus on a subset of Excessive Actions, which we refer to as 'Repetitive Actions', or cases where the same behavior is performed within the same scope, or function call, of a pattern instance at run-time. After applying said behavioral deviations to the modular grime taxonomy, we generated a taxonomy of behavioral grime, which is shown in Fig. 2.

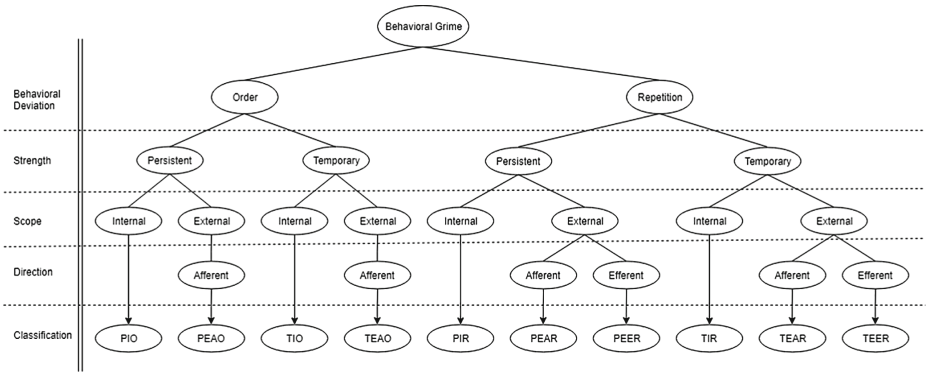

Fig. 2. Behavioral grime taxonomy. Dimensions of behavioral grime are listed on the left, and corresponding characterizations are shown in the taxonomy tree.

The dimensions for this taxonomy are mirrored from the modular grime taxonomy [19], which are explained as follows. Strength refers to the strength of a relationship between two UML members; Persistent Strength refers to a UML association while Temporary Strength refers to a UML use-dependency. Scope refers to the context of the relationship between two UML members; Internal Scope refers to a relationship between two pattern members, and External Scope refers to a relationship between one pattern member and one non-pattern member. Direction refers to the direction of the relationships. Afferent Direction refers to an incoming relationship while Efferent Direction referring to an outgoing relationship. In the taxonomy, the Classification row refers to the acronym that captures that type of behavioral grime; for example, the PIO classification is an acronym for 'Persistent-Internal-Order' grime. This behavioral grime taxonomy

closely mirrors the modular grime taxonomy presented in [19], with two exceptions. First, we have incorporated the 'Behavioral Deviations' dimension, which corresponds to the type of behavioral grime (Order or Repetition). Second, one will notice that the taxonomy is not symmetrical across Order and Repetition sub-trees; specifically, the sub-tree pertaining to External Efferent Order (-EEO) type grime is non-existent. This is because this sub-tree represents an outgoing relationship from a pattern member to a non-pattern member can not be in an incorrect order; such relationships are not captured by the design pattern, and thus cannot be in an incorrect order.

RQ2, RQ3: **RQ2** and **RQ3** are concerned with identifying behavioral grime within and across multiple pattern instances. To answer these questions, consider Table 4, which summarizes the grime counts found from our analysis. Each cell in the table refers to a count of behavioral grime across all patterns instances of the corresponding pattern type. Note that no instances of Order grime were found across the entire analysis, and thus we will refrain from showing Order grime results. This does not imply that Order grime does not exist, but rather it means we failed to detect any in this study.

Table 4. Count of behavioral grime across each pattern instance

Behavioral grime type	Singleton	Object-Adapter	State	Template method
PIR	0	296	645	15
PEAR	0	2028	377	60
PEER	390	583	896	392
TIR	24	229	842	266
TEAR	0	4289	921	153
TEER	2088	6822	10320	3053

RQ4, RQ5: These research questions are concerned with identifying the potential relationship between structural and behavioral grime. To answer these questions, we began by generating a pairwise scatter-plot for each type of structural and behavioral grime, which is shown in Fig. 3. Structural grime is shown on the x-axis, and Repetition behavioral grime is shown on the y-axis. Points in the scatter-plot represent the count of modular grime and repetition grime for a single pattern instance.

RQ6: This research question is concerned with the relationship between behavioral grime and pattern quality, as measured by our surrogate quality-metrics, Pattern Instability (M5) and Pattern Integrity (M6). Similarly to **RQ4** and **RQ5**, we began by generating pairwise scatter-plots for these metrics for each behavioral grime type to visually assess trends. This scatter-plot is shown in Fig. 4.

Fig. 3. Pairwise scatter-plots illustrating the relationships between structural grime, shown on the x-axis, and behavioral grime, shown on the y-axis.

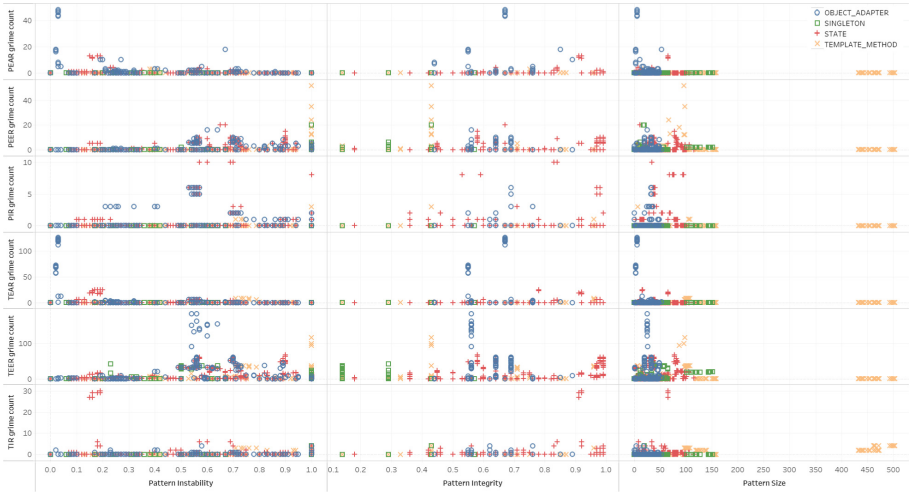

Fig. 4. Pairwise scatter-plots of pattern quality, measured via surrogate metrics Pattern Instability and Pattern Integrity, pattern size, and behavioral grime.

RQ7: This research question is concerned with identifying if the size of a design pattern instance is related to the amount of behavioral grime in that pattern instance. Similarly to the previous research questions, we began by generating a scatter-plot to visually assess the data. This scatter-plot is shown in Fig. 4.

Table 5. Correlation coefficients (r-values) and corresponding p-values for each pairwise metric (pattern quality, pattern size, and grime type). In each cell, coefficients are presented first and p-values are presented second. Bold values represent strong relationships, $r > 0.60$ or $r < -0.60$ and statistically significant p-values at the $\alpha < 0.05$ level. Separations within the table refer to separate research questions.

	PEAR	PEER	PIR	TEAR	TEER	TIR
PEA	0.2021/**0.00**	0.0002/0.99	0.0153/0.19	0.2132/**0.00**	0.0324/**0.01**	0.0694/**0.00**
PEE	0.0358/**0.00**	0.3339/**0.00**	0.2095/**0.00**	0.0704/**0.00**	**0.5814**/**0.00**	0.1285/**0.00**
PI	−0.0084/0.48	0.0352/**0.00**	0.0264/**0.03**	−0.0006/0.96	0.4053/**0.00**	0.0764/**0.00**
TEA	**0.6086**/**0.00**	0.1430/**0.00**	0.1355/**0.00**	0.5781/**0.00**	0.2050/**0.00**	0.3026/**0.00**
TEE	0.0006/0.96	0.2227/**0.00**	0.2018/**0.00**	0.0225/0.06	**0.6763**/**0.00**	0.0549/**0.00**
TI	0.0762/**0.00**	0.3547/**0.00**	0.3702/**0.00**	0.0612/**0.00**	0.5374/**0.00**	0.2633/**0.00**
Pattern Instability	−0.1888/**0.00**	0.0255/**0.03**	−0.0100/0.40	−0.1659/**0.00**	−0.0445/**0.00**	−0.0300/**0.01**
Pattern Integrity	0.1555/**0.00**	0.1470/**0.00**	0.2206/**0.00**	0.1179/**0.00**	0.2204/**0.00**	0.2119/**0.00**
Size2	−0.0118/0.32	0.0810/**0.00**	0.0951/**0.00**	−0.0117/0.32	0.0932/**0.00**	0.2341/**0.00**

To assess the strength of each relationship in **RQ4-7**, we calculated pairwise correlation coefficients and corresponding p-values. The nature of our data is a count, which falls under the ratio numeric scale, and a visual assessment of the scatter-plots suggests a linear relationship; therefore we chose to use Pearson's method to calculate correlation coefficients and generate p-values. The application of Pearson's requires addressing two primary assumptions; the normality assumption and the independence assumption. We may say we have satisfied the normality assumption because an advantage of using Person's is that the normality assumption is not applicable for larger sample sizes, of which our data is. However, we cannot say we have satisfied the independence assumption. Specifically, each data point comes from a single pattern instance in a single software version, and pattern instances may appear in more than one software version, meaning grime in a future version might be, and likely is, dependent on grime in previous versions. We alleviate this concern because of the number of pattern instance evolutions we have detected, which is captured in Table 3, but we cannot say we have satisfied the independence assumption. Regardless, the correlation coefficients and corresponding p-value for each pairwise metric across **RQ4-7** is listed in Table 5, with strong relationships ($r > 0.60$ or $r < -0.60$) and statistically significant p-values at the $\alpha < 0.05$ level shown in bold. Each p-value corresponds to the probability that the correlation coefficient we received was not due to chance, under the assumption that the true correlation coefficient is zero, which implies a very weak relationship.

5 Discussion

The following discussion points highlight the significance of design pattern research as knowledge communication artifacts in software reuse. We extend the body of knowledge by categorizing and evaluating behavioral grime through the exploration of selected pattern evolutions.

The first series of statistical tests focused on understanding the relationships between structural and behavioral grime. For nearly all pairwise comparisons between structural and behavioral grime, very low p-values were found, suggesting that the results we received were not due to chance, and that no relationship exists between structural and behavioral grime types. This is an interesting result, because comparing the correlation coefficients from Table 5 to the respective p-values in Table 5, many correlation coefficients are quite small, which would normally suggest a higher p-value. However, because our sample sizes were large, we found a correlation coefficient that, while being non-zero in many cases, was based upon enough data to supply a confident statistical estimate. This means that the correlation coefficient estimates we found may be close to the true value of the correlation coefficients, but more experiments need to be performed to confirm this position.

With respect to relationships between structural and behavioral grime, specifically of interest is the behavioral grime type TEER (Temporary External Efferent Repetition). Grime of this type manifests itself as non-pattern members that are used by a pattern, but only as a use-dependency (not an association). Such items constitute a deviation from a pattern's specification, which imply the pattern implementation is more difficult to reuse in the future. TEER grime was moderately correlated with two structural grime types (PI and TI), yet was also strongly correlated with two structural grime types (PEE and TEE). The correlations with PEE and TEE do not come as a surprise, considering the structural forms of that grime type dictate behavioral allowances. In other words, the presence of TEER grime cannot exist without the presence of one of either PEE or TEE. However, recall that TEER grime is specifically a *repetition* of behaviors. This means that while a pattern instance may have PEE and TEE structural grime, manifested as a relationship between pattern members and non-pattern members, the relationship is called upon more than once within the scope of a pattern's operation. Conceivably, these usages could originate from poorly constructed logical flows within code, in which the same logical call, or operation, might be performed at different points in a single operation. To assert this thought, we manually reviewed the code of one state pattern instance and discovered that the instance was setting the same state at multiple different places, all within the same operation. This practice is not strictly discussed in the State pattern's best practices, but certainly a cleaner and more reusable version of the pattern instance would be one in which the state would be set once per operation. While future research is required to reveal the true effects of such a practice, such revelations illustrate why behavioral deviations are an important topic to study.

The second series of statistical tests focused on capturing the relationships between behavioral grime and pattern quality, with respect to our surrogate metrics, Pattern Instability and Pattern Integrity. Nearly all pairwise p-values reported as very low, suggesting we reject the possibility that no relationship exists between behavioral grime types and pattern quality. Interestingly, the correlation coefficients for Pattern Instability are low and negative, hinting that a weak but present inverse relationship exists between Pattern Instability and behavioral grime. In other words, an increase in behavioral grime is associated with one, or both, of the following: A pattern instance's efferent coupling decreases while its afferent coupling does not decrease, or a pattern instance's afferent coupling increases faster than its efferent coupling. In most cases, the size of a pattern instance always increased over its evolution, suggesting that the second option holds true; that afferent coupling increases faster than efferent coupling. Put another way, as pattern instances aged and evolved, they tended to be used more by non-pattern members, not that they made use of more non-pattern members. In these cases, results suggest that behavioral grime increases as well; regardless, the increased coupling between non-pattern members and pattern members inhibits future pattern reuse.

Our results pertaining to Pattern Integrity are seemingly counter-intuitive; the results suggest that as Pattern Integrity increases, i.e., as a pattern instance more closely follows its specification, the amount of behavioral grime within that pattern instance increases as well. One would envision that behavioral grime would decreases as Pattern Integrity increases, because that would suggest a refactoring of said pattern instance, aligning it more closely with pattern standards. However, two likely explanations are plausible. First, the case could be that the pattern instance is evolving and new pattern members are being added that conform to the respective SPS and IPS, yet these new pattern members contain behavioral grime. Second, the case could be that refactorings are being performed that better align existing pattern members to their SPS and IPS, yet the refactorings introduce more behavioral grime. In either case, a more robust and extensive study is required to solve this conundrum.

The third series of statistical tests focused on finding the relationship between behavioral grime and pattern size. Our expectations were that as a pattern instance evolved and grew, it would also gain more behavioral grime. Behavioral grime types PEAR and TEAR reported relatively large p-values, suggesting that we are unable to assume that no relationship exists between PEAR/TEAR and pattern size. However, the other types of behavioral grime reported very small p-values, suggesting we can reject the null, and that evidence exists to support a relationship between behavioral grime and pattern size. Looking at the correlation coefficients, we see small positive coefficient values, strengthening our initial expectations. While these values are not as large as expected, we can claim that the evidence from this study suggests pattern instances gain behavioral grime as they get larger. While the increasing size of a pattern instance over its evolution is indicative that the pattern is being reused, the presence of behavioral grime may imply that the pattern's growth rates are slowing down. Future work

will address this question, looking at the growth rates of behavioral grime as they pertain to pattern size.

6 Threats to Validity

There are several design and implementation considerations in this study that threaten the validity of the results. External validity is concerned with the generalization of results. In this study, we limited ourselves to 20 minor-release versions of five Java projects, chosen based on popularity from the online repository GitHub. While we attempted to systematically select projects so that our results would be generalizable, we can only claim that our results hold true for the projects under analysis. More case studies following this same process are necessary before more general claims can be made. Internal validity refers to the ability to reach causal conclusions based on the study design. Internal validity is minimal in this study because we make no causal claims, just correlations. Future studies will be directed at increasing this body of knowledge, thus we will explore causal links, yet for this study only correlations were used. Construct validity refers to the choice of independent and dependent variables, with respect to conclusion. Construct validity is threatened in our study because of our use of the Pattern Integrity and Pattern Instability metrics as surrogate metrics for pattern quality. Our rationale for choosing these two surrogate metrics comes from theory that suggests a very small value for Instability increases system stability, positively affecting quality, and that high values for Integrity correspond to more standard and robust implementations, also positively affecting quality.

7 Conclusion

Our research goals focused on the exploration and initial understandings of behavioral deviations, as they pertain to design pattern evolution and software reuse. To this end we have constructed a taxonomy that classifies behavioral grime types. Furthermore, we designed and implemented a case study wherein we measured counts of structural and behavioral grime, as well as quality and size, across pattern instance evolutions pertaining to four design pattern types, originating from 20 versions of five open source software projects. We evaluated the relationships between structural and behavioral grime and found statistically significant cases of strong correlations between specific types of structural and behavioral grime. We identified statistically significant relationships between behavioral grime and both choice quality metrics, as well as pattern size. Patterns are a means of knowledge communication through the reuse of common solutions, and these findings provide important directions that can help practitioners in reducing problems encountered through the evolution of software components.

References

1. Caldiera, V.R.B.G., Rombach, H.D.: The goal question metric approach. Encycl. Softw. Eng., 528–532 (1994)
2. Collard, M.L.: Addressing source code using SrcML. In: IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension (IWPC 2005). Citeseer (2005)
3. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? Assessing the evidence from change management data. IEEE Trans. Softw. Eng. **27**(1), 1–12 (2001)
4. Feitosa, D., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y.: Correlating pattern grime and quality attributes. IEEE Access **6**, 23065–23078 (2018)
5. Feitosa, D., Avgeriou, P., Ampatzoglou, A., Nakagawa, E.Y.: The evolution of design pattern grime: an industrial case study. In: Felderer, M., Méndez Fernández, D., Turhan, B., Kalinowski, M., Sarro, F., Winkler, D. (eds.) PROFES 2017. LNCS, vol. 10611, pp. 165–181. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69926-4_13
6. France, R., Kim, D., Song, E., Ghosh, S.: Metarole-based modeling language (RBML) specification v1. 0. Technical report 02-106, Computer Science Department, Colorado State (2002)
7. Gamma, E.: Design Patterns: Elements of Reusable Object-oriented Software. Pearson Education India, Noida (1995)
8. Griffith, I., Izurieta, C.: Design pattern decay: the case for class grime. In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, p. 39. ACM (2014)
9. Izurieta, C.: Decay and Grime Buildup in Evolving Object Oriented Design Patterns. Colorado State University, Fort Collins (2009)
10. Izurieta, C., Bieman, J.M.: How software designs decay: a pilot study of pattern evolution. In: First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pp. 449–451. IEEE (2007)
11. Izurieta, C., Bieman, J.M.: Testing consequences of grime buildup in object oriented design patterns. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 171–179. IEEE (2008)
12. Juristo, N., Moreno, A.M.: Basics of Software Engineering Experimentation. Springer, Heidelberg (2013)
13. Kim, D.K.: A meta-modeling approach to specifying patterns. Ph.D. thesis, Colorado State University. Libraries (2004)
14. Lajoie, R., Keller, R.K.: Design and reuse in object-oriented frameworks: patterns, contracts, and motifs in concert. In: Object-Oriented Technology for Database and Software Systems, pp. 295–312. World Scientific (1995)
15. Li, W., Henry, S.: Maintenance metrics for the object oriented paradigm. In: 1993 Proceedings First International Software Metrics Symposium, pp. 52–60. IEEE (1993)
16. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, Upper Saddle River (2002)
17. Reimanis, D., Izurieta, C.: Towards assessing the technical debt of undesired software behaviors in design patterns. In: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD), pp. 24–27. IEEE (2016)
18. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual. Pearson Higher Education, New Delhi (2004)

19. Schanz, T., Izurieta, C.: Object oriented design pattern decay: a taxonomy. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, p. 7. ACM (2010)
20. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. **32**(11), 896–909 (2006)
21. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Object Technology Series (1998)