Design Pattern Decay: The Case for Class Grime

Isaac Griffith and Clemente Izurieta
Department of Computer Science
Montana State University
PO Box 173880
+1 (406) 994-4780
isaac.griffith@msu.montana.edu
clemente.izurieta@cs.montana.edu

ABSTRACT

Context: We investigate class grime, a form of design pattern decay, wherein classes of the pattern realization have extraneous attributes or methods, which obfuscate the intended design of a pattern. **Goal**: To expand the taxonomy of class grime using properties of class cohesion. Using this expanded taxonomy we explore the effect that forms of class grime have on pattern realization understandability. **Method**: A pilot study utilizing a formal experiment to explore the effects of class grime on design pattern understandability. The experiments used simulated injection of 8 types of class grime into design pattern realizations randomly selected from 16 design pattern types from a set of 6541 realizations from 520 distinct software systems. **Results**: We found that for each of the 8 identified class grime forms, understandability was negatively affected. **Conclusion**: This work serves as early communication of research for the validation of the extended taxonomy as well as the method of grime injection used in the experiment.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design – *Design Concepts, Object-oriented design methods;* D.2.11 [**Software Engineering**]: Software Architectures – *patterns.*

General Terms

Measurement, Design, Experimentation.

Keywords

Software Architectures, Object Oriented Design Patterns, Software Decay.

1. INTRODUCTION

Design patterns [8] over the last two decades have reached wide spread use in the software engineering community. Yet even with such a wide spread adoption and well-studied implementation practices, design pattern realizations are not immune to decay over their evolution [9-12]. The decay of pattern realizations, specifically design pattern grime, involves the obfuscation or deviation of pattern structure and behavior from the original intent [10]. In this paper, we are concerned with the effects of class grime buildup on design pattern understandability (a measure of how easy it is to learn and comprehend the design of a software system [2]).

In order to evaluate this phenomena we have extended the existing grime taxonomy [14] to include new forms of class grime. Within this taxonomy we must first verify whether such forms of grime constitute discernible disharmonies. Thus, our questions of interest are as follows: *i)* Is there a difference between how types of class grime affect design pattern understandability? *ii)* Is a difference in the mean change in understandability due to each subtype of class grime? and *iii)* Is there a difference between the classifications of class grime: scope, context, and strength?

This paper is organized as follows: Section 2 explores the background concepts and related work; Section 3 describes class grime, proposes an extended grime taxonomy, and defines the types of grime to be used in the experimental process; Section 4 details the experimental design and underlying method; Section 5 covers the analysis and discusses the results from the experiments; Finally, section 6 concludes the paper and presents paths for future research.

2. BACKGROUND AND RELATED WORK

Software decay is a form of software evolution wherein a system evolves such that it becomes “harder to change than it should be” [6]. Izurieta and Bieman [10] identified two new forms of software decay, involving design patterns: *design pattern grime* and *design pattern rot*. Empirical studies have only confirmed the existence of grime. Initially, grime was divided into three disjoint categories: *class*, *modular*, and *organizational* (see type level of Figure 1) [10].

Seminal work by Izurieta [9] showed 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 [11]. Schanz and Izurieta [14] further expanded the taxonomy of modular grime into six disjoint types of grime. They conducted empirical studies across

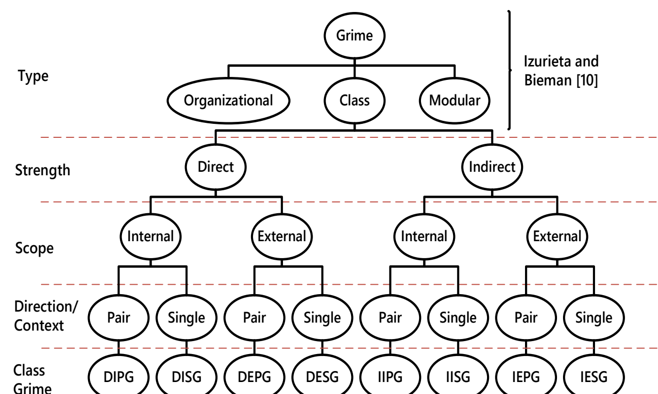

Figure 1. The extended class grime taxonomy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEM'14, September 18–19, 2014, Torino, Italy.
Copyright 2014 ACM 978-1-4503-2774-9/14/09...$15.00.
http://dx.doi.org/10.1145/2652524.2652570

open source systems to validate and refine this extended taxonomy. Further studies on grime have shown implications of grime on the testability of a system [12]. Although evidence for class grime has been inconclusive to date [11], we believe that this was due to a less than refined definition of class grime, which the research herein is a step towards validating.

3. CLASS GRIME TAXONOMY

The object in this study is class grime. Class grime is divided into eight specific subtypes using properties of class cohesion to define three categories: strength (direct or indirect), scope (internal or external), and context (singular or pair), forming the taxonomy depicted in Figure 1.

3.1 Class Cohesion

Cohesion is used to describe how well constructed a class is [4]. The higher the cohesion of a class the closer aligned its internal components are towards a common goal. In design pattern realizations, the classes should represent individual responsibilities of the pattern and if the specification is implemented correctly each class should have high cohesion, thus cohesion provides a basis to determine whether a design pattern realization's classes have been afflicted with class grime.

3.1.1 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 2, 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 more flexible and weaker relationship between the method and attribute, but one which is more amenable to refactoring.

3.1.2 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 (3.1.3)) 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 2, 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.

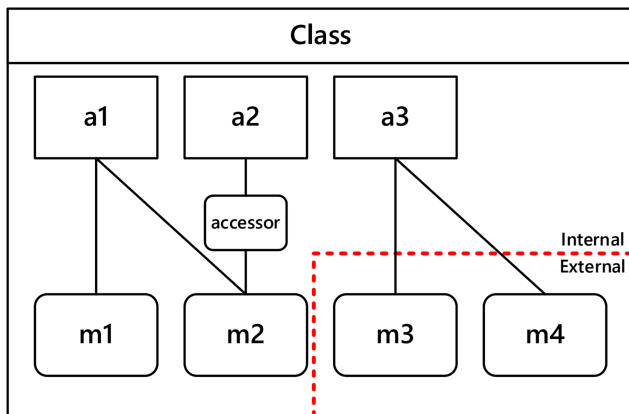

Figure 2. Conceptual diagram of taxonomy categories within a class.

3.1.3 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 [4]. 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) [3] 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) [5] 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.

3.2 Grime Categories

Before defining each grime category, we need to formally define a few concepts. Let P be a specialization of RBML [7] that describes a design pattern. The set of classes that describes P is denoted by $C(P)$. For some class $c \in C(P)$, the set of methods defined by c are denoted by $M(c)$ and the set of attributes by $A(c)$. A relationship, r, exists between an attribute $a \in A(c)$ and a method $m \in M(c)$, if m uses a via direct/indirect access (denoted as $r \in Direct$ or $r \notin Direct$). The set of method calls to m_i from methods within the same class as m_i is denoted by $calls(m_i)$. Finally, a method m_i or method pair (m_i,m_j) is *internal* iff for some $c \in C(P) \wedge \left(m_i \in M(c) \vee (m_i,m_j) \in M(c)\right)$ or is *external* iff for some $c \in C(P) \wedge \left(m_i \notin M(c) \vee (m_i \notin M(c) \vee m_j \notin M(c))\right)$ and $i \neq j$.

Direct Internal Pair Grime (DIPG). The set of invalid direct internal class relationships between pairs of methods and attributes within the classes of a pattern. DIPG can be observed when $(m_i,m_j) \in Internal$, $(r_i,r_j) \in Direct$, $r_i.attribute = r_j.attribute$, and TCC decreases.

Direct Internal Single Grime (DISG). The set of invalid direct internal class relationships between single methods and attributes within the classes of a pattern. DISG can be observed when $m_i \in Internal$, $r_i \in Direct$, and RCI decreases.

Direct External Pair Grime (DEPG). The set of invalid direct external class relationships between pairs of methods and attributes within the classes of a pattern. DEPG can be observed when $(m_i,m_j) \in External$, $(r_i,r_j) \in Direct$, $(calls(m_i) = \emptyset \wedge m_i \in External) \vee \left(calls(m_j) = \emptyset \wedge m_j \in External\right)$, and TCC decreases.

Direct External Single Grime (DESG). The set of invalid direct external class relationships between single methods and attributes within the classes of a pattern. DESG can be observed when $m_i \in External$, $r_i \in Direct$, $calls(m_i) = \emptyset$, and RCI decreases.

Indirect Internal Pair Grime (IIPG). The set of invalid indirect internal class relationships between pairs of methods and attributes within the classes of the pattern. IIPG can be observed when $(m_i,m_j) \in Internal$, $(r_i,r_j) \notin Direct$, and TCC decreases.

Indirect Internal Single Grime (IISG). The set of invalid indirect internal class relationships between single methods and attributes within the classes of a pattern. IISG can be observed when $m_i \in Internal$, $r_i \notin Direct$, $calls(m_i) = \emptyset$, and RCI decreases.

Indirect External Pair Grime (IEPG). The set of invalid indirect external class relationships between pairs of methods and attributes within the classes of a pattern. IEPG can be observed when $(m_i, m_j) \in External$, $(r_i, r_j) \notin Direct$, $r_i.attribute = r_j.attribute$, $(calls(m_i) = \emptyset \wedge m_i \in External) \vee (calls(m_j) = \emptyset \wedge m_j \in External)$, and TCC decreases.

Indirect External Single Grime (IESG). The set of invalid indirect external class relationships between single methods and attributes within the classes of a pattern. IESG can be observed when $m_i \in External$, $r_i \notin Direct$, $calls(m_i) = \emptyset$, and RCI decreases.

4. PILOT STUDY

The purpose of the pilot study is to validate and refine each grime category in the proposed taxonomy. A formal experiment was conducted to examine the effect of class grime on the understandability of design pattern realizations. We tested the following hypotheses to determine if class grime affects pattern realization understandability.

$H_{1,0}$: There is no change in mean pattern realization understandability due to class grime.

$H_{2,0}$: There is no difference in the change in understandability between indirect and direct class grime types.

$H_{3,0}$: There is no difference in the change in understandability between internal and external class grime types.

$H_{4,0}$: There is no difference in the change in understandability between single and pair class grime types.

4.1 Methodology

In order to answer the questions posed above, we elected to use a randomized complete block design for this experiment. The blocking factor is design pattern type, which has been set to the following 16 pattern types: Abstract Factory, (Object) Adapter-Command, Composite, Decorator, Façade, Factory Method, Flyweight, Mediator, Observer, Prototype, Proxy, Singleton, State, Strategy, Template Method, and Visitor. The response variable is the change in *understandability* as measured by the QMOOD software quality model [2]. Each treatment in this experiment is an injection of one of the eight types of design pattern grime (see Section 3.2).

4.1.1 Systems Studied

The experimental units under consideration are design pattern realizations randomly selected from each of the 16 design pattern types. Each design pattern realization has been extracted from a subset of the Percerons component database [1], which includes 6521 distinct design pattern realizations (spread across the 16 specific design patterns used) and contained in over 520 open source software systems.

4.1.2 Data Collection[1]

Data collection was conducted as follows. First we randomly selected the design pattern realizations from the design pattern types to form the blocks. We then randomly assigned one of the grime injection treatments to each of the realizations in a block. We then randomized the treatment/instance pairs. By processing the randomized list we locate each realization's source files, parse them and collect the necessary metrics. The parser generates a model representing the information stored in the software (see Figure 3). Using the appropriate grime injection method, we inject the methods and attribute or method uses into the model representing the software (via a prototype grime injection tool). Once the injection is complete we measure the metrics a second time. After all realizations have been processed, we calculate the difference and record the observations for analysis.

[1] Dataset available at: http://www.isaacgriffith.com/datasets.html.

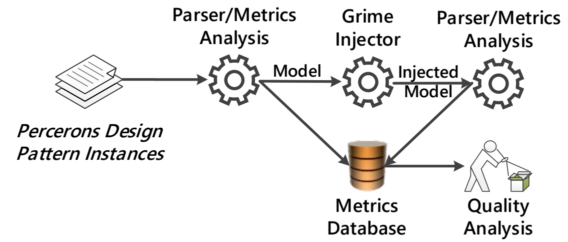

Figure 3. Injection and analysis process.

5. ANALYSIS AND DISCUSSION

The results for the simulated injection of the eight types of class grime into each of the 16 design patterns is depicted in Figure 4. There is strong evidence that not all grime effects are equal ($F_{7,127} = 54.58$, $p < 0.0001$). Further evidence is provided in Table 1, where the estimates of the mean change in understandability due to each grime type are shown. Figure 4, suggests that the grime types form two distinct groupings, based on the second classification criteria (scope). In the plot it appears that direct class grime types (DEPG, DIPG, DESG, and DISG) have a smaller negative effect on mean pattern understandability than their indirect counterparts.

We conducted a set of comparisons between each of the categorical levels of class grime. Table 2 provides the results of these contrasts. There is strong evidence for a difference between indirect and direct class grime and internal and external class grime, but there is

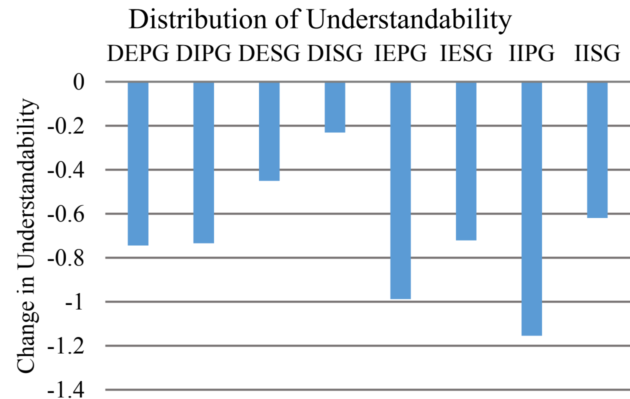

Figure 4. Plot of mean change in understandability across each type of class grime. Here, understandability is a real number ≥ 0 and the change in understandability can then be any real number.

Table 1. Mean change in understandability per class grime type.

Effect	Estimate	t value	p value
DEPG	-0.74434138	-19.06	<.0001
DESG	-0.73430188	-11.52	<.0001
DIPG	-0.45006700	-18.80	<.0001
DISG	-0.23026344	-5.90	<.0001
IEPG	-0.98814313	-25.30	<.0001
IESG	-1.15465844	-18.46	<.0001
IIPG	-0.72082738	-29.56	<.0001
IISG	-0.61929588	-15.86	<.0001

Table 2. Difference in effects of grime categories on mean change in understandability.

Effect	Estimate	t value	p value
Direct v. Indirect	1.32395	11.99	<.0001
Single v. Pair	-0.16486	-1.49	0.1386
External v. Internal	-1.60099	-14.49	<.0001

only marginal evidence for a difference between single and pair class grime.

In summary, we have found that class grime as a whole has a negative effect on design pattern realization understandability. We can also say that indirect class grime types have a greater negative effect than direct class grime types. Furthermore, we can also say that internal has less of a negative effect than external class grime types on design pattern realization understandability. These results indicate overall that the identified class grime types should be considered when modifying design pattern realizations but further evaluation as to the effects of this form of grime on other quality aspects is in order. We can infer that the injected grime caused the reduction in pattern realization understandability, and these results are generalizable to realizations of the 16 design patterns in the larger context of open source systems implemented in the Java™[2] programming language.

6. CONCLUSION AND FUTURE WORK

In this paper we have specified an extended taxonomy of class grime and conducted a pilot simulation study to evaluate the effect of class grime on pattern realization understandability, as measured using the QMOOD quality model. The findings indicate that the identified forms of class grime negatively impact understandability; which indicate that these follow the definition of grime. There is a threat to the construct validity of this study because classes in patterns may grow as a results of "other" functionality added to them. Whilst this is grime in the strict sense of the word (i.e., the definitions provided in Section 3.2), it may not be perceived as grime from a developer's point of view because it was intentional. Lastly, there is a second threat to construct validity due to the prototype nature of the injection tool. The grime injector was designed in order to inject grime compliant with the definitions presented in Section 3.2, but there is currently no separate validation step that verifies these artifacts are grime.

In future work we intend to use the taxonomy developed in this study to develop automated detection techniques for class grime in order to explore the evolution of design pattern realizations, in the context of class grime. This will then provide a method which can be expanded to include modular and organizational grime as well.

7. REFERENCES

[1] Ampatzoglou, A., Michou, O., and Stamelos, I. 2012. Building and mining a repository of design pattern instances: Practical and research benefits. *Entertainment Computing* 4, 2 (Apr. 2013), 131-142. DOI= http://dx.doi.org/10.1016/j.entcom.2012.10.002.

[2] Bansiya, J. and Davis, C.G. 2002. A hierarchical model for object-oriented design quality assessment. IEEE *Trans. Soft. Eng.* 28, 1 (Jan. 2002), 4-17. IEEE-CS, Los Alamitos, CA, USA. DOI= http://dx.doi.org/10.1109/32.979986.

[3] Bieman, J.M. and Kang, B.K. 1995. Cohesion and reuse in an object-oriented system. In *Proceedings of the ACM Symposium on Software Reusability* (Seattle, WA, USA, April 23 – 30). SSR'94. ACM, New York, NY. 259-262.

[4] Briand, L.C., Daly, J.W., and Wust, J.K. 1998. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering* 3, 1 (Mar. 1998), 65-117. DOI= http://dx.doi.org/10.1023/A:1009783721306.

[5] Briand, L., Morasca, S., and Basili, V. 1993. Measuring and assessing maintainability at the end of high level design. In *Proceedings of IEEE Conference on Software Maintenance* (Montreal, Canada, September 27 – 30). CSM'93. IEEE. 88-87.

[6] Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., and Mockus, A. 2001. Does code decay? Assessing the evidence from change management data. IEEE *Trans. Soft. Eng.* 27, 1 (Jan. 2001), 1-12. IEEE-CS, Los Alamitos, CA, USA. DOI= http://dx.doi.org/10.1109/32.895984.

[7] France, R.B., Kim D.K., Song, E., and Ghosh S. 2004. A UML-based pattern specification technique. IEEE *Trans. Soft. Eng,* 34, 5, (Mar. 2004), 193-206.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Boston, MA.

[9] Izurieta, C. 2009. *Decay and grime buildup in evolving object oriented design patterns*. Ph.D. Dissertation. Colorado State University, Fort Collins, CO, USA. Advisor James Bieman. AAI3385139.

[10] Izurieta, C. and Bieman, J.M. 2007. How software designs decay: a pilot study of pattern evolution. In *Proceedings of the First International 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.

[11] Izurieta, C. and Bieman, J.M. 2008. Testing consequences of grime buildup in object oriented design patterns. In *Proceedings of the First International Conference on Software Testing, Verification, and Validation* (Lillehammer, Norway, April 9 – 11, 2008). ICST 2008. 171-179. DOI= http://dx.doi.org/10.1109/ICST.2008.27.

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

[13] Izurieta, C., Vetro, A., Zazworka, N., Cai, Y., Seaman, C., and Shull, F. Organizing the technical debt landscape. In *Proceedings of the Third International Workshop on Managing Technical Debt* (Zurich, Switzerland, June 1, 2012). MTD 2012. IEEE, 23-26. DOI= http://dx.doi.org/10.1109/MTD.2012.6225995.

[14] Schanz, T. and Izurieta, C. 2010. Object oriented design pattern decay: a taxonomy. In *Proceedings of the 2010 ACM-IEEE International Symposium on Emprical Software Engineering and Measurement* (Bolzano-Bozen, Italy, September 16 – 17, 2010). ESEM 2010. ACM, New York, NY, 7:1-7:8. DOI= http://doi.acm.org/10.1145/1852786.1852796.

[2] http://www.oracle.com/technetwork/java/index.html