# Structural and Behavioral Taxonomies of Design Pattern Grime

Clemente Izurieta
Montana State University
Bozeman, MT 59717
clemente.izurieta@montana.edu

Derek Reimanis
Montana State University
Bozeman, MT 59717
derek.reimanis@msu.montana.edu

Isaac Griffith
Idaho State University
Pocatello, ID 83209
grifisaa@isu.edu

Travis Schanz
Fast Enterprises LLC
South Burlington, VT 05403
tschanz@gentax.com

## Abstract

Design Patterns represent the encapsulation of good design experiences and agreed upon solutions to common problems; however, as they evolve, they tend to develop *grime* –non-pattern related design components. Grime is a form of software decay that obfuscates the realization of a pattern and has decisively negative consequences on quality attributes of the pattern and consequently its embedding software. Grime comes in structural and behavioral forms. In this paper we synthesize a series of grime classifications that today form a general taxonomy. The taxonomy represents a validated and peer reviewed accumulation of knowledge that is continually evolving.

## 1 Introduction

The evolution of design patterns represents the evolution of concepts that capture domain experience. Patterns were introduced and adopted widely by the object oriented community, and this can be traced to a marquee event when the Gang of Four book [Gam95] was adopted by academics and practitioners alike. As systems evolve however; the pressures to release early, the experience and turn over of software engineers, and the complexities of designs all contribute to the decay of software systems. The measurement of such decay is complex and our contribution has focused on design patterns. We can think of design patterns as micro-architectures that are embedded in larger systems. Because their structure and behavior can be described by meta-modeling languages such as the Role Based Modeling Language (RBML) [Fra02], this allows us to compare realizations of patterns extracted from source code against their intended architecture; thus we have an ability to measure their drift as the pattern realization evolves over time.

The drift of a design pattern from its original intent can be described as *rot* or *grime* [Izu09] [Izu07]. Design pattern rot is the breakdown of a design pattern such that a critical element in the pattern ceases to exist. Thus, a realization of a pattern that experiences rot is no longer a representation of its micro-architecture. Rot is difficult to find because as the realization evolves, it becomes harder to identify. Design pattern grime is the buildup of unrelated artifacts in classes that play roles in a design pattern realization. These artifacts do not contribute to the intended role of a design pattern. Over time the pattern realization becomes hidden from practitioners.

The manuscript described herein provides a detailed description of a taxonomy for design pattern grime. Each subsection describes one aspect of the taxonomy. At the highest level we differentiate between structural and behavioral categories.



In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

## 2 Related Work

### 2.1 Software Evolution

Although a comprehensive summation of software evolution is beyond the scope of this paper, it is important to highlight key contributions that influenced the development of this taxonomy.

The earliest contributions and seminal work can be attributed to Lehman's revised laws of software evolution [Leh97]. Although controversial for their subjectivity, Lehman established a platform from which operational approaches to software evolution measurements could be derived. The common trends proposed by the laws in software growth required validation that have been the subject of many studies.

Studies associated with software aging that influenced this work include the early insights of Parnas [Par94]; which uses an analogy between software systems and medical systems to describe software aging. He uses medical terms, which equate refactoring to major surgery. Parnas also applies the notion of second opinions and describes the cost associated with preventative measures. Eick et al. [Eic01] use a number of generic code decay indices (CDIs) to analyze the change history of a telephone switching system to investigate decay.

Recent work in design pattern grime evolution has been performed by Feitosa et al. [Fei17]. They found that design pattern grime has a tendency to accumulate linearly, suggesting the quality of a pattern worsens as the grime of that pattern increases over subsequent releases.

It is also important to note that although evolution studies of design patterns continue to grow, little research has been performed outside the open source community. Further, the evolution of error propagation and uncertainty of measurements, although addressed by [Izu13] remains an under studied component.

Taxonomies represent a natural progression of evidence collected from multiple empirical studies associated with the evolution of design patterns, and is essential. "*A taxonomy promotes the classification of grime into ordered groups that are disjoint and complete while preserving natural relationships between categories*" [Sch10]. The classification, description and naming of various forms of grime as applicable to each individual design pattern have evolved since *circa* 2010.

### 2.2 Role Based Modeling Language

RBML is a visually oriented language defined in terms of a specialization of the UML metamodel that is used to verify and specify generic or domain specific design patterns. Kim et al. [Kim04] and France et al. [Fra02] introduced the Role-Based Metamodeling Language (RBML) for characterizing generic and domain-specific design patterns. RBML is based upon UML and uses the same syntax as UML. It consists of a number of behavioral and structural diagrams with each one describing different parts of the design pattern. A design pattern specification consists of two sub-specifications, the Structural Pattern Specification (SPS) and the Interaction Pattern Specification (IPS). 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, and details the flow of information that occurs as a design pattern is in operation, i.e., at program run-time. SPSes are analogous to UML class diagrams, whereas IPSes are analogous to UML sequence diagrams. Both diagrams exist at a meta-level that describes design specifications, which is referred to as the M2 level [Fra02].

## 3 Taxonomy

We divide the taxonomy of design pattern grime into two major categories: structural and behavioral. Structural refers to the changes observed via static analysis of source code or the designs; which are extracted into UML [UML97] class diagrams. UML class diagrams of design pattern realizations can be measured for compliance against the structural RBML meta-model that characterizes (potentially) an infinite number of UML models of design patterns. Behavioral refers to the deviations observed from a flow of information perspective that captures the operational side of a design pattern at run time. UML sequence diagrams can also be characterized by RBML, and we can measure an extracted UML sequence diagram of a design pattern realization against its RBML meta-model. Figure 1 shows the highest level of the hierarchy.

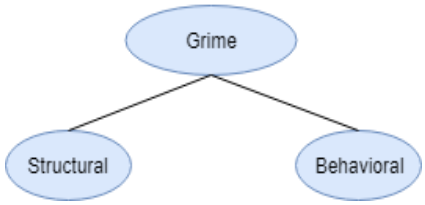


Figure 1: Design Pattern Grime Taxonomy

The following subsections describe structural and behavioral grime respectively. The description represents an abridged high level overview. Formal mathematical descriptions of each grime classification are available for modular grime [Sch10]. A full definition in currently under development.

### 3.1 Structural Grime

Structural grime is classified into three main categories: modular, class, and organizational. Modular grime is indicated by increases in the coupling of the pattern as a whole, by tracking the number of relationships (generalizations, realizations, associations, dependencies) pattern classes have with external classes. Class grime is associated with the classes that play a role in the design pattern and grime is indicated by increases in the number of ancestors of the class, the number of public attributes, and lack of cohesion. Organizational grime refers to the distribution and organization of the files and namespaces that make up a pattern. Figure 2 shows the first level of structural grime.

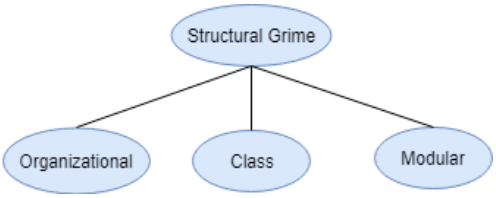


Figure 2: Structural Grime

#### 3.1.1 Modular Grime

Modular grime was further developed and validated by [Sch10] and *strength, scope and direction* were used to classify it at its highest level. Figure 3 shows the hierarchy where the left most column displays the dimensions and classification.

*Coupling* can be classified on an ordinal scale according to strength [Bie04]. Strength is determined by the difficulty of removing the coupling relationship. Persistent and temporary coupling are the most common forms in object oriented systems. The *Strength* of the relationship can be measured by afferent (Ca) and efferent (Ce) coupling to refer to the direction of a coupling relationship [Mar94]. The afferent coupling or fan-in is the count of in-bound relationships and the efferent coupling or fan-out is the count of out-bound relationships of a set of classes. Finally, the *Scope* refers to the boundary of a coupling relationship and can be either internal or external. A class belonging to a design pattern develops a relationship with external scope if another class (not in the design pattern) is coupled with the former. A relationship has internal scope if the coupling involves two classes belonging to the same realization of a design pattern.

#### 3.1.2 Class Grime

Class grime was identified by Griffith and Izurieta [Gri14]. The class grime category was extended using the properties of class cohesion. Cohesion [Bri98] is used to describe the integrity of the construction of a class. High cohesion in a class indicates close alignment of the internal components towards a common goal. In design pattern realizations, classes should have distinct responsibilities and if implemented correctly, then the specification will have high cohesion. Thus, cohesion provides a basis to determine whether a design pattern realization has been afflicted with class grime. Figure 4 shows the hierarchy of class grime. *Strength* is indicated by the method in which attributes are locally accessed by the methods of a class. 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). Direct attribute access provides a stronger and quicker but brittle relationship between a method and an attribute. Indirect attribute access implies a more flexible and weaker relationship between the method and attribute, but one which is more amenable to refactoring because it is also considered good use of design. *Scope* can either be internal or external. Internal scope refers to attribute access by local methods. External scope refers to attribute access by at least one local method that is not defined by the pattern specification. Finally, *Direction* (or *Context*) refers to the types of relationships used by surrogate metrics to measure cohesion. The majority of cohesion metrics take one of two perspectives: single-method use or method pair use of attributes [Bri98]. Two metrics capture this dimension: *Tight Class Cohesion* (TCC) [Bie95] which measures the cohesion of a class by looking at pairs of methods with attributes in common, and the *Ratio of Cohesive Interactions* (RCI) [Bri93] metric which measures the cohesion of a class by looking at how individual methods use attributes.

#### 3.1.3 Organizational Grime

Organizational grime was developed by Griffith [Gri19]. Figure 5 depicts the classification according to this taxonomy. Organizational grime refers to the distribution and organization of the files, packages and namespaces that make up a design pattern. The development of the organizational grime hierarchy comes from the following design principles [Mar03]:

- The Acyclic Dependencies Principle (ADP): Dependencies between packages should not form cycles
- The Stable Dependencies Principle (SDP): Depend in the direction of stability
- The Stable Abstractions Principle (SAP): Abstractness should increase with stability

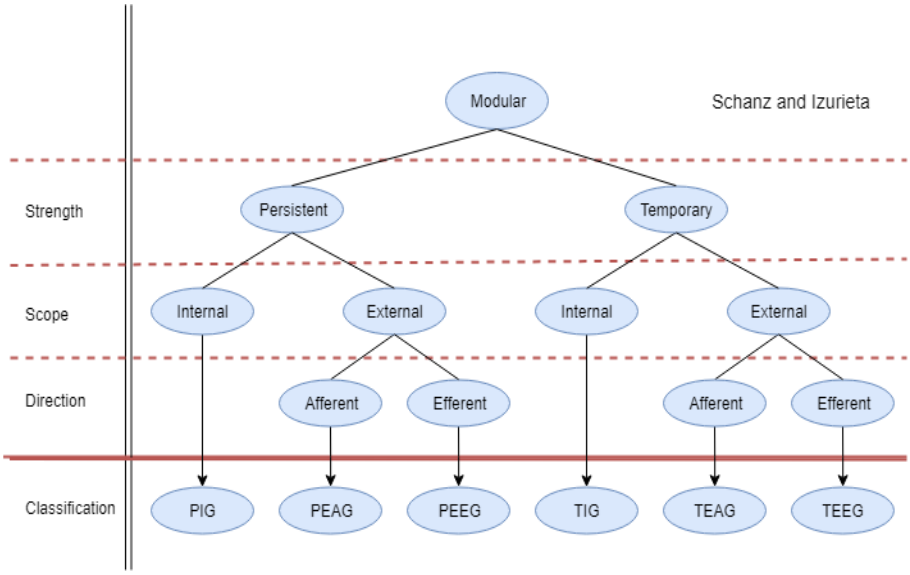


Figure 3: Modular Structural Grime

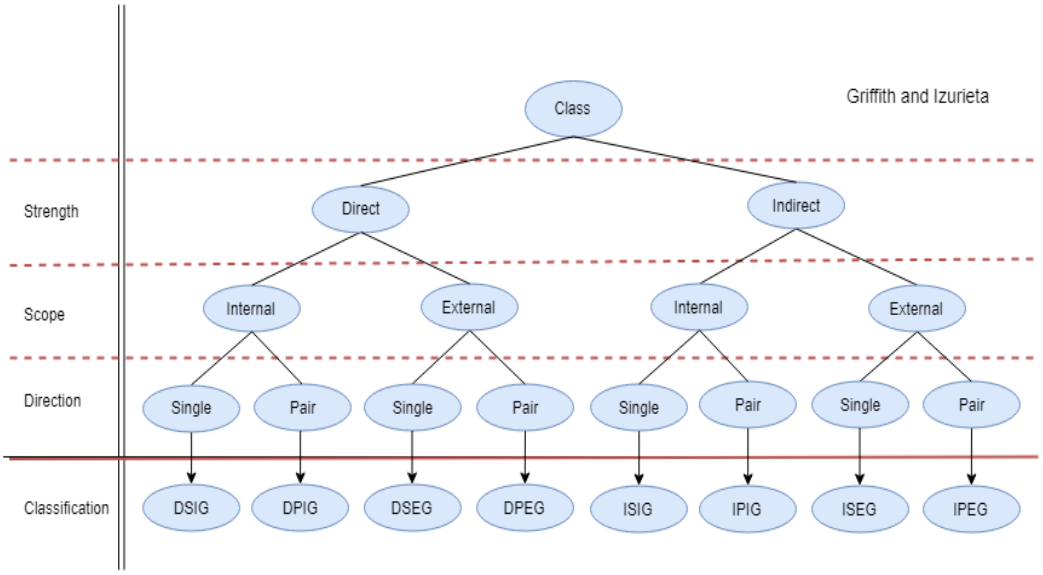


Figure 4: Class Structural Grime

- The Common Closure Principle (CCP): Classes in a package should be closed to the same kinds of changes
- The Common Reuse Principle (CRP): Classes in the same package should be reused together

These principles describe the coupling between packages and the cohesion within a package. Using the properties of package coupling and cohesion we have divided package grime into twelve specific subtypes.

Package coupling is used to develop the modular subtype of organizational grime. We consider three properties of coupling between packages. The first is the *Strength*, which can be either *Persistent* or *Temporary*. Persistent couplings are those created by inheritance, realization, and associations. Temporary couplings include use dependencies. *Scope* can be either *Internal* or *External*. Internal couplings are those that are caused by classes within the same pattern realization but spread across packages. External couplings are relationships between packages that are caused by external classes interacting with pattern classes across packages. The final property is *Direction*. This dimension refers to how the coupling affects cyclic dependencies between packages; which we label as cyclical, and the flow of stability between packages; which we label as unstable. When we consider whether the new dependency causes cycles between packages we are in the cyclical context, and when we consider the flow of dependencies towards stability, then we are in the unstable context. Together these concepts are used to form the modular branch of organizational grime.

Package cohesion is used to develop the package subtype of organizational grime. We consider only the *Scope* and *Context* dimensions. Scope can be either *Internal* or *External*, both referring to the addition of a new class or type to a package. If the new class or type is also a member of the pattern under consideration, then its scope is internal, otherwise it is external. Context takes the form of either *Closure* or *Reuse*. Closure indicates whether a new class or type fits within the package by being closed to similar changes as the other classes. Reuse indicates that we are concerned with how well a class integrates into its containing package based on how tightly it couples with the remaining classes. Together these concepts are used to form the package branch of organizational grime.

### 3.2 Behavioral Grime

Behavioral grime refers to the behavioral elements embedded in a design pattern. Behavior is encapsulated in the constructors and the operations of the design pattern. Reimanis and Izurieta [Rei15] state that "*structural grime is incapable of capturing whether or not a design pattern is behaving as intended. A pattern instance may have no structural grime, but the runtime execution of the pattern may not match the expected runtime execution of the pattern.*" Reimanis and Izurieta [Rei16] identify two specific types of errant behaviors: *Excessive Actions* and *Improper Order of Sequences*. Both of these behaviors were applied to the modular grime taxonomy [Rei19], to help generate a taxonomy of behavioral grime, which is shown in figure 6.

The dimensions of the behavioral grime taxonomy are as follows: *Strength* refers to a relationship between two UML members where *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 fan-in while *Efferent Direction* refers to a fan-out relationship. The Classification row at the bottom of the figure refers to the acronym that captures the type of behavioral grime; for example, the TIO classification is an acronym for Temporary-Internal-Order grime.

The dimensions and corresponding levels of the behavioral grime taxonomy closely mirror the modular grime taxonomy dimensions and levels because there is an inherent relationship between modular and behavioral grime. Modular grime dictates the unwanted presence of relationships between two UML members, which includes all combinations of pattern members and non-pattern members. Because of this, modular grime provides a high-level constraint on undesired pattern behaviors. However, the behavioral grime taxonomy does not mirror the modular grime taxonomy identically; specifically, the External-Efferent levels of Order Behavioral grime are missing. This is because those levels are nonsensical for Order grime. External-Efferent Order grime corresponds to a behavior from a pattern member to a non-pattern member that is out of order. Proper pattern order is dictated by pattern members calling each other in the correct order, and this definition does not include non-pattern members. Thus, the External-Efferent level of the behavioral grime taxonomy is missing.

## 4 Operationalization of Grime Evolution

In order to validate the taxonomy, we submit one operationalization approach. Tracking drift of design pattern realizations *in-situ* from their intended design requires multiple steps including detection, compliance

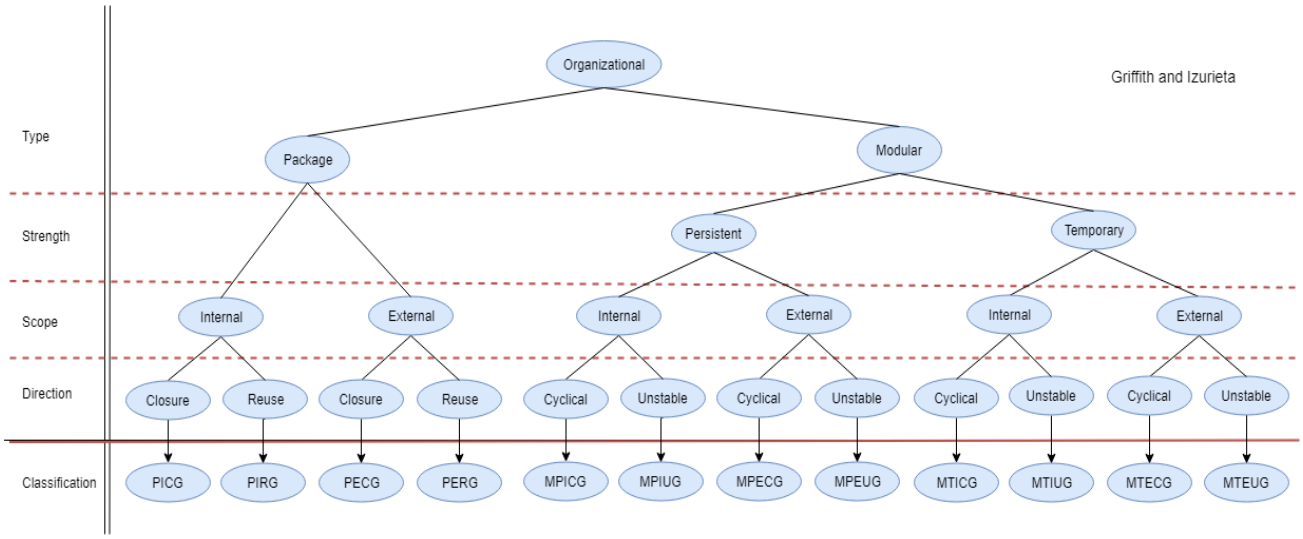


Figure 5: Organization Structural Grime

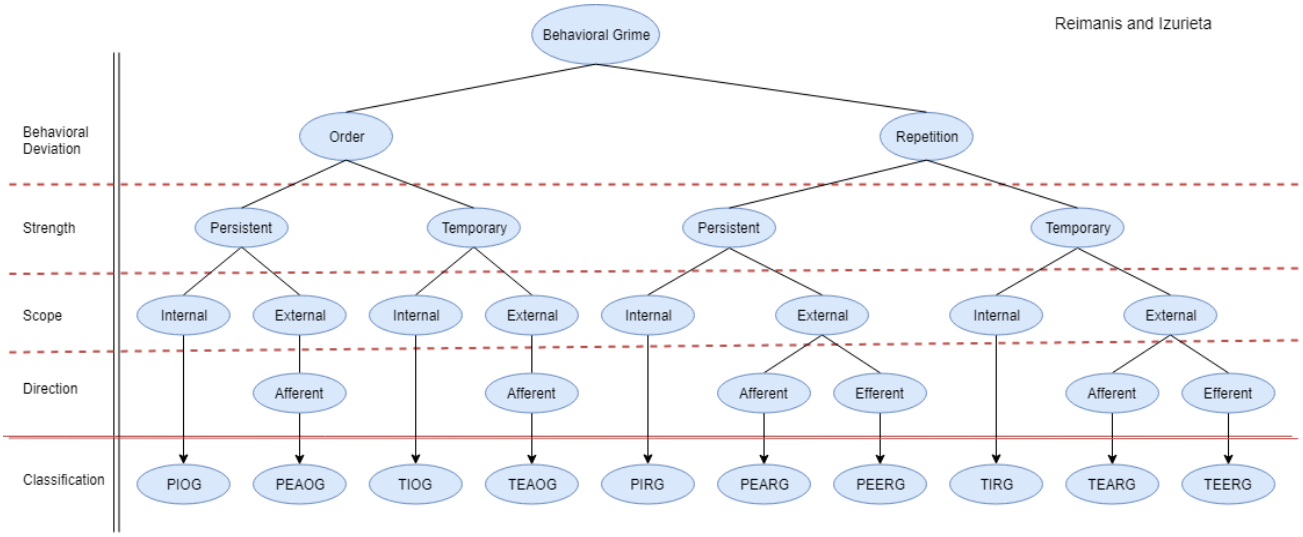


Figure 6: Behavioral grime taxonomy. Dimensions of behavioral grime are listed on the left, and corresponding characterizations are shown in the taxonomy tree

.

checking, and tracking drift over multiple releases.

Detection is a difficult problem that has been tackled by many researchers. We have leveraged many tools to help with the detection of the pattern realizations we track. They range from the simplest tools [DP1] that traverse your code and provide hints as to the location of potential realizations, to the more involved tools that build internal representations of graphs where nodes represent software classes and edges represent different types of relationships between nodes [Tsa06].

Once a design pattern realization is identified in the source code, we extract a UML class and a UML sequence diagram from the realization. These diagrams are used as representations of individual design pattern realizations that can be compared against the RBML characterization of the pattern to check for compliance. The RBML represents an abstraction of a pattern solution that can be thought of as the oracle for the pattern. The realization pattern's corresponding UML diagrams can be compared against the pattern's RBML to provide a quantifiable way of measuring drift, and thus, grime.

Implementations of design pattern realizations 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 patterns 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. Figure 7 exemplifies the process of conformance checking from design pattern realizations to their corresponding characterizations in RBML. The diagram at the top depicts the structural conformance of a simple Observer pattern realization extracted from source code against its RBML SPS, and the picture at the bottom depicts the corresponding conformance check of the behavior from a UML sequence diagram against the RBML IPS.

Conformance checking of the algorithm depicted in figure 7 has been implemented by Strasser et al. [Str11]. To compare an RBML specification and a UML diagram, the authors use a divide-and-conquer algorithm developed by Kim and Shen [Kim08], and works by breaking the RBML and UML diagrams into blocks, which are defined as any two classes or classifiers (classes and interfaces in UML) which have a relationship between them. Three kinds of relationships define three kinds of block types: association blocks, generalization blocks, and dependency blocks. The algorithm implemented by Strasser et al. only focuses on the structural components.

An algorithm for asserting behavioral conformance was designed by Kim [Kim03] and later formalized by Lu and Kim [Lu11]. This algorithm begins by establishing structural conformance via the algorithm presented by Kim and Shen [Kim08]. If structural conformance is not reached, behavioral conformance cannot be reached. After structural conformance has been established, the algorithm evaluates the presence of expected behaviors, which we refer to as stack-calls, mapping expected stack-calls to their respective member in the RBML behavioral specification. A pattern realization is said to conform to a RBML specification when all members of the RBML specification have at least one mapping from the pattern realization. Though the pattern realization may have stack-calls that do not have a mapping to the respective behavioral specification; such stack-calls are an indication of a pattern's flexibility. A pattern's characterization allows for full behavioral conformance even if other non-necessary stack-calls are included. However, these non-necessary stack-calls may constitute behavioral grime if they negatively impact the qualities of the design pattern. Therefore, after evaluating behavioral conformance, we evaluate the order and repetition of all stack-calls that are mapped to at least one RBML behavioral member. If a stack-call appears out of the expected order or is repeated unnecessarily, it is labelled as behavioral grime, the specific form of which is found based on its UML properties.

## 5 Tool Support

To exemplify the conformance checking and evolution of distinct modular grime types we describe two tools developed by the Software Engineering Laboratories (SEL) at Montana State University.

Using the algorithm described in section 4, Strasser et al. [Str11] developed a scoring function that can be visualized using a tool [1] that automatically compares the structural RBML SPS against UML diagrams extracted from source code that represent design pattern realizations. The tool reports whether or not the diagrams match, the score, if there were any errors, and displays the two diagrams. Figure 8 is a screen shot that show how a realization of a Visitor pattern (bottom of the screen) is checked for conformance against its RBML.

On the *Diagram Selection* section of the pane, the user has access to drop down menus for choosing the RBML and UML diagrams to compare. In the middle panes *RBML/UML Diagram Image*, both diagrams are displayed (the RBML diagram is displayed on top and the UML diagram is displayed immediately below). The results of running the algorithm (c.f. section 4) are displayed on the right hand side screen. Errors are labeled and displayed in the *Error Log* pane. The tool reports problems that the algorithm found, such

[1] https://code.google.com/archive/p/rbml-uml-visualizer/

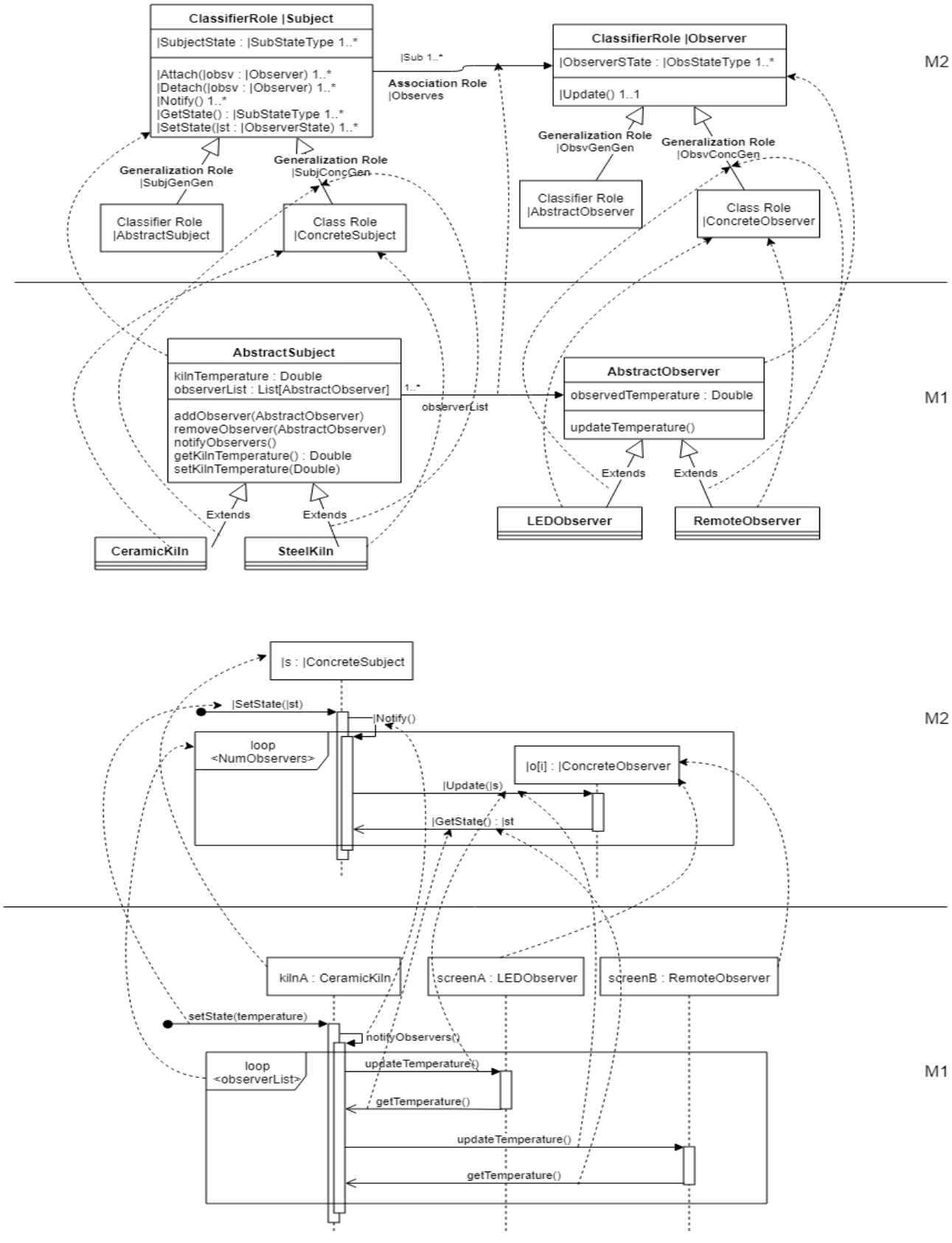


Figure 7: RBML Compliance checking of design pattern realizations

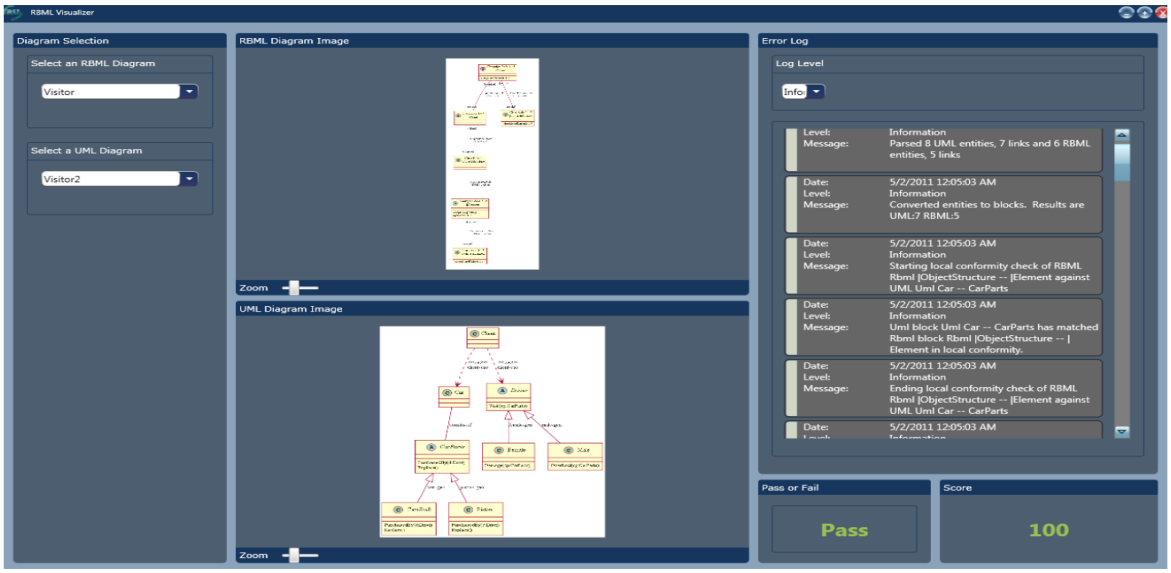


Figure 8: RBML tool to perform structural compliance checking of design pattern realizations

as mismatches between UML and RBML artifacts. Directly below in the box labeled *Pass or Fail*, the tool reports whether there was a UML diagram found that conforms to the RBML diagram. Immediately to the right is the score calculated using the scoring equation described above.

To visualize the evolution of design patterns Schanz and Izurieta [Sch10] developed a prototype to visualize grime. It allows the user control over the level of importance (i.e. a weight) for a desired pattern grime. These controls provide flexibility when exploring the evolution of a design pattern realization over its history. Figure 9 shows the evolution of pattern grime for a realization of the Singleton pattern where all forms of grime are given equal weights.

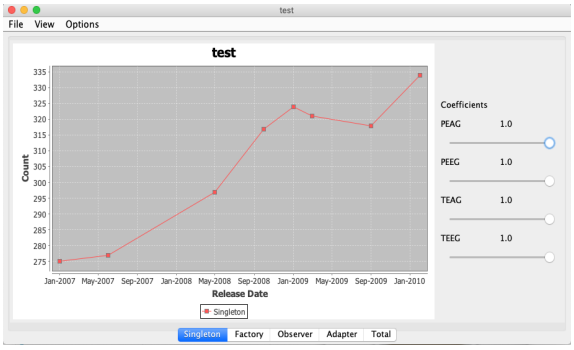


Figure 9: Singleton grime growth

Figure 10 shows the evolution of a realization of the Observer pattern where the user is mostly interested in the evolution of Permanent External Afferent Grime (PEAG), and with a 0.5 weight, also shows slight interest in Temporary External Afferent Grime (TEAG).

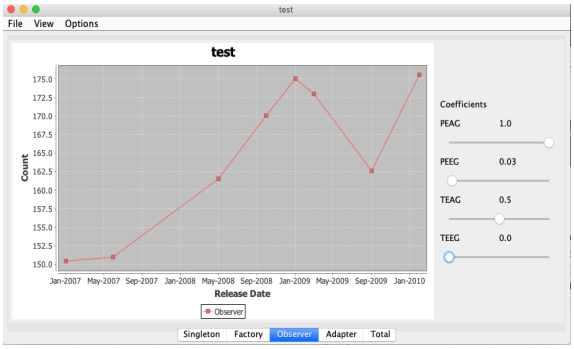


Figure 10: Observer grime growth

## 6 Motivation and Contribution Discussion

Although the usage of design patterns is promoted as being consistent with good design principles that help with maintainability and extensibility of software systems; practitioners often indicate that design patterns do not evolve as intended. Early results by the authors indicated that this was consistent with practitioner's observations, yet the decay observed in specific instances of design patterns could not be categorized due to a lack of taxonomy. To better understand if specific types of grime rot were more likely to occur than others, we set out to investigate how grime occurs and to categorize the different types. This allows researchers to compare and contrast the types of grime that are more likely to occur, and which design patterns are more succeptible to such grime.

Although the obfuscation of design pattern instances is mainly due to couplings that elements of

the pattern develop over time, we also found that the strength, direction and scope of couplings help refine the taxonomy. We found that not all couplings are equal in terms of the contribution to decay. Further, we found that cohesion of classes that are part of a design pattern played an important role in contributing to decay. Finally, the behavioral aspects of grime required that we adjust the dimensions to fit the dynamic (i.e. runtime) definitions.

Over the last ten years, our investigations have incrementally added to the formation of the taxonomy presented herein. Further, the aggregation of each aspect of the taxonomy into a cohesive hierarchy represents a contribution to the body of knowledge in this space.

To aid in comparing how different types of grime evolve over time, we developed visualization techniques and tools that allow users to focus on specific types of grime observed. Specifically, the tools allow users to parameterize weights (i.e., coefficients) associated with variables representing different types of grime from the taxonomy.

## 7 Conclusion

Research provides ample evidence that suggests that as design patterns age, the realizations of patterns remain and grime builds up. Such grime buildup can have negative and adverse consequences on many qualities of designs. Although the design pattern realizations seem to survive as systems evolve, they become obfuscated, thus making them difficult to detect, maintain and refactor –"*the original realization of design patterns remain, and the decay is measured around the grime that grows around the pattern realization over a period of time.*"

These studies helped provide the necessary motivation to investigate the specifics of grime buildup in design pattern evolution; which lent itself to the development of a taxonomy. Our motivation to explore the taxonomy across three different dimensions, namely *Strength*, *Scope*, and *Direction* is driven by the metrics that can be used as surrogates to capture and quantify change. Although other choices are possible, this approach facilitated the validation of the proposed work through empirical analysis in open systems.

Two major classifications of design pattern grime were found: *structural*, and *behavioral*. Each classification is then subdivided into further sub categories. For each category we developed synthetic examples to help calibrate a formal definition represented in the RBML meta-modeling language. The evolution of individual pattern realization's grime change can now be measured against its formal characterization and tracked for each pattern as the system evolves.

## 8 Future Work

Although significant work had led to this point, we continue to develop and refine the mathematical definitions of each form of grime (outside the scope of this manuscript). Further, taxonomy is never complete, and we suspect rare forms of *change* exist that may not be fully captured by this taxonomy yet. Continued efforts to define, calibrate, and empirically validate each form of grime are on-going and we expect to expand the research to not only include open source, but also commercial systems.

## References


[Gam95] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object Oriented Software. *Addison-Wesley, Reading MA*, 1995.

[Fra02] France,R.,Kim,D.,Song,E.,Ghosh,S. *Metarole-basedmodelinglanguage(rbml) specification v1. 0. Tech. rep., 0. Technical Report 02-106.* Computer Science Department, 2002.

[Izu09] Izurieta C. *Decay and grime buildup in evolving object oriented design patterns* Colorado State University, 2009.

[Izu07] Izurieta C., Bieman J. *How Software Designs Decay: A Pilot Study of Pattern Evolution.* First International Symposium in Empirical Software Engineering and Measurement, ESEM'07, Madrid, Spain, 2007.

[Leh97] Lehman M.M., *Laws of Software Evolution Revisited.* Proc of the 1996 European Workshop on Software Process Technology (EWSPT). France, 1996 Lecture Notes in Computer Science 1149, pp. 108-124, 1997.

[Par94] Parnas D.L., *Software Aging.* Invited Plenary Talk. 16th International Conference ICSE 1994, pp. 279-287, May 1994.

[Eic01] 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 Transactions on Software Engineering, 27(1):1-12, 2001.

[Fei17] Feitosa, D., Avgeriou, P., Ampatzoglou, A., Nakagawa, E.Y. *The evolution of design pattern grime: An industrial case study.* Intl. Conference on Product-Focused Software Process Improvement, pp. 165-181, Springer, 2017.

[Izu13] Izurieta C., Griffith I., Reimanis D., Luhr R. *On the Uncertainty of Technical Debt Measurements.* IEEE ICISA 2013 Intl. Conference on Information Science and Applications, Pattaya, Thailand, June 24-26, 2013.

[UML97] OMG *The Object Management Group OMG 2.0* http://www.omg.org

[Sch10] Schanz T., Izurieta C. *Object Oriented Design Pattern Decay: A Taxonomy.* 4th International ACM-IEEE Symposium on Empirical Software Engineering and Measurement, ESEM'10. Bolzano, Italy, 2010.

[Gri14] Griffith I., Izurieta C. *Design Pattern Decay: The Case for Class Grime.* 8th International ACM-IEEE Symposium on Empirical Software Engineering and Measurement, ESEM'14. Torino, Italy, 2014.

[Bri98] Briand, L.C., Daly, J.W., and Wust, J.K. *A unified framework for cohesion measurement in object-oriented systems.* Empirical Software Engineering 3, 1, 65-117. 1998.

[Bie04] Bieman J., Wang H. *Evaluating the Strength and Impact of Design Pattern Coupling.* Submitted manuscript IEEE Transactions on Software Engineering, 2004.

[Bri93] Briand, L., Morasca, S., and Basili, V. *Measuring and assessing maintainability at the end of high level design.* In Proceedings of IEEE Conference on Software Maintenance. Montreal, Canada, pp. 88-87, 1993.

[Mar94] Martin R. *OO Design Quality Metrics-An Analysis of Dependencies.* Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA, 1994.

[Bie95] Bieman, J.M. and Kang, B.K *Cohesion and reuse in an object-oriented system.* Proceedings of the ACM 2nd Symposium on Software Reusability, WA, USA, New York, NY. 259-262. 1995

[Gri19] Griffith I. *Design Pattern Decay A Study of Design Pattern Grime and its Impact on Quality and Technical Debt.* PhD Dissertation in progress.

[Mar03] Martin R. *Agile software development: principles, patterns, and practices.* Prentice Hall PTR, 2003.

[Rei15] Reimanis D., Izurieta C. *A Research Plan to Characterize, Evaluate and Predict the Impacts of Behavioral Decay in Design Patterns.* IEEE, ACM IDoESE, 13th International Doctoral Symposium on Empirical Software Engineering, Beijing, China 2015.

[Rei16] Reimanis D., Izurieta C. *Towards assessing the technical debt of undesired software behaviors in design patterns..* IEEE 8th International Workshop on Managing Technical Debt (MTD), pp. 24-27, Raleigh, N.C. 2016.

[Rei19] Reimanis D., Izurieta C. *Behavioral Evolution of Design Patterns: Understanding Software Reuse through the Evolution of Pattern Behavior..* 18th International Conference on Software and Systems Reuse. ICSR, Cincinnati, OH. 2019.

[DP1] Softpedia *Design Pattern Finder* https://www.softpedia.com/ Accessed: 04, 2019.

[Tsa06] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T. *Design pat- tern detection using similarity scoring* IEEE transactions on software engineering 32(11), pp. 896-909, 2006.

[Kim04] D.-K. Kim, R. France, and S. Ghosh *A uml-based language for specifying domain-specific patterns.* Journal of Visual Languages & Computing, vol. 15, no. 3-4, pp. 265-289, 2004.

[Str11] Strasser S., Frederickson C., Fenger K., Izurieta C. *An Automated Software Tool for Validating Design Patterns.* ISCA 24th International Conference on Computer Applications in Industry and Engineering, CAINE '11, Honolulu, HI, November 2011.

[Kim08] D.-K. Kim and W. Shen *Evaluating pattern conformance of uml models: a divide-and-conquer approach and case studies.* Software Quality Control, vol. 16, pp. 329-359, 2008.

[Kim03] D.-K. Kim *A meta-modeling approach to specifying patterns.* Ph.D. Disseration, 2003.

[Lu11] Lu, Lunjin and D.-K. Kim *Required behavior of sequence diagrams: Semantics and refinement* 16th IEEE International Conference on Engineering of Complex Computer Systems, pp. 127-136, 2011.