# Testing Consequences of Grime Buildup in Object Oriented Design Patterns

Clemente Izurieta, James M. Bieman
*Department of Computer Science*
*Colorado State University*
*Fort Collins, CO 80525*
*Email: {cizuriet, bieman}@colostate.edu*

## Abstract

*Evidence suggests that as software ages the original realizations of design patterns remain in place, and participants in design pattern realizations accumulate "grime" – non-pattern-related code. This research examines the consequences that grime buildup has on the testability of general purpose design patterns. Test cases put in place during the design phase and initial implementation of a project can become ineffective as the system matures. The evolution of a design due to added functionality or defect fixing increases the coupling and dependencies between many classes that must be tested. We show that as systems age, the growth of grime and the appearance of anti-patterns increase testing requirements. Early recognition and removal of grime and anti-patterns can potentially improve system testability.*

## 1. Introduction

Successful software systems continuously evolve in response to external demands for new functionality and bug fixes. One consequence of such evolution to systems built with design patterns is an increase in code within design pattern participants that does not contribute to the "mission" of individual design patterns. This added non-pattern code, or "grime", can lower the effectiveness of software test suites by increasing the number of program elements that must be tested. In addition, grime buildup can induce the formation of known testing anti-patterns.

Work on understanding design deterioration is scarce. Parnas [14] describes symptoms and causes for software decay, and uses an analogy between medical systems and software to describe the aging of a system. Parnas however does not suggest methods for tracking such decay. Eick et al. [9] use a number of generic code decay indices (CDIs) to further understand this phenomenon. They use the change history of a telecom switching system to track various CDIs. Examples of CDIs include the number of deltas, lines added or deleted as part of a change, the number of developers implementing a change, the historical number of changes in a given time interval, the frequency of changes, the span of a change in terms of the number of files that the change touches, etc.

Prior studies by Izurieta and Bieman [12] find evidence of grime buildup, non-pattern code in realizations of various general purpose design patterns. The characterization of grime buildup in design patterns is the first step towards understanding its broader impacts. Our goal here is to evaluate the impact that grime buildup has on design testability. As software evolves, the consequences of grime buildup on test requirements can impede fault detection. We study the consequences on the total number of test requirements necessary to have adequate testing of design patterns, and develop methods to compute the minimum necessary number of tests. We also investigate the formation of testing anti-patterns in realizations of design patterns, which reduces the testability of systems.

This paper is organized as follows. In section two we provide some background and specific definitions of grime buildup and review our prior work. Section three describes surrogate measures used to analyze the testability of a design pattern. Section four illustrates the testability measures on selected design pattern realizations of the open source JRefactory [13] refactoring tool. We demonstrate the growth of test requirements and formation of anti-patterns. Section five provides an analysis of the results, and discusses some threats to the validity of the case study. Section six explores related work that suggests possible improvements to the testability of design patterns.

978-0-7695-3127-4/08 $25.00 © 2008 IEEE
DOI 10.1109/ICST.2008.27

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

## 2. Decay and Grime Definitions

Software aging can affect the capability of design pattern realizations to provide pattern-specific behavior. In prior work [12] we define *decay* as the deterioration of the internal structure of system designs. *Design pattern decay* is the deterioration of the structural integrity of a design pattern realization. To experience decay, a pattern realization must undergo negative changes (deterioration) through subsequent releases and evolution. To evaluate decay we use the Meta Role Based Modeling Language (RBML) [10], which is defined in terms of a specialization of the UML metamodel. The structural integrity of a design pattern realization is determined by systematically checking its classifiers (classes, interfaces, etc.) and associations against its formal RBML specification. Informal pattern definitions, such as those described by Gamma et al. [11], are not sufficient to evaluate structural integrity of a design pattern realization.

*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. Grime is observed in the environment surrounding the realization of a pattern. Different forms of grime are identified. *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, etc. *Modular grime* is indicated by increases in the coupling of the pattern as a whole by tracking the number of relationships (generalizations, associations, dependencies, interface realizations) pattern classes have with external classes. *Organizational grime* refers to the distribution and organization of the files and namespaces that make up a pattern. Grime is relative to the role that a design pattern plays. What is considered grime from a design pattern point of view may represent adequate functionality from a different design perspective.

In a pilot case study of the JRefactory system we tracked the evolution of instances of the Visitor, State, and Singleton general purpose design patterns over a period of four years and found no evidence of structural decay. The instances of each pattern were tested for conformance with the RBML specification of the pattern, and no structural violations were found. Minimal conformance is achieved when a pattern realization meets all the constraints specified by its RBML specification.

Although no structural violations were found, we identified a form of grime buildup involving new external relationships to other artifacts of the system, which reduces modularity. Relationships between classes that are not specifically part of the RBML specification of a design pattern are considered external. These artifacts do not contribute to the intended role of a design pattern.

An increase in coupling with classes (possibly from other patterns), indicates deterioration of modularity and is a symptom of grime buildup. This form of grime also points to the deterioration of the surrounding environment of the design pattern. As dependencies increase, the system becomes harder to extend and the testability of the pattern is restricted. Additionally, a higher number of dependencies decreases the comprehensibility of the pattern. Even though the design pattern realization remains as the system evolves, it becomes obscured. For example, in our project we find instances of classes that belong to the Visitor pattern, that later implement Java interfaces that are not specified in the pattern RBML specification. While this would be allowed by the RBML specification, it breaks the modularity of the pattern.

Figure 1 displays the modular grime buildup of the Visitor and Singleton patterns in JRefactory. The figure also displays the relationship of the modular grime buildup against the total number of classes that participate in the pattern. In all cases, we see growth in the number of new external relationships compared to the number of classes participating in the pattern realization. This evidence suggests that as patterns evolve, they develop grime in the form of relationships that break down its modularity.

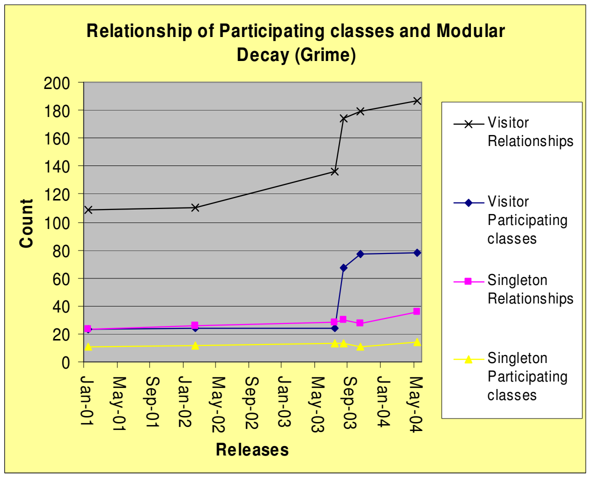

**Figure 1.** Relationship of modular grime and participating classes in the Visitor and Singleton design patterns in JRefactory [12].

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

Figure 2 shows the relationship for the Visitor and Singleton patterns separately. We see clear increases in coupling for each realization.

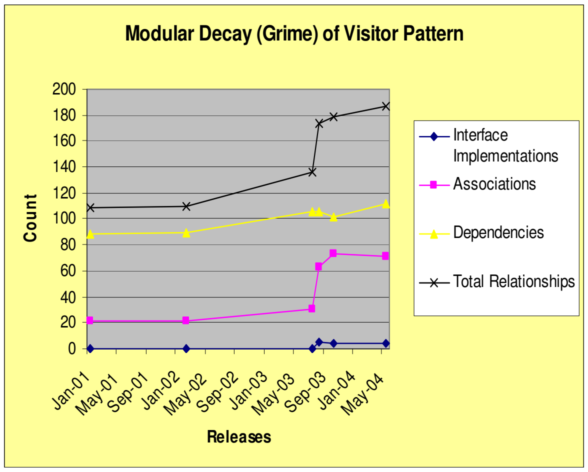

**Figure 2a.** Modular Grime buildup in a realization of the Visitor pattern in JRefactory

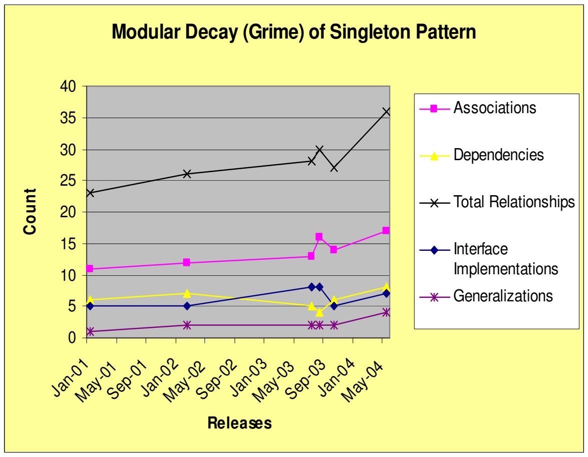

**Figure 2b.** Modular Grime buildup in a realization of the Singleton pattern in JRefactory

We observe growth in the number of new external relationships compared to the number of classes participating in the pattern realization. As patterns evolve, they develop relationships that break down their modularity. The data plotted only counts those relationships that do not play a role in the semantics of the design pattern, and are considered grime buildup from the perspective of the intended role of the design pattern.

In the case of a State pattern realization, only three realizations were found. All instances of the State pattern are not referenced, and show no signs of either structural breakdown or grime buildup. The instances of the State pattern have not evolved, thus no decay is observed. The code and UML diagrams extracted using Altova [1] show no deltas throughout the lifecycle of the system. A pattern realization that is not used can be thought of as suffering from *dormant rot*:

"Software that is not currently being used gradually becomes unusable as the remainder of the application changes. Changes in user requirements and the software environment also contribute to the deterioration." [16].

## 3. Testability of Design Patterns

We evaluate the consequences of grime buildup on test effectiveness in terms of specific test evaluation criteria. Binder [6] suggests that at its most abstract, tests should demonstrate the relationships that must hold for a system under test. To "cover" a relationship it must be executed by at least one test case. The design of object oriented systems is driven in large part by the relationships of the objects and classes that make up the system. Evaluating a full design can be daunting. However, by focusing on the design patterns that make up the system we can gain a better understanding at a localized level. If grime buildup is present, then we can begin to understand which design patterns are more prone to such deterioration. Additionally, if design patterns claim to improve overall system design, we should see little evidence of grime buildup.

Tsai et al. [15] categorizes design patterns into two distinct groups, static and dynamic. Static patterns are typically used where changes to the design are not anticipated. The Singleton pattern is an example of a static pattern. Dynamic patterns allow for extensibility either at runtime or compile time, and new functionality is achieved via polymorphic constructs. Examples of dynamic patterns include the Visitor pattern and the State pattern.

This research focuses on analyzing static and dynamic patterns and the testing consequences suffered as a result of grime buildup. The increase in number of relationships that do not play a part in the intended use of the design pattern can create structures, or "anti-patterns" that hinder testing. Anti-patterns can make testability efforts intractable and can quickly render tests ineffective. This is especially true with dynamic patterns, where inheritance hierarchies grow causing the potential number of paths that need to be tested to increase quickly.

To evaluate testability, we look for empirical evidence of the emergence of testing anti-patterns in designs. An anti-pattern *"describes a commonly occurring solution to a problem that generates*

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

*decidedly negative consequences."* [7] Anti-patterns develop as a result of increased coupling.

To track the increased coupling in design patterns we follow the evolution of various realizations of the Visitor, State, and Singleton patterns over a period of four years in the JRefactory [13] open source system.

## 3.1. Growth of Relationships and Test Requirements

We count the number of relationships (associations, interface implementations, and dependencies) that develop as a result of grime buildup, and observe the consequences on test requirements in terms of the number of test cases necessary to adequately cover a design pattern. Relationship counts are tallied per individual classes.

Class relationships are subject to many kinds of faults which must be tested. Examples of faults include incorrect multiplicities, which can generate missing or erroneous links between classes, errors in the creation or deletion of the runtime objects that must satisfy the constraints specified in the UML, etc. In the case of a binary association ($n$=1) between classes, four possible combinations must be tested [6]. For each combination an *accept* and a *reject* test case is necessary, thus yielding eight possible scenarios. In the case of n-*ary* associations, 8$n$ possible scenarios must be tested. The 8$n$ scenarios cover the basic boundary conditions, but an additional constant number of tests can be added to cover typical scenarios that are found from operational profiles. Thus, we express the minimum number of n-*ary* association test cases necessary using the linear model $A(n, k) = 8nk + c, c>=0$. The variable $k$ indicates the total number of such relationships found in the release. The consequences of not testing such combinations increase the fault proneness of the system.

Aggregation associations involve a relationship between the whole and its parts. Each element of an aggregation has an independent lifetime. Since aggregation is a kind of association, $A(n, k)$ already includes the multiplicity tests, however additional test cases are necessary to cover the test requirement for independent creation and destruction of the whole and each of its parts. We express the minimum number of test cases as $AG(n, k) = A(n, k) + 4nk$.

Composition relationships require testing of the transitive property. Composition associations involve a relationship between the whole and its parts, where the part is created and destroyed along with the whole. Thus, the lifetime of a part is dependent on the whole. Since $A(n, k)$ already covers the multiplicity tests, we express the minimum number of test cases as $C(n, k) = A(n, k) + 2nk$. The last term of this equation covers the sequential creation and destruction of the whole and its parts.

Generalization is also a transitive relationship. For any hierarchy with depth greater than or equal to three, at least two test cases are necessary. A class needs to check its relationship with its immediate parent, and by transitivity the relation must also hold with its grandparent. The minimum number of test cases necessary is thus expressed as $G(n, k) = 2nk$.

Finally, the number of dependency relationships is fully code dependent and the number of test cases necessary to cover such temporal relationships varies. Temporal relationships between any two classes exist when a method of one class defines an object of type of the other class. The lifetime of such object is bounded by the lifetime of the method that defines it. When the method finishes executing, it goes out of scope, and the object ceases to exist. Thus, we define $D(n, k) = c, c>=0$.

As systems evolve, new relationships develop between classes. These relationships may or may not have been intended in the original design. Such relationships may be the consequence of modular grime buildup. Without necessary updates to the testing suites of such systems, the possibility of faults, grows.

These test requirement computations do not take into account the complications that arise from the formation of testing anti-patterns, which in turn, further increase the count of test requirements as inheritance hierarchies develop. Additional research is required to understand how these equations are affected by the development of anti-patterns.

## 3.2. Test Anti-patterns

We look for the formation of several types of testing anti-patterns. In particular, we look for empirical evidence of two anti-patterns described by Baudry et al. [2, 5]. These anti-patterns involve inheritance hierarchies and polymorphism. Figure 3 shows the *concurrent-use-relationship* anti-pattern, where two paths exist from A to C. Class A has a transitive use path through B and B' to C. This scenario is described as an anti-pattern because A can change the state of C through one path, and read C from another path. Maintaining consistency of the state of class C can become hard to do, especially when multiple paths exist through a polymorphic hierarchy. As the number of relationships in design patterns grow, some design pattern realizations are likely to develop this form of anti-pattern.

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

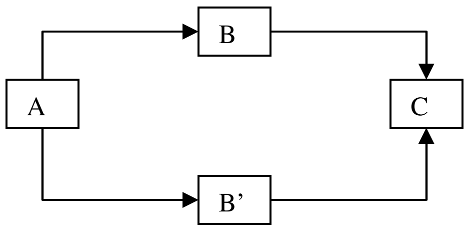

**Figure 3.** Concurrent-Use-Relationship test anti-pattern [5].

The second anti-pattern is called *self-use-relationship*, which is displayed in Figure 4. Self-usage identifies potential self referential loops in the design, which must be tested for potential infinite loops. Self references can occur at a single class level or through multiple transitive class paths.

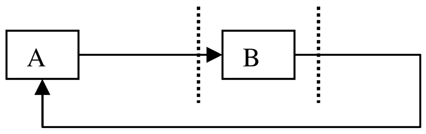

**Figure 4.** Self-Use-Relationship test anti-pattern [5].

In addition to the *concurrent-use-relationship* and *self-use-relationship* anti-patterns, we also look for anti-patterns described by Brown et al. [7] that develop as a result of grime buildup.

The *lava flow* anti-pattern is an example of dormant rot. It involves occurrences of code that remain unchanged through the lifecycle of a product. This unchanged code is detrimental to designs because, as the rest of the system evolves to conform to a new operational domain, this code lies dormant. Test cases and requirements of the dormant code may still be viable at a unit level, however when tested at a system level, this code may not work correctly because the environment and the other code in the system around it have changed. Dormant code, if not checked early, can lead to further deterioration of the system because new developers do not want to remove code that is not understood.

We also look for evidence of the *swiss army knife* anti-pattern, which occurs when classes try to implement too many methods. Symptoms include a constant increase in methods that may not have anything to do with the original intent of the class in the design pattern, or by the sudden implementation of methods via realizations of new interfaces.

## 4. Observed Effects on Test Requirements

Evidence indicates that test requirements increased and anti-patterns develop as a result of grime buildup in real systems. The process of counting relationships that form as a result of grime buildup was automated, however manual intervention was still required to distinguish between relationships that are not part of the intended role of the pattern, and those that extend the pattern in intended ways. We can compute the minimum number of test requirements necessary to provide adequate test coverage of anti-patterns. We demonstrate the consequences of grime buildup by manually identifying design anti-patterns formed.

We examine the grime buildup and its effects on testability of the JRefactory open source system. JRefactory is written in the Java language and is available through SourceForge.net. JRefactory supports many refactoring operations in a system, and automatically updates the java source files as appropriate. We studied versions 2.6.12, 2.6.38, 2.7.05, 2.8.00, 2.9.00, and 2.9.19. These releases represent the evolution of the software over a period of almost four years.

### 4.1. Observed Growth in Test Requirements

We evaluate the consequences of modular grime buildup on the adequacy of test requirements by counting the number of tests necessary to provide adequate coverage.

First we analyze the impact that associations have on test requirements. Figure 5, displays the normalized values for Visitor and Singleton design patterns in JRefactory. The equation for computing the number of tests for associations in JRefactory is given by $A(n, k) = 8nk + c, c>=0$. By normalized we mean that the x-axis values now represent equally spaced intervals for the various releases of the software. We used CurveExpert [8] statistical software to create our graphs.

Although the Singleton realization yielded slightly different results than for Visitor, both results are monotonically increasing.

The test requirements for aggregation and composition yield no additional significant insights because they are both defined in terms of associations. Specifically, the equations for computing the number of tests for aggregation and composition are given by $AG(n, k) = A(n, k) + 4nk$, and $C(n, k) = A(n, k) + 2nk$ respectively. Plotting these curves yield multiples of the association information.

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

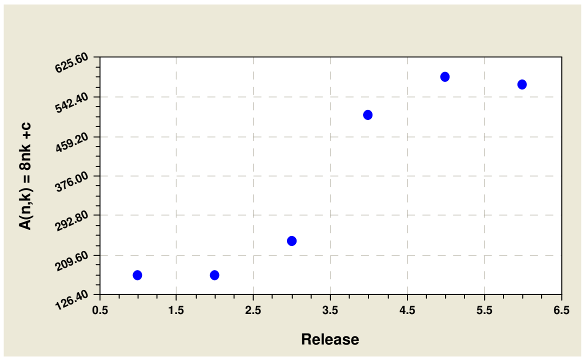

**Figure 5a.** Test requirement count for associations in the Visitor pattern of JRefactory

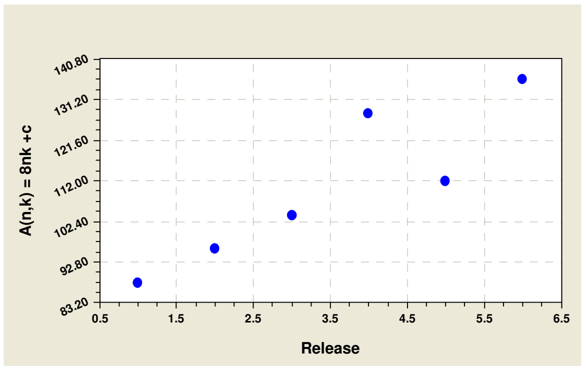

**Figure 5b.** Test requirement count for associations in the Singleton pattern of JRefactory

For dependencies, where D*(n, k)* = c, *c>=0*, we obtain the results shown in figure 6a for the Visitor pattern.

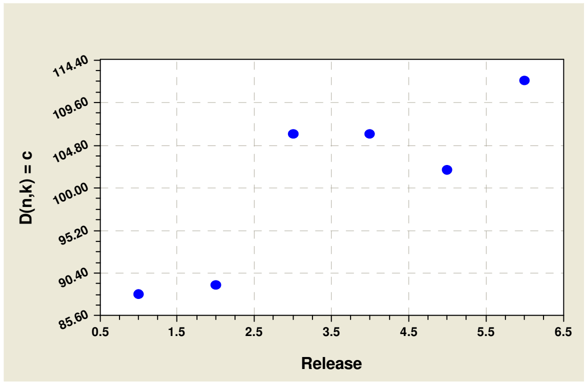

**Figure 6a.** Test requirement count for dependencies in the Visitor pattern of JRefactory

The Singleton instance yielded the values displayed in figure 6b.

Generalization consequences are defined by the equation G*(n, k)* = *2nk* + *c*. There are no generalizations in the evolution of the Visitor realization studied. However, in the case of the Singleton pattern we found data as shown in figure 7.

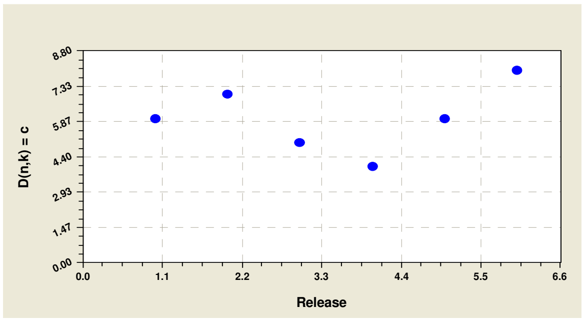

**Figure 6b.** Test requirements count for dependencies in the Singleton pattern of JRefactory

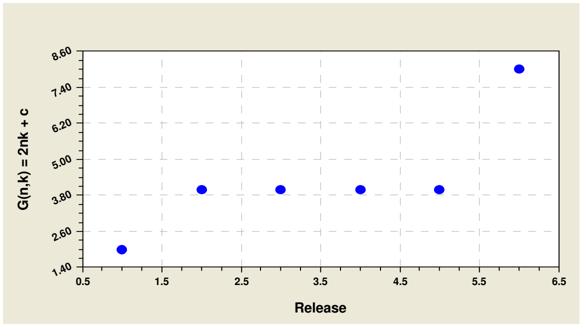

**Figure 7.** Test requirements count for generalization in the Singleton pattern of JRefactory

In general we found that realizations of the Visitor and Singleton patterns show either tapering or growth in the number of test cases necessary to test new grime buildup. Some "dips" in dependency counts were found. These are analyzed in section five.

## 4.2. Observed Appearances of Test Anti-patterns

The following examples provide evidence to support the formation of testing anti-patterns as systems evolve.

In the first example we observe the evolution of an inheritance hierarchy in a realization of the Visitor pattern. The new hierarchy formed approximately two years after the first release of JRefactory. In this example the gray arrows represent the inheritance hierarchies, the black arrows are associations, and the dashed line represents a use relationship.

Figure 8 illustrates an example of the *self-use-relationship* anti-pattern.

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

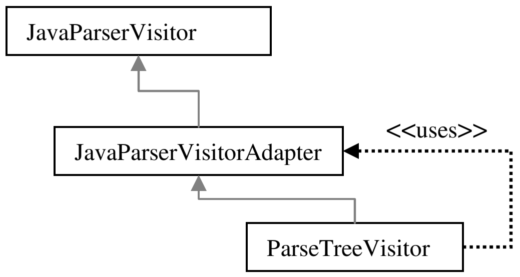

**Figure 8.** Self-Use-Relationship anti-pattern in JRefactory

The self usage reference occurs because up to eight visit methods of the ParseTreeVisitor class call *super.visitor()* before entering their own logic, which creates a circular dependency. In the worst case, each visitor will visit every concrete element in the subject hierarchy, producing a quadratic in the number of paths that must be tested. The example circular dependency traverses a use dependency and a generalization relationship.

We also find evidence of the formation of anti-patterns described by Brown et al. [7]. As described in section 2, three realizations of the State pattern were studied with no evidence of evolution found. The State pattern never evolves, but it is also never used. This is an example of dormant rot, or dead code.

In another example, we find evidence of the *swiss-army-knife* anti-pattern. The original design pattern was not intended to implement the methods defined by a new interface. Figure 9 illustrates the example found in the JRefactory system.

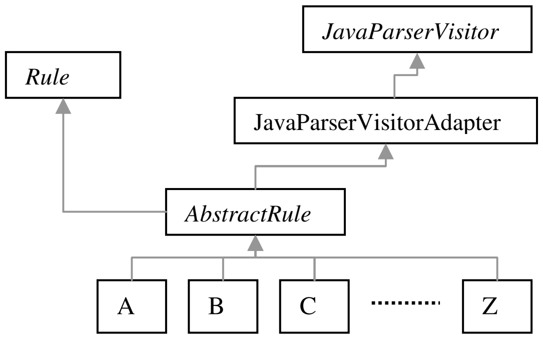

**Figure 9.** Swiss army knife anti-pattern in JRefactory

The JavaParserVisitorAdapter class did not appear until version 2.9.00, which is approximately two years after the original design. The AbstractRule class develops a realization from the Rule interface which affects the entire testing of the hierarchy that implements AbstractRule. This form of anti-pattern may be evidence of a lack of focus by the developers, and can lead to many potential testability issues.

In some cases the anti-patterns are found in the original design studied and remain for the duration of the study. Such findings are considered *foundational grime*. Design pattern decay or grime is considered *foundational* if the first identified realization of a pattern studied has already undergone some form of deterioration from prior versions of the software. If no prior versions of the software exist, then no decay or grime buildup is possible.

Figure 10 illustrates the earliest version of a *concurrent-use-relationship* anti-pattern found in all versions of a realization of the Visitor pattern in JRefactory. Clearly, the "summary" hierarchy of classes can be accessed through concurrent paths. A client of class MoveMethodRefactoring can reach various "summary" classes via two paths. The concurrent access to the "summary" hierarchy is worsened by the inheritance hierarchies involved in both paths because polymorphism must be taken into account when testing. When an instance of the class MoveMethodRefactoring uses an instance of the MoveMethodVisitor class, then it must consider objects of type ChildrenVisitor as well. Baudry et al. [4] find that *"the Visitor pattern is especially known to be difficult to test because of an extensive use of polymorphism."* They provide a testability grid for design patterns that considers the number of paths and self usages to test as a result of anti-patterns.

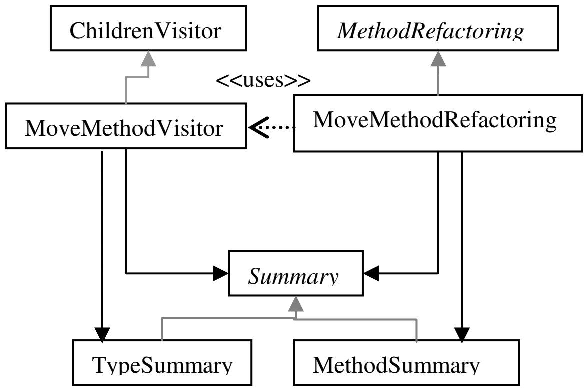

**Figure 10.** Concurrent-Use-Relationship anti-pattern in JRefactory

The formation of anti-patterns as a consequence of grime buildup is likely to be pervasive. To evaluate

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

this, other open source systems are under investigation and early evidence suggests similar results.

## 5. Analysis

The results clearly show that as the JRefactory design evolved so does the coupling of the classes involved in design patterns. At a design pattern level, we find evidence of grime buildup for various realizations of design patterns and an increase in the number of relationships that design pattern classes develop.

Design patterns develop non-pattern relationships (grime), with classes that do not play roles in a pattern, and sometimes with classes in other patterns. The extent of the non-pattern relationship buildup is measured by counting the minimum number of test cases necessary to test the relationships in a design pattern.

The computations yield the expected growth of test requirements as patterns evolve over a number of releases. Most computations show growth patterns, however some dips were observed. Such dips in the curves may be due to a refactoring process that occurred at the specified time period. After further investigation, we determined that some functionality had been moved to a different part of the system, thus lowering the relationship count. After the dip in count was experienced, the counts resume a monotonically increasing behavior, suggesting that refactoring of code is a temporary solution.

Additional research is necessary to refine the computation of the minimum number of test cases required to adequately test a pattern. The equations are one dimensional in the sense that only immediate classes suffering from grime buildup are considered. Grime buildup consequences are more far reaching than just immediate classes.

The formation of anti-patterns is a consequence of grime buildup that can greatly increase the required number of tests. We have found examples of anti-patterns in different realizations of design patterns in JRefactory, and we need to understand the effects that anti-pattern formations have on the equations.

The Visitor pattern appears to suffer from more grime and decay than the Singleton pattern, with higher test consequences. Also, the Visitor pattern is a dynamic pattern, since it can be extended via polymorphism. We found that this extensibility opens the possibility for the formation of testing anti-patterns.

Many additional realizations of patterns need to be studied to improve the validity of results. To improve content validity, we need to investigate additional variables beyond relationships between classes. Examples include class grime, which indicates grime buildup inside a class regardless of the associations that it has. Organizational grime buildup is another form of grime that can be used to investigate the physical files and directories that make up a pattern. Additional variables studied can have significant effects in the test suites developed for software, and could yield additional test requirements. A refinement of the existing equations is necessary to account for additional variables and the development of anti-patterns.

Internal validity focuses on the cause and effect relationships. In this study one can try to determine whether an increased number of test requirements is directly dependent on the grime buildup of a software pattern. The data demonstrates this relationship for JRefactory. Temporal precedence must also be determined when examining the internal validity of a system, and in the case of JRefactory we find that as grime buildup occurs, test requirements increase due to new relationships. The formation of testing anti-patterns may follow.

Finally, external validity refers to the ability to generalize results. Clearly, demonstrating the consequences in a single subject does not support general conclusions. Further studies of additional systems, additional design patterns, and additional grime buildup measures are required.

## 6. Related Work

Baudry et al. [3] propose improving the testability of designs by inserting testability constraints to design patterns when they are instantiated. They propose attaching the constraints at a UML meta-model level, so that when a design pattern is instantiated, stereotypes are added to classifiers and relationships. This allows a code developer to follow a design closer to the intended roles. In other words, if a link from class A to class B has a *"create"* stereotype, then you should only create this relationship when instantiating a class of type B, and not when reading from class B.

We propose carrying this idea further by augmenting the RBML that is used to check for conformance of a design pattern. The RBML will be used to specify the constraints. The idea is to explicitly display unacceptable constructs, such that when a realization of a design pattern is checked for conformance, no classifier or relationship in the realization should bind to a constraint in the RBML.

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.

## 7. Conclusions

It is not possible to stop the aging and deterioration of designs. Evidence 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 the testability of designs. Testability of designs is an important quality attribute. We have focused on testability of design patterns because patterns represent smaller localized parts of the design.

JRefactory, a successful real world open source system is the test subject for this study. We observed the growth of test requirements which measurably increase testing requirements. We developed and applied a method to compute the minimum test requirements necessary to test various relationships between classes. We also found evidence of *concurrent-use-relationships*, *self-use-relationships*, *swiss army knife*, and *lava flow* testing anti-patterns.

## 8. References

[1] Altova Umodel 2006. Altova. http://www.altova.com

[2] Baudry, B., Sunye, G. Improving the Testability of UML Class Diagrams. First International Workshop on Testability Assessment, 2004. IWoTA 2004. Proceedings. Nov. 2004, pp. 70- 80.

[3] Baudry, B., Traon, Y., Sunye, G. Testability Analysis of a UML Class Diagram. Software Metrics Symposium, Ottawa, Canada. June 2002, pp. 54-63.

[4] Baudry, B., Traon, Y., Sunye, G., Jezequel, J.M. Measuring and Improving Design Patterns Testability. 9th International Software Metrics Symposium. September 2003, pp. 50-59.

[5] Baudry, B., Traon, Y., Sunye, G., Jezequel, J.M., *"Towards a Safe Use of Design Patterns to Improve OO Software Testability,"* Proceedings of the 12th International Symposium on Software Reliability Engineering, ISSRE '01, pg 324.

[6] Binder R.V., *"Testing Object Oriented Systems. Models, Patterns, and Tools,"* Addison-Wesley Publishers, 2000.

[7] Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J. Anti Patterns. Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, Inc. 1998.

[8] Curve Expert 1.3 Statistical Software. A Curve Fitting System for Windows. v1.38, 2006. http://curveexpert.webshop.biz

[9] 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, 2001, 27(1):1-12.

[10] France, R., Kim, D.K., Song, E., Ghosh, S. Metarole-Based Modeling Language (RBML) Specification V1.0.

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

[12] Izurieta, C., Bieman, J.M. How Software Designs Decay: A Pilot Study of Pattern Evolution. 1st ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '07, Madrid, Spain, September 2007.

[13] JRefactory Opens Source Software. http://jrefactory.sourceforge.net

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

[15] Tsai, W.T., Tu, Y., Shao, W., Ebner, E. Testing Extensible Design Patterns in Object-Oriented Frameworks through Scenario Templates. 23rd Annual International Computer Software and Applications Conference, 1999. COMPSAC apos;99. Proceedings. Volume , Issue , 1999 Page(s):166 – 171.

[16] Wikipedia. http://en.wikipedia.org/wiki/Software_rot

Authorized licensed use limited to: MONTANA STATE UNIV BOZEMAN. Downloaded on February 10, 2010 at 09:05 from IEEE Xplore. Restrictions apply.