Preprint: Appeared in Proc. First. Int. Symp. Empirical Software Engineering and Measurement (ESEM), pp. 459-461, 2007.

How Software Designs Decay: A Pilot Study of Pattern Evolution

Clemente Izurieta
Department of Computer Science
Colorado State University
Fort Collins, Colorado, USA
01-970-481-6172
cizuriet@colostate.edu

James M. Bieman
Department of Computer Science
Colorado State University
Fort Collins, Colorado, USA
01-970-491-7096
bieman@colostate.edu

ABSTRACT
A common belief is that software designs decay as systems evolve. This research examines the extent to which software designs actually decay by studying the aging of design patterns in successful object oriented systems. Aging of design patterns is measured using various types of decay indices developed for this research. Decay indices track the internal structural changes of a design pattern realization and the code that surrounds the realization. Hypotheses for each kind of decay are tested. We found that the original design pattern functionality remains, and pattern decay is due to the "grime", non-pattern code, that grows around the pattern realization.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *software evolution, software decay, software grime.*

General Terms
Measurement, Design, Experimentation.

Keywords
Software Engineering, Evolution, Open Source Software, Software Decay, Software Grime Buildup.

1. INTRODUCTION
Software design decay is a consequence of software evolution. Decay is most apparent when the time required to make changes in a software system increases, regardless of the amount of resources available. Although studies in software aging do exist, they are scarce. Parnas [5] uses an analogy between software systems and medical systems to describe software aging. He uses the term *software geriatrics*, which equates refactoring to major surgery, applies the notion of second opinions, and describes the cost associated with preventative measures. Eick et al. [1] use a number of generic code decay indices (CDIs) to analyze the change history of a telephone switching system. Little or no work has studied how design patterns decay.

In section two we give specific definitions of decay. Section three describes our pilot case study and the different instances of patterns that we track as the system evolves. We also describe the hypotheses evaluated by the case study. Section four describes the results and section five explores threats to the validity of the case study.

2. DECAY DEFINITIONS
We focus on design patterns in object oriented systems to characterize decay. Informal pattern definitions, such as those in Gamma et al. [4], are not sufficient for identifying decay. A precise specification is necessary, but more importantly, the specification must be usable in a practical sense. We use the Meta Role Based Modeling Language (RBML) [2], which is defined in terms of a specialization of the UML metamodel.

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. 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.

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. We have identified different forms of grime. *Class grime* is associated with the classes

ESEM'07, September 20–21, 2007, Madrid, Spain.

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) 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 be adequate functionality from a different design perspective.

This paper focuses on modular and organizational grime measurements. To measure modular grime, we use the number of relationships that a realization of a pattern develops throughout its evolution. Organizational grime is measured by counting the number of files that implement the pattern, as well as the number of namespace dependencies developed by a pattern realization.

3. PILOT CASE STUDY

This case study tracks the evolution of various instances of general purpose design patterns from the JRefactory [3] Open Source System. JRefactory is a refactoring tool for the Java programming language and is available through SourceForge.net. It allows you to perform many refactorings of a system, and 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.

The following Null hypotheses are tested:

$H_{1,0}$: There are few structural violations of design pattern realizations.

$H_{2,0}$: The number of external relationships of a design pattern realization remains the same over time.

$H_{3,0}$: The namespace organization of the components that make up a design pattern remains the same over time.

4. RESULTS

We tracked the evolution of instances of the Visitor, State, and Singleton general purpose 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 an instance meets all the constraints specified by its RBML specification. Figure 1 depicts an example. Each class in the realization maps to an RBML role in the specification. Similarly, each association in the pattern

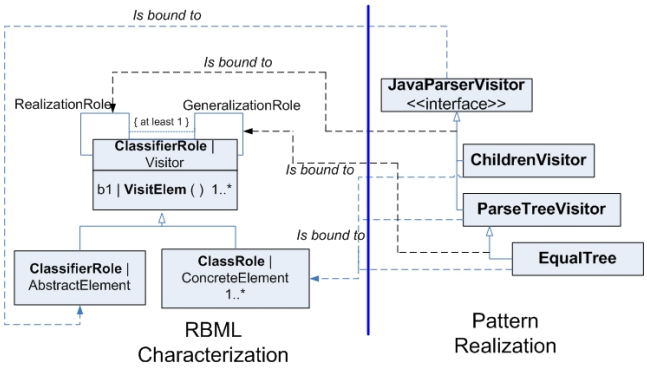

Figure 1. A simple example of structural conformance of a realization of the Visitor pattern against its RBML specification.

realization maps to an association in the RBML specification. The mapping is indicated by the "is bound to" labels in the diagram. The lack of structural violations gives us no evidence to refute $H_{1,0}$. We are currently investigating additional Open Source systems and the findings also suggest that we cannot refute the null hypothesis $H_{1,0}$. We have observed *near-instances* of patterns. A *near-instance* is a close match to a pattern RBML specification, but violates some requirements.

The modularity of a pattern realization determines its capability to localize the functionality necessary to perform its intended role. Changes performed on pattern realizations should have minimal implications on other aspects of the system. A pattern realization that lacks modularity can affect other system classes. Although structural violations are rare, we find a form of grime buildup involving new external relationships to other artifacts of the system, thus reducing modularity. Figure 2 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 we must consider the alternative hypothesis of $H_{2,0}$ for this pilot case study. As patterns evolve, they develop relationships that break down its modularity.

We examine the organization of various design patterns in the system. In one realization of the Visitor pattern, 86 files make up the package of release 2.6.12. This package is dependent on 6 external packages. By the time version 2.9.19 is released, 115 files make up the package and the number of dependencies has grown to 8 external packages.

Additionally, the package has been physically moved to a different directory.

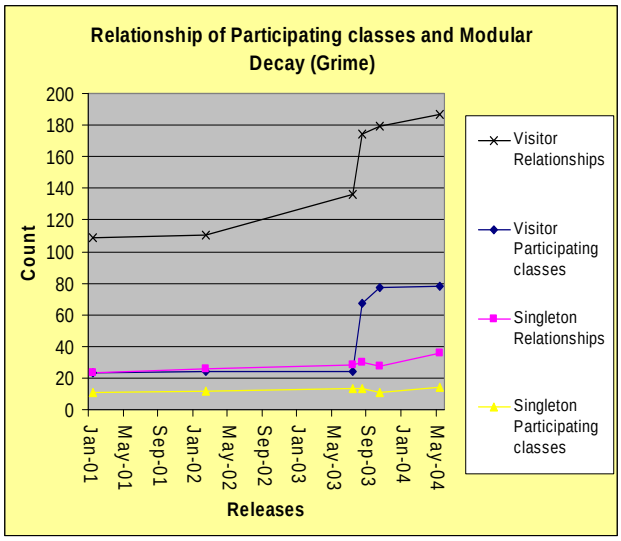

Figure 2. Relationship of Modular Grime and participating classes in the design pattern.

In version 2.6.12, the subject package is a child of the visitor package, whereas in version 2.9.19, the subject and visitor packages are siblings. Other instances show similar growth.

We found that the number of physical files that make up the implementation of design patterns remains constant throughout the evolution of the system, providing evidence that although patterns are evolving, there is no growth in the number of physical files that implement them. This in turn, contributes to organizational grime buildup. We find evidence of this in realizations of the Builder, Factory, Singleton, Adapter, Command, Iterator, State, and Visitor patterns. This evidence suggests that we must consider the alternative hypothesis for $H_{3,0}$. In Figure 3 we can see that the number of files remain constant for all but the Adapter pattern.

5. THREATS TO VALIDITY

There are threats to validity in all case studies. Construct validity refers to the meaningfulness of measurements, and to validate this you must show that the measurements are consistent with an empirical relation system. We find that some pattern realizations show more grime than others, however, this determination is dependent on the assessment of the examiner, and the strictness of the RBML specification that is used to characterize the pattern. Internal validity focuses on the cause and effect relationships. In this study one can try to determine whether decay and grime are directly related to the evolution of the software through its release history, and one can infer that temporal precedence does exist, in fact it is necessary, as pattern instances evolve. In other words,

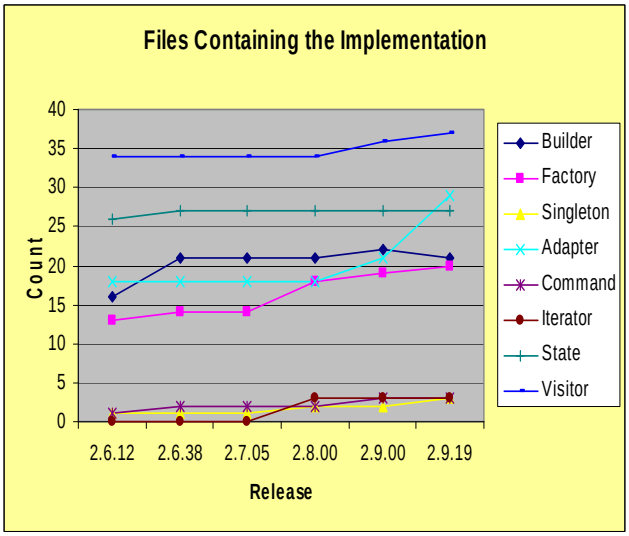

Figure 3. Organizational Grime Buildup.

less decay and grime exists at an earlier rather than later stage of the lifecycle. Finally, external validity refers to the ability to generalize results. Additional pattern instances and Open Source case studies are required beyond this pilot case study to make a further assessment.

6. CONCLUSIONS

This research is a first step towards more comprehensive studies, and aims to further the understanding of design decay. We focused on design patterns in object oriented systems, and found evidence to suggest that 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. Results from this research will provide a means for helping to identify refactoring areas in systems. Refactoring deals with the re-design or re-coding of the internals of software in order to *prevent* further decay and grime buildup.

7. REFERENCES

[1] 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.

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

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

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

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