DISSERTATION

DECAY AND GRIME BUILDUP IN EVOLVING OBJECT ORIENTED DESIGN PATTERNS

Submitted by

Clemente Izurieta

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2009

COLORADO STATE UNIVERSITY

April 29, 2009

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY CLEMENTE IZURIETA ENTITLED DECAY AND GRIME BUILDUP IN EVOLVING OBJECT ORIENTED DESIGN PATTERNS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Committee Member: Dr. Sudipto Ghosh

Committee Member: Dr. Ross McConnell

Committee Member: Dr. Alexander Hulpke

Adviser: Dr. James M. Bieman

Department Head: Dr. L. Darrell Whitley

ABSTRACT OF DISSERTATION

DECAY AND GRIME BUILDUP IN EVOLVING OBJECT ORIENTED DESIGN PATTERNS

Software designs decay as systems, uses, and operational environments evolve. As software ages the original realizations of design patterns may remain in place, while participants in design pattern realizations accumulate *grime* – non-pattern-related code. This research examines the extent to which software designs actually decay, rot and accumulate grime by studying the aging of design patterns in successful object oriented systems. By focusing on design patterns we can identify code constructs that conflict with well formed pattern structures. Design pattern rot is the deterioration of the structural integrity of a design pattern realization. Grime buildup in design patterns is a form of decay that does not break the structural integrity of a pattern but can reduce system testability and adaptability. Grime is measured using various types of indices developed and adapted for this research. Grime indices track the internal structural changes in a design pattern realization and the code that surrounds the realization. In general we find that the original pattern functionality remains, and pattern decay is primarily due to grime and not rot. We characterize the nature of grime buildup in design patterns, provide quantifiable evidence of such grime buildup, and find that grime can be classified at organizational, modular and class levels. Organizational level grime refers to namespace and physical file constitution and structure. Metrics at this level help us understand if rot and grime buildup play a role in fomenting disorganization of design patterns. Measures of modular level grime can help us to understand how the coupling of classes belonging to a design pattern develops. As dependencies between design pattern components increase without regard for pattern intent, the modularity of a pattern deteriorates. Class level grime is focused on understanding how classes that participate in design patterns are modified as systems evolve. For each level we use different measurements and surrogate indicators to help analyze the consequences that grime buildup has on testability and adaptability of 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 classes that must be tested. We show that as systems age, the growth of grime and the appearance of anti-patterns (a form of decay) increase testing requirements. Additionally, evidence suggests that, as pattern realizations evolve, the levels of efferent and afferent coupling of the classifiers that participate in patterns increase. Increases in coupling measurements suggest dependencies to and from other software artifacts thus reducing the adaptability and comprehensibility of the pattern. In general we find that grime buildup is most serious at a modular level. We find little evidence of class and organizational grime. Furthermore, we find that modular grime appears to have higher impacts on testability than adaptability of design patterns.

Identifying grime helps developers direct refactoring efforts early in the evolution of software, thus keeping costs in check by minimizing the effects of software aging. Long term goals of this research are to curtail the effects of decay by providing the understanding and means necessary to diminish grime buildup.

Clemente Izurieta
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 2009

ACKNOWLEDGEMENT

I owe an immense amount of gratitude to my advisor, Dr. James Bieman, and to Dr. Ross McConnell for his help and direction during my early years in the program. They have provided me with invaluable advice and have shown me the importance of working hard and persevering. My appreciation also goes to the members of my committee, Dr. Sudipto Ghosh and Dr. Alexander Hulpke for taking the time to review my work.

I also appreciate the support I received from my sponsors, Hewlett Packard Company and Intel Corporation, and to my manager of many years both at Hewlett Packard and Intel, Bruce Bigler. His support was invaluable.

Finally, I thank my wonderful wife for what seemed to be an endless number of nights spent on this program.

DEDICATION

This dissertation is dedicated to my grandmother Magdalena, because the first thing she said to me after I received my Masters degree was "When are you going to get your doctorate?" It never left my mind…

To Mom and Dad, it all began in December of 1980. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Successful software systems continuously evolve in response to external demands for new functionality and bug fixes. One consequence of such evolution is an increase in code that does not contribute to the mission of the original intended design. The appearance of such code was not anticipated by the original designers of the system and may introduce faults. Such faults, along with changes to the operational environment of the software contribute to the deterioration and decay of system designs. When systems decay, they can become unmanageable and eventually unusable.

Object oriented design patterns are an integral part of the design of systems today. They represent an agreed upon way of solving a problem; *"Instead of code reuse, with patterns you get experience reuse"* [34]. Design patterns have become popular for a number of reasons, including but not limited to claims of easier maintainability and flexibility of designs, reduced number of defects and faults [40], and improved architectural designs. We are interested in understanding to what extent such claims are true, and whether systems maintain early levels of quality as designs evolve. In general, it is difficult to analyze the evolution of an overall design. However, design patterns provide a frame of reference – a recognizable structure or micro-architecture. We can observe the effects of changes to design patterns over time.

Design patterns provide an initial point of reference that can be extracted from a design and whose evolution can be followed. It is a common belief that software designs decay over time and we want to determine the extent of decay as it pertains to design

patterns. We show that as a consequence, the adaptability and testability of designs deteriorate, and that as systems built with design patterns evolve, added non-pattern code, or "grime", affects pattern participants of individual design patterns.

We first characterize the nature of decay, rot, and grime in general purpose design patterns used in object oriented software systems and provide quantifiable evidence of such decay and grime. We then examine the consequences on test effectiveness and adaptability by using appropriate surrogate measures in an observational case study of three open source systems. We study different indices that track the internal structural changes of patterns as the software ages.

1.1 Approach

Initial empirical work and results [47] [48] suggest that design patterns do not structurally breakdown or rot, but as designs evolve, design pattern realizations tend to be obscured as new associations develop between classes. The number of associations that do not play a part in the intended use of the design pattern tend to increase, while the original pattern remains. We identify different levels of grime that can occur in design patterns as they evolve. Grime occurs at organizational, modular and class levels. Organizational level grime refers to namespace and physical file constitution and structure. Metrics at this level help us understand if decay and grime buildup play a role in fomenting disorganization of design patterns. Measures of modular level grime indicate the increase in coupling of evolving classes belonging to design patterns. As dependencies between design pattern components increase without regard for pattern intent, the modularity of a pattern deteriorates. Class level grime is focused on

understanding how classes that participate in design patterns are modified as systems evolve.

Design pattern realizations in selected systems from the open source community are tracked over a number of releases, to study decay and grime buildup. We test hypotheses that focus on evaluating how the indices for decay and grime buildup affect evolution of patterns in designs. We analyze the consequences to testability as a result of grime buildup and observe how changes that are not consistent with extensibility mechanisms of design patterns increase grime and thus lower the adaptability of patterns. Empirical results suggest that deterioration of design patterns occurs mostly as a result of modular grime buildup.

1.2 Contribution

There have been few prior studies on software aging. Little or no prior work focuses on understanding how a design decays, and the consequences of decay on external quality attributes. We show that the original realization of the design pattern generally remains, and the decay consists of the grime that grows inside and around the pattern realization over a period of time. The intellectual merits and potential long term benefits of this research include the identification of design decay and grime, which helps with reallocation of resources, refactoring strategies, documentation, test re-engineering, reduction of error proneness, and decreases in maintenance effort of design patterns before they become unmanageable.

2. BACKGROUND

This research progressed from an initial study of software evolution in general [58] [59] to a focus on the evolution of object oriented design patterns. Design patterns have become pervasive in software designs and the lack of studies associated with design pattern evolution coupled with claims of pattern-based improvements in software designs posed an interesting research challenge. In section 2.1 we describe major contributions to the field of software evolution. In Section 2.2 we present current research in software design decay and we describe the formation of anti-patterns that can occur as software ages. We find no evidence of design decay research performed specifically on design patterns. In order to understand how design patterns evolve and decay, a formal definition of design patterns is required. This led us to a number of pattern specification languages. Section 2.3 describes various languages for formalizing the specification of design patterns. In section 2.4 we present material on possible effects of decay on testability and adaptability of designs, and section 2.5 provides a summary.

2.1 Software Evolution

Software evolution refers to software system changes over time. Possible changes experienced by systems include changes in size, functionality, and features, along with internal structural and design changes.

The study of software evolution began in earnest in the late 1960s. Lehman [59] introduced the study of software evolution. Belady and Lehman [10] first studied the

evolution of the OS/360 system. The data produced by this early study prompted Lehman to propose the first three laws of software evolution. There are now eight laws which continue to be revised. Lehman conducted numerous studies on commercial E-type systems, where an E-type system is described by Scacchi [77] as "an application embedded in its setting of use." In other words, there is feedback between an E-type system and its operating environment causing continued evolution of the software system. Lehman's feedback ontology comes from the domain of control systems, which is traditionally tied to the physical sciences rather than software engineering. The feedback ontology's use to model software evolution does not take into account ontologies found in fields such as social networks, cultural demography, and lately communities of developers such as those found in the open source movement. The second and seventh laws are directly related to software decay. Lehman's second law of evolution—*Increasing Complexity* —states that as a system evolves, so does its complexity as measured by the size of the system. Lehman's seventh law of evolution—*Declining Quality* —describes how E-type systems, unless rigorously maintained, are perceived as having declining quality.

Turski [83] provides additional evidence of increasing complexity, finding that the development of software follows an inverse square growth function. As a system evolves, the changes adopted by successive versions cause resources to be diverted to manage the increased complexity, thus causing the overall growth rate of a system to decline.

Decay is a consequence of evolution that negatively affects the maintainability of systems. It manifests itself through the increasing complexity of a software system as

expressed by Lehman's second law. Continuing changes in the operational domain also affect the quality of the system. Lehman's seventh law of *declining quality* addresses this issue directly. The environment in which a system operates will change causing the perceived quality of an unchanged system to decline. S-type systems are also subject to implicit evolution as they relate to external environmental factors. According to Lehman [59], "an S-type system is a mathematically correct program relative to an existing and pre-stated specification." This research does not address S-type system evolution.

The laws proposed by Lehman remain controversial, mainly because it is difficult to test them due to their subjective nature. Finding surrogate measures to represent changing attributes is at best an approximation exercise that requires heuristics, thus making the laws difficult to test. Scacchi [77] states that it is unclear how to objectively measure such attributes as "user satisfaction", used in Lehman's first law. There exists no direct accounting that correlates system growth to user levels of satisfaction. System "complexity" is another example. It is unclear how to measure the vaguely defined complexity of evolving software as used in Lehman's second law of "increasing complexity." While not directly tied to declining quality, Lehman's third law of "self-regulation" states that software systems tend to dictate their own self-regulating processes, which seems to indicate a kind of internal stabilization method, but again there are no clear measures to track such self regulating processes.

Many prior studies examine software evolution with a focus on overall size measurements of the subject systems. For example, Godfrey and Tu [38] studied evolution in terms of the size of the Linux operating system. They "expected to find that Linux was growing more slowly as it got bigger and more complex." They found that

certain sub-systems are growing at a super linear rate, which contradicted results from commercial systems. Super linear growth was only found in driver sub-systems of the development releases. Another study by Izurieta and Bieman [49] found no evidence of super-linear growth, even in the driver sub-systems. The latter study was carried out on stable releases of Linux and FreeBSD.

Gustafsson et al. [43] studied the evolution of software from an architecture centric point of view. They developed *Maisa*, a system for measuring and predicting software quality during the design stages of the development process. *Maisa* computes these quality metrics from a given UML design. This methodology is an architecture-centric approach that allows developers to mitigate deviation from the original design before entering the implementation phase of the project. During the implementation phase, they used a separate system, *Columbus*, to reverse engineer code and search the software for design patterns. These patterns can then be given as inputs to *Maisa*, and compared to the original UML design.

Bieman et al. [11] studied five systems, three commercial and two open source systems, to observe changes to design patterns as systems evolve. The study found that class size is a factor in predicting the change effort in only two of the five systems. They also found that in four of the five systems pattern classes are more change prone than other classes. This result contradicts the expected evolution involving patterns where changes are made by adding new concrete classes rather than by changing existing pattern classes.

Mattsson and Bosch [65] observe the evolution of Object Oriented Frameworks (OOF). Frameworks provide a base for building applications of a similar domain. They

study three frameworks from commercial and open source domains. Their studies examine three methods that allow management to make better decisions based on the evolution of frameworks. They identify change prone modules.

Many additional studies in software evolution and feedback can be found in the book by Madhavji et al. [63]. This book is a comprehensive collection of revised academic papers that are concerned with the evolution of systems following their initial release. The book focuses on theoretical concepts and theory, as well as current practices. Various aspects of evolution are examined, including requirement changes, architectural evolution, object oriented evolution, open source systems, the laws of evolution, the gathering of feedback that drives evolution, etc. However, it contains no studies related to the deterioration and decay of software.

Prior studies of software evolution focus on size and growth, architecture, frameworks, and change proneness. Although the body of work in software evolution studies is large, very few report on design decay, a consequence of evolution. The following section describes the state of research on design decay.

2.2 Maintenance and Decay

Evolution is directly tied to changes to a system that occur over time. Belady and Lehman [10] assert that it is impossible to inject new code to older systems without introducing new defects or faults, which represents decay. Software evolution involves changes to designs, architectural specifications, and code as a result of fault repairs, new or changing requirements for additional functionality, improved performance, improved storage and memory usage, etc. These changes, which are implemented under time and resource constraints, contribute to the decay of a system. Additionally, these changes

affect the test effectiveness, adaptability, reliability, and performance of a system. Fenton and Pfleeger [30] classify changes to software into various categories, but design decay is specifically related to the adaptive and corrective categories. These types of changes are dependent on external environment changes and bug reports, hence our interest in finding characterizations of negative changes (decay) to a system.

It is safe to assume that as systems increase in size, the amount of resources spent creating new deltas to the source code increases. Some reasons for increases in maintenance [45] efforts for larger systems include: new developer staff that lack understanding of the overall code, poor architecture and designs, lack of documentation, deterioration of designs, and higher fault densities. Eick et al. [29] state that the difficulty in changing software is reflected in the time necessary to implement a change, the quality of the new code after a change, and the cost of a change as it relates to resources. In most commercial environments, time and quality of software development are controlled with tight schedules and quality regression test suites respectively. Resources are harder to control. As a system increases in size, more resources are necessary to maintain its quality. Turski [83] added that the increasing amount of resources is due to the higher psychological complexity exhibited by the software, and that an inverse square growth function can be expected over a sequence of releases. Since resources are finite, more resources are diverted to maintenance rather than developing new functionality. This research is concerned with internal negative changes to the structure of designs that affect maintainability.