Software Qual J (2013) 21:289–323
DOI 10.1007/s11219-012-9175-x

A multiple case study of design pattern decay, grime, and rot in evolving software systems

Clemente Izurieta · James M. Bieman

Published online: 7 February 2012
© Springer Science+Business Media, LLC 2012

Abstract Software designs decay as systems, uses, and operational environments evolve. Decay can involve the design patterns used to structure a system. Classes that participate in design pattern realizations accumulate grime—non-pattern-related code. Design pattern realizations can also rot, when changes break the structural or functional integrity of a design pattern. Design pattern rot can prevent a pattern realization from fulfilling its responsibilities, and thus represents a fault. Grime buildup does not break the structural integrity of a pattern but can reduce system testability and adaptability. This research examined the extent to which software designs actually decay, rot, and accumulate grime by studying the aging of design patterns in three successful object-oriented systems. We generated UML models from the three implementations and employed a multiple case study methodology to analyze the evolution of the designs. We found no evidence of design pattern rot in these systems. However, we found considerable evidence of pattern decay due to grime. Dependencies between design pattern components increased without regard for pattern intent, reducing pattern modularity, and decreasing testability and adaptability. The study of decay and grime showed that the grime that builds up around design patterns is mostly due to increases in coupling.

Keywords Design concepts · Object-oriented design methods · Design patterns · Software evolution · Decay · Grime · Technical debt · Multiple case study

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

Present Address:
C. Izurieta (✉)
Department of Computer Science, Montana State University, Bozeman, MT 59717, USA
e-mail: clemente.izurieta@cs.montana.edu

J. M. Bieman
Department of Computer Science, Colorado State University, Fort Collins, CO 80523, USA

does not contribute to the original mission of the initial intended design. The appearance of such new code was not anticipated by the original designers of the system and may introduce faults and lower system adaptability. Thus, new code along with changes to the operational environment of the software contributes to the deterioration and erosion of system designs. As a result, systems can become unmanageable and eventually unusable, forcing systems into retirement (Van Gurp and Bosch 2002).

Object-oriented design patterns are an integral part of the design of many systems today. They represent an agreed upon way of solving a problem, as concisely characterized by Freeman and Freeman: "Instead of code reuse, with patterns you get experience reuse" (Freeman and Freeman 2004). Design patterns are 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 (Guéhéneuc and Albin-Amiot 2001), 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 the structure of an overall design; thus, our intent is to focus on how well-understood patterns evolve. Design patterns provide a frame of reference—a recognizable structure or micro-architecture we can measure against. We can observe the effects of changes to design patterns over time with a focus on the consequences to the adaptability and testability of designs.

In prior work, we introduced the notion of design pattern decay and grime and performed a single-case exploratory (Yin 2003) pilot study of the effects of decay on one small part of an open-source system, JRefactory (Izurieta and Bieman 2007). We studied a small number of pattern realizations that provided an initial glimpse into the decay of design patterns. We found that coupling increased and namespace organization became more complex due to design pattern grime—non-pattern-related code, but we did not find changes that "break" the pattern (now classified as design pattern *rot*)—the changes did not prevent a pattern realization from meeting its responsibilities. We also examined the effects of design pattern grime on the testability of JRefactory (Izurieta and Bieman 2008); a handful of patterns were examined, and we observed an increase in the number of test requirements and the emergence of test anti-patterns (Baudry and Sunye 2004; Baudry et al. 2001, 2002, 2003). These initial studies provide the rationale behind the propositions presented in this paper. To understand *why* or *how* design patterns decay, we developed complementary theories that required the refinement of prior definitions of pattern decay. Thus, we have significantly enhanced the prior definitions of decay and grime, and introduced the notion of design pattern rot. We have also expanded our pattern mining to include the entire code base of JRefactory, and added two additional open-source systems—ArgoUML and eXist. The patterns studied in all three systems are "intentional" design patterns, in that the names used for classes, methods and attributes clearly demonstrate the developer's intent. We also study the consequences of decay on both testability and adaptability of design patterns.

We first characterize the nature of decay, rot, and grime in design patterns used in evolving object-oriented software systems, show how to quantify their extent and impact, and propose a model for studying decay in the context of object-oriented design patterns. We then examine the consequences on test effectiveness and adaptability in an explanatory multiple case study of three open-source systems. We found little evidence of design pattern rot. However, we found that systems generally exhibit increase in design pattern grime primarily as a result of increased coupling. The grime buildup tended to have a greater negative impact on testability than on adaptability.

2 Design decay, pattern rot, and grime

Software evolution has long been recognized as a natural process for any long-lived software system (Lehman 1980). Software changes include fault repairs, added features, performance enhancements, responses to changes in hardware and software platforms. Evolving systems tend to exhibit increases in size, functionality, and features, along with internal structural and design changes. Although such changes are necessary to keep software relevant, evolution can make a system more difficult to maintain and test.

2.1 Design decay

Belady and Lehman (1971) assert that it is impossible to inject new code into older systems without introducing new defects or faults. These changes, which are implemented under time and resource constraints, contribute to the decay of a system and can affect reliability, test effectiveness, adaptability, and performance.

Parnas (1994) uses an analogy between software systems and medicine to describe software aging. The analogy uses the term *software geriatrics*, equates refactoring to major surgery, applies the notion of second opinions, and describes the cost associated with preventative measures. His thesis is that we cannot expect to slow decay as long as developers without adequate training build complex systems. A better understanding of the nature of design decay can only help professional software engineers deal effectively with aging systems.

Eick et al. (2001) use a number of generic code decay indices (CDIs) to analyze the change history of a telephone switching system. The CDIs more aptly indicate change and do not demonstrate that change is negative. Eick's CDIs include the number of changes or 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, and the span of changes in terms of the number of files that the change touches. These indices do not meet our definition of decay because they may not represent negative effects on design integrity.

An improved understanding of design decay depends on good ways to accurately measure and characterize it in software systems. We define *decay* as the deterioration of the internal structure of system designs, where the internal structure of a design can be represented using UML class diagrams. Internal structural breakdown of a design is caused by changes that do not conform to intended architectural methodologies. Changes include violation of encapsulation, failure to follow predefined coding styles, and failure to meet agreed upon quality measures such as inheritance depth, cyclomatic complexity, and number of methods. In general, accurately measuring decay is very difficult to do, because predefined quality measures vary for individual designs. Decay is most apparent when there are increases in the time required to make changes to a software system regardless of the available resources. Thus, decay lowers the adaptability of the system leading to further atrophy and aging. Decay can also increase the number of test requirements and thus lower testability. Code decay can be thought of as a form of software (d)evolution that affects the performance, reliability, testability, and adaptability of a system and is thus directly related to software maintenance.

2.2 Design pattern rot

By focusing on micro-architectures (pattern realizations) of designs, we can compare well-formed structures against design quality violations much more accurately. Thus, we define *design pattern rot* as the breakdown of the structural integrity of a design pattern realization. To experience rot, a pattern realization must undergo negative changes (deterioration) through subsequent releases and evolution. We determine whether a change to a design pattern realization reduces the realization's structural integrity by comparing the resulting structure to that specified by the pattern's RBML specification.

RBML is the Meta Role-Based Modeling Language, which is defined in terms of a specialization of the UML metamodel (France et al. 2002, 2004; Kim 2004). Like UML, RBML is visually oriented and different aspects of the language can be used to model structural as well as behavioral aspects of patterns. Pattern specifications consist of a Structural Pattern Specification (SPS) and a set of Interaction Pattern Specifications (IPSs). The former represents the class diagram view of design patterns. An SPS specifies the static structure of patterns including participants, their properties and relationships. A set of IPSs specifies interactions inside design patterns and represents the dynamic aspects of the pattern. When mining source code for design patterns, we encounter many valid realizations of a pattern. For example, a valid realization of the Observer pattern may initially have a single observer class. After many releases, the number of observers may change significantly. Alternatively, a more drastic change may include the addition of intermediate classes between the subject and observer classes. This case still represents a valid realization of an Observer pattern. The formalization of design pattern descriptions through RBML allows us to represent multiple variants of a pattern. The RBML representations can be thought of as Pree's metapatterns (Pree 1994). For in depth descriptions of RBML, see Kim (2004) and France et al. (2002, 2004). The structural integrity of a design pattern realization is determined by systematically checking its classifier roles (classes and interfaces) and relationship roles (associations, generalizations, etc.) against its formal RBML specification. In RBML, classifier and relationship roles have multiplicities associated with them, thus restricting the number of classifiers and relationships that can appear in a pattern realization. A multiplicity of "*" (which provides a lower bound 0) allows a classifier or relationship to be optional. Further, attributes and operations in a realization of a pattern are abstracted in RBML using structural and behavioral features, respectively; each of which also has a multiplicity associated with it. The combination of multiplicities with additional Object Constraint Language (OCL) model element constraints provides a powerful method to characterize groups of pattern realizations. A single *core deviation* from the formal RBML specification stops a pattern realization from conforming to said pattern. For example, the absence of an intended (non-optional) association between two participant classes in a realization of a design pattern as a result of evolution represents a core deviation or pattern rot if the realization now fails to implement a key concept of the pattern. A developer may have unintentionally deleted the association as a result of a lack of understanding of design patterns, or because the pattern may not have been documented.

2.3 Design pattern grime

Pattern rot is not the only consequence of evolution. *Design pattern grime* is decay that does not break the structural integrity of a pattern; instead, it involves the buildup of unrelated artifacts in classes that play roles in a design pattern realization. Unrelated artifacts do not contribute to the intended responsibilities of a design pattern.

We have identified three disjoint types of grime buildup in object-oriented systems (Izurieta and Bieman 2007). Modular grime is due to added coupling of pattern classes to non-pattern classes. Class grime is due to internal additions to pattern classes that are not relevant to the pattern's responsibilities. Organizational grime is due to changes in the namespace and physical organization of a pattern realization. While true that patterns can be augmented in useful ways through the addition of non-pattern related code, developers may not be cognizant of the augmentations that a pattern has undergone. This can occur as a result of poor documentation or developer turnover. Thus, the effort required for a developer to identify a realization of a pattern in a design, or to make changes to the pattern increases because, over time, augmentations distort the realization of the pattern. We deem grime to be harmful if artifacts unrelated to the original intent of the pattern develop over time. The differences between useful and harmful grime are described under the description of each grime type.

Figure 1 depicts the relationship between the notions of decay, rot, and grime. The figure provides an abstract representation of how decay is classified. When a violation of how a given design pattern is meant to evolve is present, we classify that violation as belonging to one of the classes (sets) depicted in this figure. It is possible that other types of decay may exist, but we find no evidence in the systems we study. Each type of grime can obscure, to some degree the implementation of pattern responsibilities. The three types of grime are disjoint, and grime and rot do not intersect. A negative change to a design pattern is categorized as either rot or grime, but not both. We leave open the possibility of the existence of decay that is neither rot nor grime. When a realization of a design pattern suffers from rot, it ceases to represent said design pattern and thus cannot accumulate grime. As a consequence, design pattern rot and grime cannot occur simultaneously in a realization of a design pattern.

Sections 2.3.1, 2.3.2, 2.3.3, and 2.4 provide detailed explanations of the metrics used to measure grime as well as the consequences of grime. Table 1 provides a summary of these metrics.

2.3.1 Modular grime and its measurement

During evolution, developers add new associations with design pattern realization components. Some of these associations are unrelated to the responsibilities of the pattern. In some cases, these added associations are harmful and represent increased coupling that

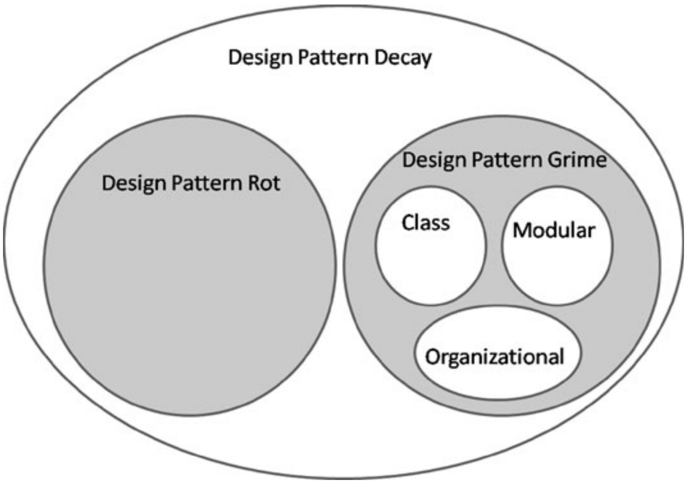


Fig. 1 The landscape of design pattern rot and grime. Pattern rot and grime are mutually exclusive

Table 1 Summary of metric types and measures used to track harmful grime buildup and its consequences

Metric type	Measures
Modular grime	Afferent (**PatCa**) and Efferent (**PatCe**) pattern coupling
Class grime	No. of methods and attributes
Organizational grime	Afferent coupling, no. of files
Testability consequences	No. of anti-patterns, increases in test requirements
Adaptability consequences	Changes in LOC, pattern instability

obscures the pattern realization. *Modular grime* is indicated by increases in the number of harmful relationships (generalizations, associations, dependencies) that pattern classes have with non-pattern classes. We describe the methodology to select and verify realizations of patterns in Sect. 3.4. Once a realization is selected, individually customized queries are implemented to gather grime statistics. For example, we do not count dependencies on the interfaces of a design pattern because they represent an agreed upon way of using the pattern; however, dependencies on concrete classes are deemed harmful. In another example, patterns are meant to be extended through generalization or specialization of classes that represent the interface. The new concrete classes increase the coupling inside a class realization, but they are not counted as grime because the coupling is not harmful.

Modular grime represents deterioration of the surrounding environment of the design pattern. As the number of relationships increase, developers are likely to find that the system has become harder to extend and the testability and adaptability of the pattern is reduced. Even though the design pattern realization remains in the design as the system evolves, it becomes obscured. The effort required for a developer to identify a realization in a design or to make changes to the pattern increases because the developer needs to understand and account for the additional couplings that distort the realization of the pattern.

We measure this breakdown in modularity by looking at the realizations of patterns over a period of time. Every relationship between classes that is not specifically part of the SPS model of the pattern indicates a type of grime buildup. Additional relationships increase coupling, thus reducing maintainability. For example, Izurieta and Bieman (2007) find instances of classes that belong to the Visitor pattern, that later develop inheritance associations from external interface classes. Although the new relationships may not be program faults, they conflict with the kinds of extensions supported by the pattern and can impact adaptability and testability.

Modular grime can be quantified by measures adapted from the afferent coupling ($C_{\mathbf{a}}$) and efferent coupling ($C_{\mathbf{e}}$) measures proposed by Martin (2002). $C_{\mathbf{a}}$ or fan-in of a package is the number of classes outside of the package that depend on classes within the package. $C_{\mathbf{e}}$ or fan-out of a package is the number of classes outside of the package that classes in the package use. The metrics fan-in ($C_{\mathbf{a}}$) and fan-out ($C_{\mathbf{e}}$) were originally defined by Henry and Kafura (1984) in terms of information flows in a given software artifact to quantitatively assess the levels of interconnectivity between components.

We use afferent pattern coupling ($PatC_{\mathbf{a}}$) and efferent pattern coupling ($PatC_{\mathbf{e}}$) as grime indicators. We compute these measures at the pattern realization level—we treat the set of classes that play a role in a design pattern realization as a special sub-package or micro-architecture. We compute afferent and efferent coupling in terms of these pattern sub-packages:

1. *Afferent pattern coupling* (*Pat*C_a) (Martin 2002) of a pattern realization is a count of a pattern's incoming dependencies, or *fan-in*. Higher PatC_a levels may lower a pattern realization's adaptability because there is a greater risk of affecting dependent modules when changes occur. Cain and McCrindle (2002) refer to afferent coupling as the "responsibility" of a class. We treat *Pat*C_a as the responsibility of a pattern realization. Afferent coupling is measured by counting a pattern's incoming edges from unidirectional associations to its participants. As modular grime buildup occurs, the value of the PatC_a of a given design pattern is likely to increase, potentially reducing its adaptability.
2. *Efferent pattern coupling* (*Pat*C_e) (Martin 2002) of a pattern realization is another indicator of modular grime. Efferent coupling, or *fan-out*, is a count of a pattern's dependencies on external classes that may or may not belong to other patterns. Cain and McCrindle (2002) refer to this measure as the "instability" of a class. We treat *Pat*C_e as the instability of a pattern realization. Outgoing dependencies can include inheritance, interface implementation, and parameter types.

2.3.2 Class grime and its measurement

When a pattern realization is initially created, its classes will contain the necessary elements, such as attributes and methods, to make it conformant to its SPS. Developers add elements as a system evolves. These new elements may not compromise a patterns structural integrity, but they generally are not relevant to the pattern's responsibilities. Such elements represent *class grime* if deemed harmful. Class grime is indicated by increases in the number of methods and public attributes in pattern classes.

Grime occurs in object-oriented systems regardless of whether classes belong to a design pattern or not. Many measures can indicate class grime. We focus on those that yield insights into what is happening to the pattern realization as a whole. As previously noted, not all class measure increases indicate grime buildup. Many classes experience change as a result of requests for new functionality or performance; however, from the perspective of the pattern, if that functionality is not related to the responsibility of the pattern, then it is considered grime. As described in Sect. 3.4, we manually select and verify each pattern realization. During verification, we develop customized queries to gather grime statistics. An informal decision is made to determine whether an existing attribute or method is useful or harmful. An initial judgment was made by the authors, and a final resolution was reached after validating the decision with an independent third-party (faculty expert). In order to count as harmful grime, agreement between all parties is required. Thus, a hybrid approach that uses manual and automated methods is used. Unlike modular grime, it is more difficult to determine harmful additions or changes to pattern classes. For example, while the appearance of public attributes is considered harmful, it is difficult to determine whether the appearance of a new method is aligned with the intent of the original pattern. Studies by Schumacher et al. (2010) reveal that a manual process to detect *God* classes produces low agreement between experts and confirms the difficulty of this task. Design patterns generally support extensions through the addition of new concrete pattern classes, rather than through modifications to existing pattern classes. Thus, pattern class measurements should remain stable. However, Bieman et al. (2003) show that classes that participate in patterns are just as change-prone as those that do not participate in patterns.

The following measures are used to track class grime:

1. The *number of public methods* is a count of the operations offered by classes that participate in design pattern definitions. Design patterns have certain operations for

various classes that must be defined in order for a realization of the pattern to be considered conformant to its RBML specification. An increase in the number of public methods is an indication that functionality is being added. Some functionality may indeed be needed and useful, but it is treated as harmful grime if it is not essential to its functionality as a pattern.

2. The *number of attributes* is a count of the data fields offered by classes that participate in design pattern definitions. Design patterns have certain attributes that must be defined in order for a realization of the pattern to be conformant to its RBML specification. For example, a Singleton pattern has a static attribute that references the instance of the class that it holds. As design pattern realizations evolve, the number of attributes can increase when functionality is added. The functionality may indeed be needed, but it is treated as grime if it is not essential to the performance of pattern responsibilities.

2.3.3 *Organizational grime and its measurement*

When patterns become part of a design they are implemented as part of a Java package or C++ namespace and are typically distributed as a number of files. The growth of a design creates additional couplings to packages that contain design patterns. Additional files are also expected to increase as new concrete classes that extend a pattern are developed. Other files may be added that are not related to pattern behavior. *Organizational grime* refers to the increase in the number of files and the coupling of the pattern files and namespaces that is not relevant to pattern responsibilities.

We track organizational grime by counting the number of packages that a design pattern belongs to, the number of distinct files that make up the design pattern, and by the couplings experienced by the packages that contain the implementation of the design patterns. Harmful grime occurs when a design pattern is spread across packages, or when the relationships between packages increase. Also, an overall increase in the size of the system (measured by the number of packages/number of patterns), while the number of pattern realizations remains constant, indicates grime buildup and decay. The following measures are used to track organizational grime:

1. *Afferent coupling* (C_a) or fan-in at the package level following Martin's original definition (Martin 2002).
2. The *number of files* containing the implementation of the various design pattern realizations is another indicator of organizational grime. A constant number of implementation files, while the pattern realizations continue to evolve and develop grime, indicate that changes and code are being added to existing files rather than being modularized, thus contributing to harmful grime buildup. Increases in file size as a result of changes made to methods that are in compliance with the pattern's RBML are not considered grime.

2.4 Potential consequences of decay

Design pattern rot represents emerging design faults, since rot indicates that a design pattern realization cannot meet all of its responsibility. The consequences of grime are less direct, as grime does not represent design faults. Rather, modular grime can increase fault proneness due to the ripple effect (Arisholm and Sjoberg 2000; Basili et al. 1996; Briand et al. 1997), and grime in general can directly lower the testability and adaptability of design patterns. While design patterns are meant to increase maintainability, decay of a design pattern in the form of grime (as found in this research) obscures the pattern's design and intended method of how the pattern is meant to be made extensible, thus reducing its maintainability.

There are at least two potential mechanisms that can impact testability (Izurieta and Bieman 2008):

1. The appearance of design anti-patterns. An anti-pattern "describes a commonly occurring solution to a problem that generates decidedly negative consequences" (Brown et al. 1998).
2. Increases in relationships (associations, realizations, and dependencies) that in turn increase test requirements.

Adaptability is affected, because grime buildup can increase the effort required to make changes. Refactoring the pattern realization may be necessary to accommodate further changes. Also, detecting the pattern realization can become more difficult because the realization is obscured.

Gilb proposed adaptability, or extendibility measures based on the total number of changes to an overall system (Gilb 1988), or the time spent on a module (Fenton and Pfleeger 1996). We are interested in the adaptability of the micro-architectures implemented as design pattern realizations. In addition, we do not have access to the process information needed to determine actual effort. Rather we look at direct effects of grime accumulation.

Code changes can be roughly indicated by changes in the number of lines of code or ΔLOC. We can observe the relationship between ΔLOC of classes that participate in design patterns and grime measures. An increase in afferent pattern coupling suggests that a pattern realization has added responsibilities as more software artifacts depend on the pattern. It becomes more difficult for a developer to make a change to the design pattern without affecting related classes. Increases in efferent pattern coupling make a pattern unstable as the pattern becomes dependent on other classes—a change in these other classes can require a modification to pattern classes. Growth observed in the ΔLOC in existing design pattern participants can increase efferent coupling. Changes in lines of code are not necessarily grimy and some patterns may exhibit changes in LOC without corresponding increases in grime measures.

We also adapt the package-level instability measure of Martin (2002) as a predictor of how hard it will be to change a software artifact. Pattern instability is computed from afferent and efferent pattern coupling measurements. The value is a ratio of efferent pattern coupling to the total number of relationships:

$$\textbf{Pattern Instability} = \mathrm{PatC}_{\mathbf{e}}/(\mathrm{PatC}_{\mathbf{a}} + \mathrm{PatC}_{\mathbf{e}})$$

Kouskouras et al. (2008) use instability, as defined by Martin. We measure pattern instability on realizations of design patterns by aggregating coupling counts of all participant classes in the design pattern.

Pattern instability is one indicator of the difficulty of modifying and adapting a given pattern realization. A higher afferent pattern coupling than efferent pattern coupling characterizes a pattern whose adaptability is limited, except through extension by adding concrete subclasses. Changes to existing pattern classes are difficult because of the large number of clients. A pattern realization with low pattern instability is a stable pattern because once established the pattern tends to remain in place and experiences little change.

High efferent pattern coupling compared to afferent coupling can point to young designs—the pattern realization has not attracted many clients, yet has many dependencies on non-pattern or server classes. As a result, the pattern realization is brittle—changes to server classes can force changes to pattern realization classes. A consequence of high pattern instability is an increase in testing requirements.

3 Case study methodology and protocol

The goal here is to observe, classify, and understand the decay of design pattern realizations in real evolving software systems. The following subsections describe the systems studied, our propositions, tools, and methods. Design documents of the systems under study are not readily available; thus, by reverse engineering the concrete code base, we can extract structural UML diagrams and mine for intended design patterns.

3.1 Systems studied

To study the consequences of grime, we examine three open-source systems. Since most of the tools available for data mining operate on Java systems, we sought systems implemented in Java. We chose two development tools and one database system. Table 2 lists all the systems and versions of software studied. We examined many open-source systems for evidence of pattern presence by using Pattern Seeker (2008), Design Pattern Finder (2008; 2009) and home grown scripts that provided "grep-like" functionality to help us identify patterns. The systems selected yielded the highest diversity of patterns and provided us with a significant number of instances of said patterns to study. Pattern Seeker is configured to search for over 100 different types of design patterns.

JRefactory (2008) 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 4 years. JRefactory has approximately 25 active contributors.

ArgoUML (2008) is an open-source UML modeling tool that includes support for all standard UML 1.4 diagrams. It runs on any Java platform and is available in ten languages with wide usage. ArgoUML features include reverse engineering with jar and class file support, OCL support, exporting of diagrams, XMI support, and the ability to run on any Java 5 or Java 6 platform. We studied versions 0.10.1, 0.12, 0.14, 0.16, 0.18.1, 0.20, 0.22, and 0.24, which represent development for approximately 5 years. ArgoUML has approximately 35 current developers and 1,400 observers.

eXist (2008) is an XML database management system and stores data according to the XML data model. We studied versions 0.8, 0.8.1, 0.8beta, 0.9, 0.9.2, and 1.0b1, which represent development for approximately 6 years. The exact number of developers participating in eXist was not readily available; however, their web site states that "the eXist-db has a large community of users and developers" (2008; 2009).

3.2 Tools

The following tools are used to mine for data and understand the structure of the systems studied:

Table 2 Systems studied

JRefactory					ArgoUML					eXist				
Version	Release date	System size (LOC)	No. classes	Pattern sizes (LOC)	Version	Release date	System size (LOC)	No. classes	Pattern size (LOC)	Version	Release date	System size (LOC)	No. classes	Pattern sizes (LOC)
2.6.12	01/2001	130,617	28	*8,411*	0.10.1	10/2002	145,485	*25*	*2,004*	0.8beta	06/2002	42,807	*13*	*479*
2.6.38	02/2002	160,778	28	*8,623*	0.12.0	08/2003	162,835	*37*	*3,230*	0.8	08/2002	48,782	*13*	*498*
2.7.05	07/2003	192,867	28	*9,568*	0.14.0	12/2003	197,376	*47*	*4,853*	0.8.1	08/2002	49,232	*13*	*498*
2.8.00	08/2003	219,846	33	*10,936*	0.16.0	07/2004	200,912	*46*	*5,418*	0.9	01/2003	61,837	*16*	*752*
2.9.00	10/2003	257,872	33	*10,205*	0.18.1	04/2005	240,439	*64*	*7,229*	0.9.2	08/2003	70,445	*13*	*674*
2.9.19	05/2004	219,608	35	*11,125*	0.20.0	02/2006	309,468	*67*	*7,746*	1.0b1	10/2006	119,037	*40*	*1,916*
					0.22.0	08/2006	288,461	*53*	*4,138*					
					0.24.0	02/2007	296,913	*51*	*4,012*					

The number of classes and pattern sizes (LOC) metrics (shown in italics) are only calculated for all participating pattern realizations

1. JDepends (2008; 2009) generates design quality metrics for the source code trees under study. JDepends calculates C_a and C_e only at the package level, so it supports the analysis of organizational grime.
2. JavaNCSS (2008; 2009) is a command line utility designed to gather code metrics for programs written in Java including cyclomatic complexity and non-commenting source statements (NCSS).
3. SemmleCode (2008; 2009) is an Eclipse plug-in that can be used to analyze the code and design of any Java Project. It is based on .QL, an object-oriented query language for structured data. Queries are constructed using the .QL language to compute specific measures, check for coding conventions, check for the appearance of harmful grime, and find bugs. .QL has a syntax that is similar to SQL. Queries are customized individually and then manually verified.
4. PatternSeeker Tool (2008) is a tool developed at Colorado State University to help with the identification of design pattern instances in code. Pattern seeker searches for specific strings to identify intentional patterns.
5. Design Pattern Finder (2008; 2009) is a GUI-based tool, similar to Pattern Seeker. It is a Windows application that searches source code directories for Gamma et al. patterns (1995).
6. AltovaUML (2006) is a commercial UML design tool that can produce UML diagrams from code. When we find possible matches for design patterns, we reverse engineer the code using AltovaUML to create a UML diagram of each candidate pattern realization. UML diagrams are manually verified.

3.3 Propositions

The following set of propositions is derived from our initial goal to understand and explain how design patterns decay. The *units of analysis* (Yin 2003) are realizations of various design patterns we find in multiple cases under study.

3.3.1 Decay, rot, and grime buildup

$\mathbf{P}_{1,0}$ Design pattern realizations do not *rot* as systems evolve

$\mathbf{P}_{2,0}$ Design pattern realizations do not exhibit increases in *grime buildup* as systems evolve

$\mathbf{P}_{3,0}$ The number of pattern realizations do not increase as the system evolves over time

3.3.2 Class grime buildup

$\mathbf{P}_{4,0}$ Increases in the number of public methods in a class that belongs to a pattern are not significant

$\mathbf{P}_{5,0}$ Increases in public attributes (fields) in a class that belongs to a pattern are not significant

3.3.3 Modular grime buildup

$\mathbf{P}_{6,0}$ Increases in afferent pattern coupling $\mathrm{PatC}_{\mathbf{a}}$ (fan-in) are not significant

$\mathbf{P}_{7,0}$ Increases in efferent pattern coupling $\mathrm{PatC}_{\mathbf{e}}$ (fan-out) are not significant

3.3.4 *Organizational grime buildup*

P$_{8,0}$ Increases in organizational grime are not significant

3.3.5 *Consequences of grime buildup*

P$_{9,0}$ There is no significant correlation between changes in lines of code (LOC) and design pattern grime buildup

P$_{10,0}$ The adaptability of design patterns, measured by pattern instability, $\mathrm{PatC}_{\mathbf{e}}/(\mathrm{PatC}_{\mathbf{a}} + \mathrm{PatC}_{\mathbf{e}})$ tends to remain unchanged as patterns evolve

P$_{11,0}$ There will be similar effects of grime buildup on the testability and adaptability of design patterns

P$_{12,0}$ Grime build-up does not affect the number of test requirements

3.4 Study process

We track the evolution of the Factory, Adapter, Singleton, State, Iterator, Proxy, and Visitor patterns in every system under investigation. Before we examined the systems, we developed specifications of each of the above pattern types using the Role-Based Meta-language (RBML) (France et al. 2002). The RBML is used to describe the essential classifiers, relationships, constraints, and multiplicities necessary for a pattern realization to be in compliance with the intended design. A pattern realization is an instance of a design pattern embedded in the code base. Many compliant realizations of a design pattern are possible. The characterization of patterns using RBML allows for the possibility of pattern realization variants to exist in the source code. An RBML expert manually verified each pattern specification to eliminate false positives.

For each realization, we perform the following steps:

1. Use *Pattern Seeker* and *Design Pattern Finder* to mine every version of software under study. These tools provide coarse statistics to help identify the total number of realizations of a given pattern. Many tools have high recall and low precision numbers (many false positives); however, these numbers are not readily available from the tool owners. We understand that finding patterns (in the worst case) is an intractable problem, so for every pattern realization identified by the tools, we used a manual process of verification to eliminate false positives. Every potential realization identified using automated means was verified using a manual process. Manual approaches work well when combined with automated tools. For example, Schull et al. (1996), demonstrate a manual approach that uses an inductive approach to identify custom patterns.
2. Use *AltovaUML* (2006) to reverse engineer the UML diagram of pattern realizations to check for conformance against the RBML specification. Checking for conformance is currently a manual process, but it can be automated (Kim and Shen 2008).
3. Mine all versions of software using available tools, and develop customized queries in the *.QL SemmleCode* language (2008; 2009) to capture all measurements for the participants of the realizations of the pattern (identified in step one) under study. Each query is designed specifically for each pattern. The customization of queries allows us to distinguish between harmful and non-harmful grime by explicitly stating which relationships to count or leave out respectively.

4. Analyze observations for the compliant pattern realizations.
5. Assess propositions.

The process of counting relationships that form as a result of grime buildup is automated (step three); however, manual intervention is still required to further help mitigate construct threats to validity.

4 Results

All versions of software were downloaded into individual directories, and separate Eclipse projects were created for each. Eclipse allows for easy organization and navigation of the code. Additionally, we took advantage of Eclipse's built in features such as name completion, automatic coloring, and plug-in support to help explore the code base of the various systems. We used queries written in the .QL language and customized scripts to generate the measurements. The ages of design pattern realizations were measured and graphed against calendar dates. Since we are studying the evolution of software patterns over time, using calendar dates is a necessary and implied time scale. Evolution studies of the Linux Operating System by Godfrey and Tu (2000) suggest using calendar dates. We also plotted results at discrete equidistant intervals representing the versions of the software; however, the lines connecting the data points do not have any significance. Lehman et al. (1998) suggest the latter approach in the study of the OS/360 system. At a high level, we gathered statistics for many design patterns; however, when observing trends at the realization level of individual patterns, we tracked the patterns described in Table 3. We found evidence of these patterns by reverse engineering the code base to extract structural UML diagrams for each pattern, as well as by using pattern finding tools such as Design Pattern Finder (2008; 2009), and Pattern Seeker (2008).

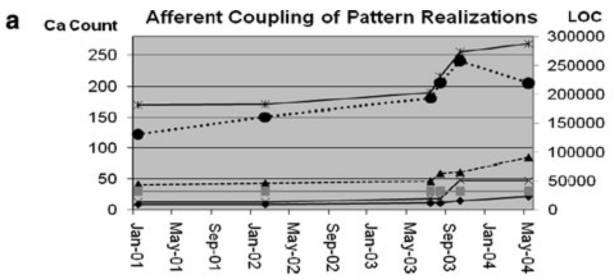
4.1 Modular grime

We found realizations of the Adapter and Factory patterns in all systems studied. Adapter pattern realizations showed evidence of afferent coupling growth in both JRefactory and eXist; however, ArgoUML's realizations experienced temporary growth between the December 2003 (release 0.14.0) and July 2004 (release 0.16.0) timeframes. The latter is attributed to a single instance that was later refactored. Not taking into account the temporary spike, we can safely conclude that ArgoUML's Adapter pattern realizations tended to remain constant. Figure 2a–c displays the afferent coupling count results for JRefactory, ArgoUML, and eXist respectively.

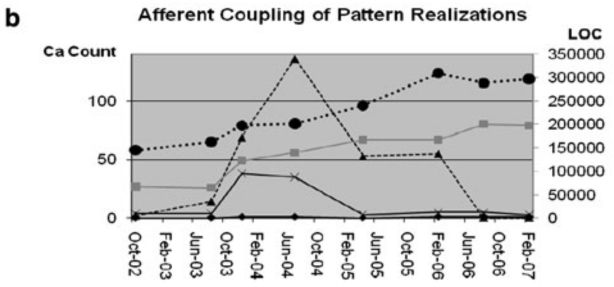
In examining ArgoUML source code, we found that the first four releases contain realizations of the Factory pattern implemented using Java classes. Beginning in April

Table 3 Realizations of studied design patterns (units of analysis) found in each system

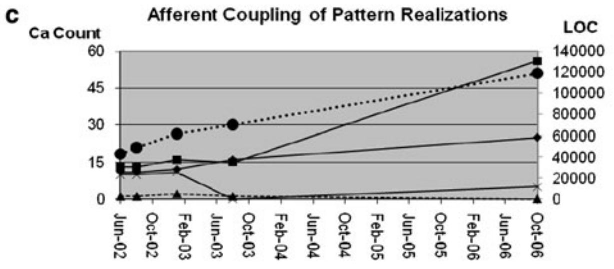
JRefactory	ArgoUML	eXist
Singleton	Singleton	Factory
State	State	Adapter
Factory	Factory	Iterator
Adapter	Adapter	Proxy
Visitor		



	Jan-01	Feb-02	Jul-03	Aug-03	Oct-03	May-04
---▲--- Factory	41	43	46	59	61	85
—×— Adapter	13	13	19	19	48	47
—◆— Singleton	9	9	11	11	15	22
—■— State	30	30	30	30	30	30
—✳— Visitor	169	170	190	215	255	268
···●·· LOC	130617	160778	192867	219846	257872	219608



	Oct-02	Aug-03	Dec-03	Jul-04	Apr-05	Feb-06	Aug-06	Feb-07
---▲--- Factory	2	14	69	136	53	55	0	0
—×— Adapter	4	4	38	35	3	5	5	3
—◆— Singleton	0	0	1	1	0	1	1	1
—■— State	27	26	49	56	67	67	80	79
···●·· LOC	145485	162835	197376	200912	240439	309468	288461	296913



	Jun-02	Aug-02	Aug-02	Jan-03	Aug-03	Oct-06
---▲--- Factory	1	1	1	2	1	0
—×— Adapter	10	10	10	11	0	5
—■— Iterator	13	13	13	16	15	56
—◆— Proxy	11	11	11	12	16	25
···●·· LOC	42807	48782	49232	61837	70445	119037

Fig. 2 **a** JRefactory afferent pattern coupling for all realizations of design patterns, **b** ArgoUML afferent pattern coupling counts for all realizations of design patterns, **c** eXist afferent pattern coupling for all realizations of design patterns

2005 (release 0.18.1), most of the Factory pattern classes were converted to Java interfaces. The patterns retained their names, but they became interfaces instead of classes. We modified our pattern recognition query to gather statistics on the pattern participants that were converted into interfaces. The drop in afferent coupling between July 2004 (release 0.16.0) and April 2005 (release 0.18.1) was due to changes in the UmlFactory class. The class originally had a $\mathrm{PatC}_{\mathbf{a}}$ count of forty-four and when separated into an interface-implementation combination, its $\mathrm{PatC}_{\mathbf{a}}$ count dropped to three, which reduced the overall afferent coupling. After February 2006, versions 0.22.0 and 0.24.0 were refactored and the implementations of the various Factory realizations were removed and replaced by a different mechanism that uses a façade approach. The interfaces for the factories remained, however, their implementations were removed. The reference counts for the Factory pattern in the ArgoUML system still show a slight increase after February 2006 in versions 0.22.0 and 0.24.0; however, these were references to the existing Factory realizations. There were no implementations of the factory classes starting with version 0.20.0; eXist' s realizations of the Factory pattern tend to remain unchanged. We did not find evidence of the Factory pattern realizations in the last version of eXist studied.

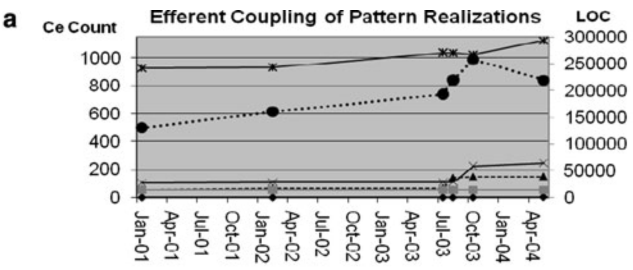
We found realizations of the Singleton pattern in JRefactory and ArgoUML. Only a single realization was available in the latter and its afferent level counts remain stable. JRefactory's afferent level counts showed constant values until August 2003 (release 2.8.00) when $\mathrm{PatC}_{\mathbf{a}}$ levels climbed sharply.

State pattern realizations were present in both JRefactory and ArgoUML. The evolution of these realizations followed distinct paths. We found three realizations of the State pattern in JRefactory. All three instances never changed (evolved) through all the releases even though the number of references to the pattern realizations does increase. This indicated that the pattern realizations continued to be used. ArgoUML's realizations of the State pattern showed a steady growth in their afferent coupling levels. A close inspection of the code did not reveal evidence of any major refactoring efforts.

Only JRefactory contained viable realizations of the Visitor pattern. Similarly, only eXist contained realizations of the Proxy and Iterator patterns. The realizations of Vistor, Proxy, and Iterator exhibit defined and steady increases in afferent coupling counts.

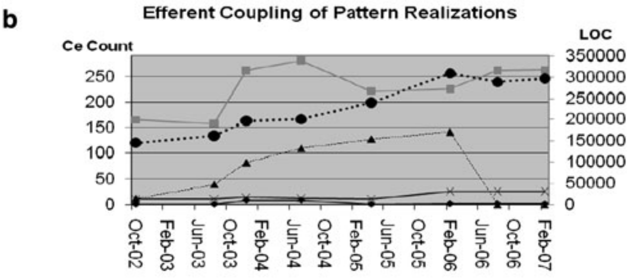
Figure 3a–c display the efferent coupling count results for JRefactory, ArgoUML, and eXist respectively. We first focus on realizations of the Adapter and Factory patterns, which occured in all of the systems studied. All realizations in all of the systems showed steady growth in efferent coupling. JRefactory showed a marked increase in efferent coupling after the July 2003 version (release 2.7.05), and eXist showed increases after the August 2003 version (release 0.9.2). In the ArgoUML system, we observed controlled growth in the cases of the Factory and the Adapter patterns. After February 2006, versions 0.22.0 and 0.24.0 of the Factory pattern were refactored and the implementations of the various Factory realizations were removed and replaced by a different mechanism that used a Façade approach. We found realizations of the Singleton pattern in JRefactory (Fig. 3a) and ArgoUML (Fig. 3b). As expected, $\mathrm{PatC}_{\mathbf{e}}$ levels remained constant for both systems. Once in place, our conjecture is that a Singleton pattern will not increase its coupling to external classes. In JRefactory, we observed a slight increase in the May 2004 release; however, this is attributed to one additional realization of the pattern introduced in version 2.9.19. As already described, the State pattern realizations remained unchanged throughout their lifecycles in the JRefactory system. Figure 3a shows the efferent coupling count. Both $\mathrm{PatC}_{\mathbf{a}}$ and $\mathrm{PatC}_{\mathbf{e}}$ remained unchanged; however, the reference counts did increase as the pattern ages. In ArgoUML, we observed moderate and subdued growth. The Visitor pattern in JRefactory (Fig. 3a) experienced continual and consistent growth

a



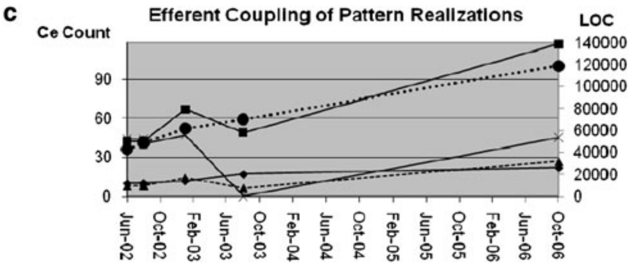
	Jan-01	Feb-02	Jul-03	Aug-03	Oct-03	May-04
---▲--- Factory	57	65	65	142	148	151
—×— Adapter	104	109	109	109	225	246
—◆— Singleton	3	3	3	3	3	6
—■— State	54	54	54	54	54	54
—✱— Visitor	927	934	1038	1037	1026	1124
···●·· LOC	130617	160778	192867	219846	257872	219608

b



	Oct-02	Aug-03	Dec-03	Jul-04	Apr-05	Feb-06	Aug-06	Feb-07
···▲··· Factory	12	39	82	110	128	142	0	0
—×— Adapter	12	12	15	13	12	26	26	26
—◆— Singleton	1	1	9	9	1	2	2	2
—■— State	165	158	262	281	221	226	262	263
···●·· LOC	145485	162835	197376	200912	240439	309468	288461	296913

c



	Jun-02	Aug-02	Aug-02	Jan-03	Aug-03	Oct-06
---▲--- Factory	8	8	8	14	6	27
—×— Adapter	44	44	40	47	0	45
—■— Iterator	42	42	42	67	49	117
—◆— Proxy	10	10	10	12	17	22
···●·· LOC	42807	48782	49232	61837	70445	119037

Fig. 3 **a** JRefactory efferent pattern coupling for all realizations of design patterns, **b** ArgoUML efferent pattern coupling for all realizations of various design patterns, **c** eXist efferent pattern coupling for all realizations design patterns

throughout its lifecycle as do realizations of the Proxy and Iterator patterns in the eXist database (Fig. 3c).

4.2 Class grime

Class grime is an internal attribute of pattern classes rather than the relationships that form part of a design pattern. Class grime measurements were taken for all classes that are intended participants of a design pattern as determined by the RBML. Every pattern realization studied maintained a constant number of classes throughout its lifecycle. The extension of pattern realizations via inheritance or specialization rarely occurred.

All of the systems studied exhibited growth in total lines of code (LOC) for every design pattern realization. This growth is slight and occured in the method implementations. With the exception of only four design pattern realizations, all realizations across all systems studied exhibited little or no increase in the number of attributes. This stability in the number of attributes suggests that no new state, beyond what is expected as a responsibility of a design pattern, was added as the pattern evolved. In a similar manner, there was only a modest increase in the total number of methods in the Adapter and Visitor patterns in the JRefactory system. In ArgoUML, only the factory pattern classes exhibited a sharp increase in the total number of methods. In eXist, there were modest increases in the number of methods in Iterator pattern classes and a very sharp increase in Adapter pattern classes. Figure 4a–c shows the respective results for the total number of public methods in JRefactory, ArgoUML and eXist.

4.3 Organizational grime

Organizational grime gives a different perspective into the evolution of software systems. It shows what is happening to the packages and the physical files that participate in the collection of software artifacts that implement the realizations of design patterns. Afferent coupling measured at package levels is similar to afferent coupling measured at the pattern realization levels but at a higher abstraction level. This can be thought of as modular grime of packages that contain the implementation of the pattern. JavaNCSS gathers the afferent coupling measurements that operate at this level. We found little evidence of significant changes in afferent couplings at the package level. The number of files and packages that contain the implementation of the design patterns also showed no evidence of changes. We used *Pattern Seeker* and *Design Pattern Finder* to gather file statistics for various design patterns.

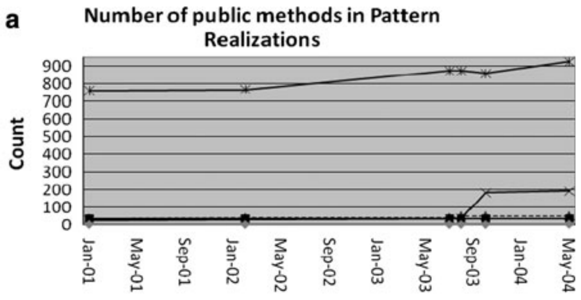
5 Discussion

We analyze the raw results described above and examine the consequences of the observed grime buildup on the testability and the adaptability of design pattern realizations. We evaluate the set of propositions posed in Sect. 3.3.

5.1 Modular, class, and organizational grime

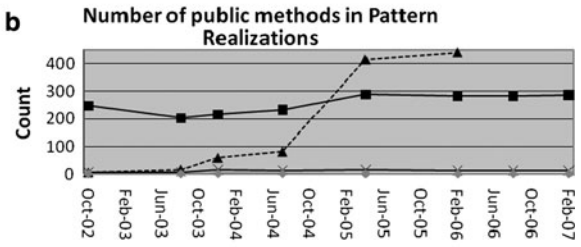
All systems studied exhibited increases in modular grime for almost every design pattern studied. Table 4 lists all the design patterns that were studied in each of the software

a



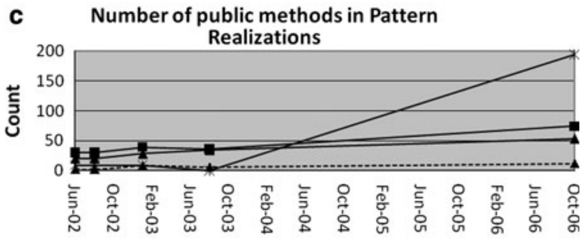
	Jan-01	Feb-02	Jul-03	Aug-03	Oct-03	May-04
---▲--- Factory	39	43	43	53	53	53
—✕— Adapter	27	30	35	40	183	192
—■— State	37	37	37	37	37	37
—◆— Singleton	2	2	2	2	2	4
—✳— Visitor	759	766	872	872	858	924

b



	Oct-02	Aug-03	Dec-03	Jul-04	Apr-05	Feb-06	Aug-06	Feb-07
---▲--- Factory	6	17	60	82	414	440		
—✕— Adapter	7	7	16	14	15	13	13	13
—■— State	247	203	215	233	289	282	283	287
—◆— Singleton	1	1	2	2	1	1	1	1

c



	Jun-02	Aug-02	Aug-02	Jan-03	Aug-03	Oct-06
---▲--- Factory	2	2	2	8	6	12
—✳— Adapter	9	9	9	9	0	194
—■— Iterator	30	30	30	39	36	74
—▲— Proxy	20	20	20	28	34	53

Fig. 4 **a** JRefactory number of public methods of all possible realizations of various design patterns, **b** ArgoUML number of public methods of all possible realizations of various design patterns, **c** eXist number of public methods of all possible realizations of various design patterns

Table 4 Observed growth of afferent/efferent coupling

	JRefactory	ArgoUML	eXist
Singleton	+ve/+ve	Flat/flat	
State	Flat/flat	+ve/flat	
Factory	+ve/+ve	Flat/+ve	Flat/+ve
Adapter	+ve/+ve	Flat/+ve	+ve/+ve
Visitor	+ve/+ve		
Iterator			+ve/+ve
Proxy			+ve/+ve

systems. Each design pattern is given a nominal evaluation that describes the growth of afferent and efferent coupling. A *Negative* evaluation means that afferent coupling tended to decrease as the pattern evolves. A *Flat* evaluation indicates that afferent coupling remained steady, and a *Positive* evaluation indicates growth in afferent coupling as the system evolved. Afferent and efferent coupling counts in all studied systems remained flat or exhibit a tendency to grow. Clearly additional patterns need to be observed. Studies by Basili et al. (1996) and Briand et al. (1997) provide evidence to support that coupling measures are useful indicators of quality. The results are the same after normalizing coupling counts by the number of realizations of design patterns in a system.

Class grime metrics provided no evidence indicating that the number of methods or attributes increased in a manner that is not consistent with the expected evolution of design patterns. The Adapter pattern realization in JRefactory did experience a sudden increase in the number of methods in October 2003; however, this was matched by an increase in LOC. We observed that only the Factory pattern realization experienced greater growth in LOC than that of the number of methods. The Visitor pattern in JRefactory had more methods and LOC counts than any other pattern studied. The growth in the number of methods in JRefactory's Visitor pattern realizations was consistent with the RBML of the Visitor pattern. This is because the increase in methods was primarily due to the addition of conformant *visit()* and *accept()* pair methods. In ArgoUML, the Adapter and Factory pattern realizations exhibited high growth in LOC until the July 2004 release. The LOC counts were quickly lowered after this release. Although grime was beginning to form, it appears that a refactoring process took place. The Factory pattern in ArgoUML exhibited a significant increase in the number of methods through all studied releases. The increase was matched by the increase in LOC until July 2004. After this release, the LOC count began to decrease while the number of methods increased, which suggests possible refactoring. In the eXist, we found no evidence of class grime for any participants of any class studied.

At an organizational level, there was not enough evidence to support grime buildup. Afferent coupling of all packages that participate in the implementation of design pattern realizations tended to remain constant, with only one exception: JRefactory after the August 2003 release. The relative adaptability of the packages that participated in design patterns did not appear to be affected by afferent coupling measured at this level.

5.2 Consequences of grime

5.2.1 Testability consequences of grime

In our prior exploratory pilot study (Izurieta and Bieman 2007), we found evidence of the formation of the following testing anti-patterns: the concurrent-use-relationship, swiss-army-knife, and self-use-relationships (depicted in Figs. 5, 6, 7). We also showed that the number of test requirements increased due to grime buildup. Concurrent-use-relationship anti-patterns (Baudry et al. 2002) manifest when transitive dependencies occur due to inheritance, which creates multiple paths to a class. Multiple paths increase the polymorphic complexity and reduce the testability of a pattern. The swiss-army-knife anti-pattern (also known as Kitchen Sink) forms when "the designer attempts to provide for all possible uses of the class," (Brown et al. 1998) by adding unrelated functionality to a single class. Finally, self-use-relationship anti-patterns (Baudry and Sunye 2004) form when classes access themselves via transitive usage dependencies.

We did not investigate whether anti-patterns actually affected readability and maintenance of designs; rather, we reported that the development of anti-patterns is a consequence of modular grime buildup in design patterns. In some cases, the formation of anti-patterns occured in the original design studied and remained throughout the evolution of the software. Anti-patterns that we found in the first release studied are classified as *foundational* when the first realization of a pattern studied has already undergone some

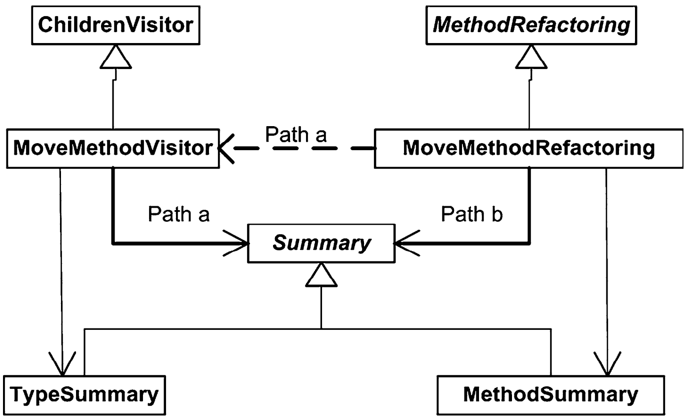


Fig. 5 Concurrent-use-relationship anti-pattern

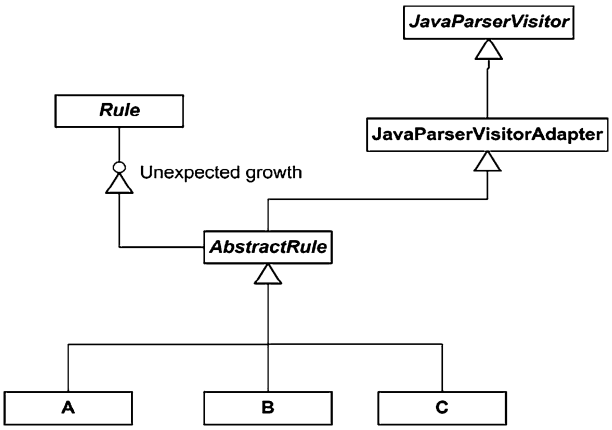


Fig. 6 Swiss-army-knife anti-pattern

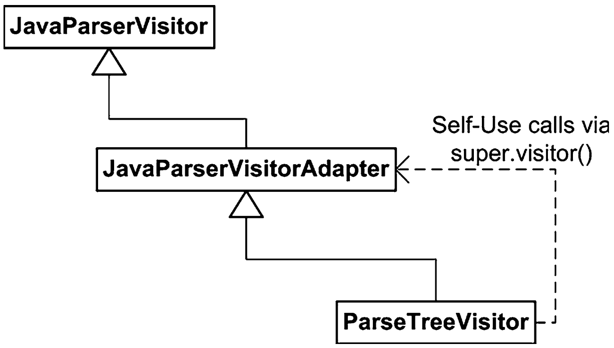


Fig. 7 Self-use-relationship anti-pattern

form of deterioration. A design is flawed if foundational anti-patterns are present in the first release. Figure 5 shows an example of the concurrent-use-relationship anti-pattern found in all versions of a single realization of the Visitor pattern. Paths a and b indicate two possible paths. Note that the *MoveMethodVisitor* class is also subject to polymorphic binding that increases the complexity by adding dynamic paths to the anti-pattern.

The "summary" hierarchy can be accessed via multiple paths. Baudry et al. (2001, 2002, 2003, 2004) describe the Visitor pattern as especially difficult to test because of polymorphism. Additional concurrent paths to a class make the testing more difficult.

In another example, we found evidence of the swiss-army-knife anti-pattern. This anti-pattern developed as a result of grime buildup. The *JavaParserVisitorAdapter* class did not develop until release 2.9.00 (2 years after the initial version). Additionally, the original design pattern was not intended to implement the methods in the *Rule* interface. As a result, the entire testing of the *AbstractRule* class was affected. Figure 6 illustrates the anti-pattern.

Finally, an example of the self-use-relationship is depicted in Fig. 7. Self-references create circular dependencies that are difficult to test. Self-Usage references occur because up to eight *ParseTreeVisitor visit()* methods call *super.visior()* before they execute their own logic creating circular dependencies. Due to polymorphism and the development of circular dependencies, the test requirements of the software increase dramatically.

We also observed that as patterns evolve, they develop relationships that break down their modularity. The modularity of a design pattern is optimal when only the necessary relationships (couplings) exist for the pattern to be compliant with its formal RBML description. Modular grime adds unnecessary couplings to a pattern realization, thus reducing its modularity. As grime buildup occurs over time, we evaluated the consequences that buildup has on the adequacy of test cases in terms of the number of test requirements. We counted the number of relationships (associations, interface realizations, and dependencies) that develop as a result of grime buildup and observed the consequences on the number of program elements that must be tested to adequately cover a design pattern. Relationship counts are tallied per individual class. In all cases studied, it is shown that test requirements increased as a result of modular grime buildup. Offutt and Amman (2008) describe how graphs can be used as abstractions to represent code, accordingly "a graph-based coverage criterion evaluates a test set for an artifact in terms of how the paths corresponding to the test cases cover the artifact's graph abstraction," thus, if the number of relationships that appear as a result of grime buildup increases in a pattern, it follows that the number of test cases also increase. Although design patterns do not exist to ensure testability of designs, a consequence of grime buildup does create more paths to test, and the numbers of paths that can be tested represent an upper bound value for testing effort. Equations for computing the number of test requirements are parameterized according to the number (k) of n-ary relationships between classes. In a binary relationship ($n = 1$) four possible combinations for each relationship must be tested (Binder 2000). Each end of a relationship is constrained by a minimum and a maximum multiplicity value, and for each combination, an *accept* and a *reject* test case is necessary, thus yielding eight possible scenarios. In the case of n-*ary* associations, $8n$ possible scenarios must be tested. The $8n$ scenarios cover the basic boundary conditions, but an additional constant number of tests can be added to cover typical scenarios present in operational profiles. The minimum number of n-*ary* association test cases necessary using the linear model is given by the following equation: $A(n,\ k) = 8nk + c,\ \ c \geq 0$. 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. The minimum number of test requirements is given by $\mathrm{AG}(n,\ k) = \mathrm{A}(n,\ k) + 4nk$.

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. The minimum number of test requirements is given by $C(n,\ k) = \mathrm{A}(n,\ k) + 2nk$. 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 generalization relationships that should be covered during testing is expressed as $G(n,\ k) = 2nk$.

Figure 8a–c depict results of the requirement counts for every realization of the patterns studied in JRefactory, ArgoUML, and eXist. The graphs display an aggregate of the total test requirements necessary to adequately test the patterns and include associations, generalizations, realizations, and dependencies. Since aggregation $\mathrm{AG}(n,\ k)$ and composition $C(n,\ k)$ are forms of associations, their counts include the number of associations $A(n,\ k)$ as part of their respective equations. We did not differentiate further between aggregation and composition. Thus, the total test requirements in Fig. 8a–c represent a lower bound of their count. For detailed results, see Izurieta and Bieman (2008).

5.2.2 *Adaptability consequences of grime*

5.2.2.1 Correlation and regression analysis A multiple linear regression model has been used to understand how changes in LOC of participating pattern classes are related to modular grime. We also use Spearman's correlation coefficient model to obtain correlation coefficients. Table 4 indicates that two out of the three systems studied show that a relationship is clearly possible; while the third system suggests that possibly a non-linear regression model may be necessary. We used Spearman's correlation coefficients because our samples were small and we need a non-parametric technique. The coefficients for coupling (afferent and efferent) showed that for JRefactory and eXist, there is a correlation with coupling; however, this was not evident in ArgoUML. According to Ott and

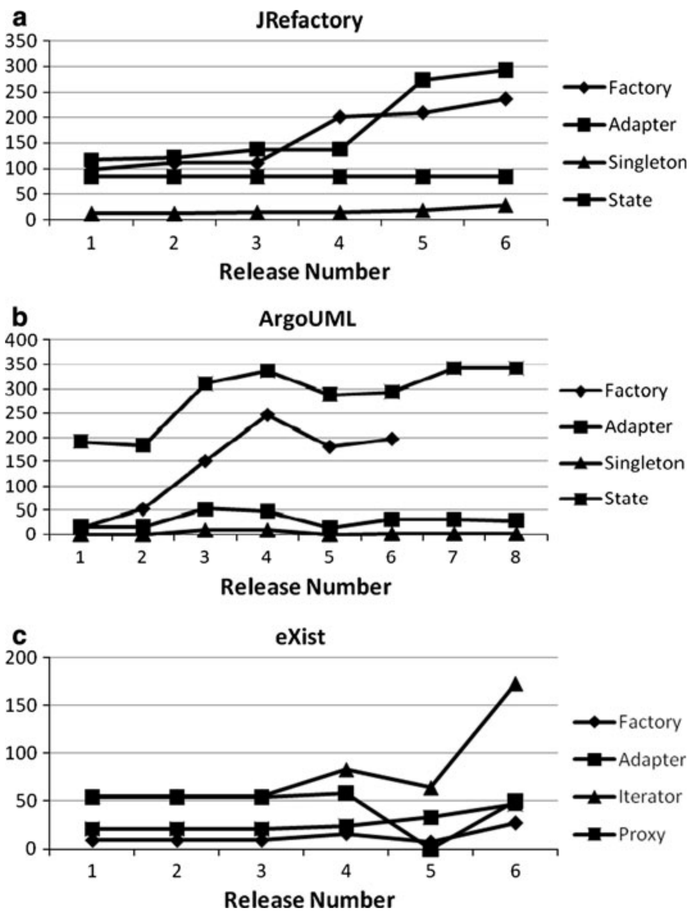


Fig. 8 **a** test requirement count of relationships (y-axis) for each of six releases of JRefactory, **b** test requirement count of relationships (y-axis) for each of eight releases of ArgoUML, **c** test requirement count of relationships (y-axis) for each of six releases of eXist

Longnecker (2001), anything greater than 0.60 implies that there is correlation. Table 5 suggests mixed results, with significant values ($p < 0.05$) highlighted in bold.

For each system under study, we applied multiple linear regression using independent variables C_a (afferent coupling) and C_e (efferent coupling). The regression was only used to identify a possible structural relationship between ΔLOC and afferent/efferent coupling that result from grime accumulation. The regression was not used to describe the predictive power that independent variables may have over the dependent variable. Regression analysis shows a statistically significant relationship occurs in eXist only; however, this may be due to a dominant change in LOCs from releases 0.9.2 to 1.0b1 (a 3-year period), and it is deemed a threat to the construct validity of the study. Thus, results indicate that not all changes to design patterns are necessarily grimy. We can summarize that modular grime counts correlate with changes in LOC in eXist and to a lesser extent JRefactory. However, the relationship between modular grime and changes in LOC in ArgoUML are not significant. This result suggests that not all changes to code in design patterns represent decay and that some changes represent improvements or new functionality.

Table 5 Spearman's correlation coefficients for afferent (C_a) and efferent (C_e) coupling versus changes in LOC with corresponding *p*-values in parentheses

	Singleton	Factory	Adapter	Visitor
JRefactory				
C_a	0.78 (0.06)	0.43 (0.38)	**0.97 (<0.01)**	0.08 (0.87)
C_e	0.77 (0.07)	0.51 (0.29)	**0.88 (0.02)**	0.65 (0.15)
	Iterator	**Factory**	**Adapter**	**Proxy**
eXist				
C_a	**0.93 (<0.01)**	−0.20 (0.7)	0.35 (0.49)	**0.95 (<0.01)**
C_e	**0.93 (<0.01)**	**0.89 (0.01)**	**0.89 (0.01)**	**0.95 (<0.01)**
	Singleton	**Factory**	**Adapter**	**State**
ArgoUML				
C_a	−0.12 (0.77)	0.31 (0.54)	0.69 (0.05)	0.51 (0.19)
C_e	0.02 (0.96)	−0.02 (0.95)	0.30 (0.46)	0.19 (0.64)

5.2.2.2 Instability analysis Average instability values for design patterns were calculated using the pattern instability metric described in Sect. 2.4, and tended to be greater than 0.60 (on average) in a range of [0–1]. Efferent coupling levels are higher than afferent coupling levels, and this suggests that the realizations of design patterns have not had a chance to be used by many clients. An instability value of zero indicates a mature pattern and is very hard to remove from its setting because the afferent counts are very high. High instability ratios may be a consequence of the short history of all the systems under study. We found no evidence of stable design patterns as indicated by the measured ratio with the exception of JRefactory's Singleton pattern. More mature systems may exhibit a different and more balanced trend.

6 Assessment of propositions

6.1 Decay, rot, and grime buildup

$\mathbf{P}_{1,0}$ Design pattern realizations do not *rot* as systems evolve

We cannot reject $\mathbf{P}_{1,0}$. Every realization of the design patterns studied showed no evidence of pattern rot. Every realization was manually checked for compliance against its RBML, and we found that while grime accumulated, the core structure of the design pattern remained throughout all releases. We manually checked realizations for five design patterns in JRefactory, four design patterns in ArgoUML, and four design patterns in eXist. In all cases, we did not find any instances of pattern rot.

$\mathbf{P}_{2,0}$ Design pattern realizations do not exhibit increases in *grime buildup* as systems evolve

We have evidence to support rejecting $\mathbf{P}_{2,0}$. We observed modular grime buildup and the deterioration of the environment surrounding pattern realizations in JRefactory and eXist. To a lesser extent we observed this in ArgoUML. In JRefactory and eXist, we observed that realizations continued to gather modular grime buildup at a steady rate; however, ArgoUML appeared to go through periods of refactoring that lowered modular grime before it increased slightly again. We have not tried to fit a curve to any data of the limited number of data points.

$\mathbf{P}_{3,0}$ The number of pattern realizations do not increase as the system evolves over time

We have evidence to support rejecting $\mathbf{P}_{3,0}$. In general, the total number of realizations increased in every system for all patterns studied, but at a slow pace

6.2 Class grime buildup

$\mathbf{P}_{4,0}$ Increases in the number of public methods in a class that belongs to a pattern are not significant

Evidence does not support rejecting $\mathbf{P}_{4,0}$. In JRefactory only the Adapter pattern showed signs of growth in the number of methods. The eXist system also exhibited growth in the number of methods in Adapter patterns as well as the Iterator and Proxy patterns; however, the growth in terms of methods only occured in the last revision of software, and thus, there is little history to support sustained growth. Prior to the last released version, we saw no evidence of such growth. ArgoUML exhibited measurable growth in the Factory pattern realizations. The observed growth in the total number of methods was matched by growth in the LOC for the respective patterns.

$\mathbf{P}_{5,0}$ Increases in public attributes (fields) in a class that belongs to a pattern are not significant

Evidence does not support rejecting $\mathbf{P}_{5,0}$. In JRefactory, all patterns appeared to maintain a constant number of attributes with the exception of the Adapter and Visitor patterns. The eXist system also exhibited restrained growth in the number of attributes. Most of the growth is also attributed to the last version of software studied. Finally, in ArgoUML, we only observed a steady growth in the State pattern realizations; however, these realizations also saw an increase in the number of classes that participate in the pattern realizations, thus bringing the average number of attributes per class down.

6.3 Modular grime buildup

$\mathbf{P}_{6,0}$ Increases in afferent pattern coupling $\mathrm{PatC}_{\mathbf{a}}$ (fan-in) are not significant

We have evidence to support rejecting $\mathbf{P}_{6,0}$. While the observed growth in afferent coupling levels was less significant than expected, we nevertheless found that afferent coupling in every design pattern studied in every system exhibited either a tendency to remain flat or show positive growth. In ArgoUML, we observed more restrained afferent coupling levels than the other systems, which may be an indication of better engineering techniques. Evidence suggests that afferent levels are not inconsequential, and we would expect to find higher levels in less successful systems.

$\mathbf{P}_{7,0}$ Increases in efferent pattern coupling $\mathrm{PatC}_{\mathbf{e}}$ (fan-out) are not significant

We have strong evidence to support rejecting $\mathbf{P}_{7,0}$. While the Singleton pattern showed no changes in efferent coupling levels (expected), all other realizations showed positive growth. Growth levels tended to show much sharper increases that their afferent level counterparts.

6.4 Organizational grime buildup

$\mathbf{P}_{8,0}$ Increases in organizational grime are not significant

There is no evidence to reject $\mathbf{P}_{8,0}$. Afferent levels at the package level as well as the number of files and packages did not provide significant evidence to suggest grime buildup

6.5 Consequences of grime buildup

$\mathbf{P}_{9,0}$ There is no significant correlation between changes in lines of code (LOC) and design pattern grime buildup

Results are inconclusive. For both JRefactory and eXist, we have some evidence that changes in LOC led to an increase in modular grime levels. Some patterns studied in these systems showed significant *p*-values and correlation coefficients that help support rejecting this proposition. ArgoUML, however, did not have significant results to help reject $\mathbf{P}_{9,0}$.

$\mathbf{P}_{10,0}$ The adaptability of design patterns, measured by pattern instability, $\mathrm{PatC}_{\mathbf{e}}/(\mathrm{PatC}_{\mathbf{a}} + \mathrm{PatC}_{\mathbf{e}})$ tends to remain unchanged as patterns evolve

Evidence did not support rejecting $\mathbf{P}_{10,0}$. We expected that as patterns evolved and modular grime built up that the afferent coupling of a design pattern would increase at a higher rate than its efferent coupling thus reducing the pattern's adaptability. High levels in efferent coupling compared to afferent coupling caused the ratio values to trend higher. On average, instability values were greater than 0.60. The Singleton pattern in JRefactory was the only exception with instability values averaging just above 0.20.

$\mathbf{P}_{11,0}$ There will be similar effects of grime buildup on the testability and adaptability of design patterns

Evidence supported rejecting $\mathbf{P}_{11,0}$. The instability of design patterns tends to be higher than 0.60. This result indicates higher efferent levels than afferent levels, and efferent values affect testability of systems. Intuitively, a higher fan-out value indicates dependencies on other parts of the system, which implies that additional test cases are necessary to adequately verify the functionality of a pattern. There is less impact on the adaptability of pattern realizations because their afferent coupling grows at a slower pace than their efferent coupling. The number of test requirements increase in all systems studied, as indicated by the measures derived in our prior work (Izurieta and Bieman 2008).

$\mathbf{P}_{12,0}$ Grime build-up does not affect the number of test requirements

We can reject $\mathbf{P}_{12,0}$. We found clear evidence that test requirements increase as a result of higher efferent levels. We found high efferent levels on almost all pattern realizations of every system under study. Test requirements grew at different pace depending on whether efferent coupling was associated with composition, dependency, generalization, or association relationships.

Table 6 provides a summary of the assessments of all propositions.

Table 6 Propositions summary

Proposition number	Proposition description	Assessment evidence suggests:
Decay, rot, and grime buildup		
$\mathbf{P}_{1,0}$	Design pattern realizations do not *rot* as systems evolve	Cannot reject
$\mathbf{P}_{2,0}$	Design pattern realizations do not exhibit increases in *grime buildup* as systems evolve	Reject
$\mathbf{P}_{3,0}$	The number of pattern realizations do not increase as the system evolves over time	Reject
Class grime buildup		
$\mathbf{P}_{4,0}$	Increases in the number of public methods in a class that belongs to a pattern are not significant	Cannot reject
$\mathbf{P}_{5,0}$	Increases in public attributes (fields) in a class that belongs to a pattern are not significant	Cannot reject
Modular grime buildup		
$\mathbf{P}_{6,0}$	Increases in afferent pattern coupling $PatC_{\mathbf{a}}$ (fan-in) are not significant	Reject
$\mathbf{P}_{7,0}$	Increases in efferent pattern coupling $PatC_{\mathbf{e}}$ (fan-out) are not significant	Reject
Organizational grime buildup		
$\mathbf{P}_{8,0}$	Increases in organizational grime are not significant	Cannot reject
Consequences of grime buildup		
$\mathbf{P}_{9,0}$	There is no significant correlation between changes in lines of code (LOC) and design pattern grime buildup	Inconclusive
$\mathbf{P}_{10,0}$	The adaptability of design patterns, measured by pattern instability, $PatC_{\mathbf{e}}/(PatC_{\mathbf{a}} + PatC_{\mathbf{e}})$ tends to remain unchanged as patterns evolve	Cannot reject
$\mathbf{P}_{11,0}$	There will be similar effects of grime buildup on the testability and adaptability of design patterns	Reject
$\mathbf{P}_{12,0}$	Grime build-up does not affect the number of test requirements	Reject

7 Related work

Establishing relationships between external quality attributes of designs and internal measures is difficult. External attributes are affected by additional factors such as availability of adequate development tools, programmer experience, schedule constraints, scope of tasks. Also establishing a relationship does not necessarily imply causality, making this a difficult problem.

Measures of many software attributes have been studied in procedural paradigms. Rombach (1990) shows how some of these internal measures can predict maintainability in a system. Measures of attributes of object-oriented systems first appeared in the early 1990s. Chidamber and Kemerer (1991) developed six measures in their work: depth of inheritance (DIT), Coupling between objects (CBO), number of children (NOB), response for a class (RFC), lack of cohesion (LCOM), and weighted methods per class (WMC). Results from Li and Henry (1993) validate a relationship between Chidamber and Kemerer's metrics and maintenance effort as measured by the number of changes made to classes.

Empirical studies link increased coupling to negative effects on the adaptability and maintenance of systems. Arisholm and Sjoberg (2000) focus on measuring *changeability* *decay*, which they define as the increased effort required to implement changes in object-oriented systems. Subsequent work by Arisholm et al. (2004, 2006) shows relationships between structural attributes of object-oriented systems and development effort. The goal of Arisholm et al. is to assess the extent to which structural properties of object-oriented systems affect changeability. They investigate static structural attributes as well as *change profile* measures. The latter depend on the former but are adjusted by the proportion of change experienced by the corresponding structural measure. Arisholm et al. state that "change profile measures cannot be used as early indicators of changeability. Instead, their intended use is to indicate trends in changeability during the evolutionary development and maintenance of a software system." (Arisholm 2006)

Basili et al. (1996) use the probability of fault detection as a surrogate measure for fault proneness. The goal of their observational study was to assess Chidamber and Kemerer's object-oriented metrics as predictors of fault-prone classes. They collected data on eight C++ information management systems that were carefully setup in a university environment, and evaluated hypotheses for each metric. Relevant to this research are the H-WMC (Weighted Methods per Class) and the H-CBO (Coupling Between Objects) hypotheses. H-WMC states that "A class with significantly more member functions than its peers is more complex and, by consequence, tends to be more fault-prone." H-CBO states that "Highly coupled classes are more fault-prone than weakly coupled classes because they depend more heavily on methods and objects defined in other classes." Univariate and multivariate logistic regression show that there is a significant relationship between these measures and fault proneness. WMC and CBO are two among five measures that are shown to be good predictors of faults.

Briand et al. (1997) show that coupling measures are useful indicators of quality in object-oriented designs. Their study also uses Chidamber and Kemerer's object-oriented metrics; however, they enhance the suite of C++ measures to include, among others, export and import coupling. Differentiating between coupling measures provides better insights into which type of coupling is more likely to increase error density and maintenance costs of designs. The study finds that import and export coupling measures appear to be significant predictors of fault proneness. Import and export coupling measures do not include generalization or specification relationships between classes.

A study by Cain and McCrindle (2002) finds a link between class coupling and team productivity; "if software is improperly coupled then people are improperly coupled." They find that unmanaged growth in coupling measures will slow programmers due to the inability of the programmers to work in isolation. Cain and McCrindle make a distinction in the direction of the coupling and state that a high fan-out value indicates instability, whereas a high fan-in value indicates responsibility. A class lacks stability when it references many external classes, making the class susceptible to faults. A class has high responsibility when many external classes depend on it. A system with high coupling levels (responsibility and instability) exposes poor information hiding and thus has repercussions on team dynamics.

Further, negative effects of coupling can have ripple effects on more distant classes. Briand et al. (1999) study the relationship between coupling and ripple effects. Directions of coupling between any pair of classes are given equal weight. This is because ripple changes can propagate in any direction along a coupling dimension. The investigation used coupling measures for identifying classes likely to contain ripple changes when another class is changed. They find that some coupling measures are related to a higher probability of common code changes and that coupling measures are good indicators of ripple effects and can be used in decision models for ranking classes according to their probability to contain ripple effects.

Studies in software evolution and its consequences on designs are ongoing. Significant prior work has increased our understanding of how evolution affects object-oriented designs and of the consequences to external quality attributes. Evolution studies have also prompted the investigation of the deterioration that occurs in designs. In particular, many empirical studies Arisholm (2006), Briand et al. (1997), and Basili et al. (1996) investigate possible relationships between independent object-oriented measures and the negative effects they may have on design quality. The last few years has seen a resurgence of understanding the notion of *technical debt* (Cunningham 1992). Technical debt is a metaphor for learning how short decisions that negatively impact a design can have long-term effects. Studies by Guo et al. (2010) and Brown et al. (2010) present many open questions that remain open today, for example, "How can one identify debt in a software development project and product, perhaps automatically?," "What are the kinds of debt?," and "How can information about technical debt be collected empirically for developing conceptual models?." The research presented in this study is a step toward understanding how to empirically measure conditions that affect long-term technical debt.

This research focuses not just on quantifying object-oriented decay, but takes us a step further toward understanding the differences between decay, rot, and grime buildup in design patterns as a result of evolution. Design patterns have a well-understood structure. Thus, they are a natural subject for this study of design evolution. We can use this structure to accurately measure changes. However, the existing definitions of design patterns are informal. Formal design pattern languages can make the analysis of design pattern integrity more precise.

8 Threats to validity

In this section, we analyze the threats to the validity of the study. There are four types of threats to the validity of any empirical study. They are internal, external, construct, and content threat validity.

Internal validity focuses on the cause and effect relationships. Data collected in this study does suggest that grime and decay are responsible for an increase in test suite numbers and the appearance of anti-patterns. However, while we have evidence that shows that there is a statistically significant correlation between decay and the changes in LOC, temporal precedence cannot be clearly established. It is possible that grime accumulates as a result of changes made to code or that the changes in LOC are indeed responsible for the accumulation of grime. We only find evidence to suggest a structural relationship. Temporal precedence is a necessary condition when examining internal validity of a system.

External validity in case studies refers to the ability to analytically (rather than statistically) generalize results (Yin 2003), and we suggest that additional case studies are necessary to provide support for our theory that patterns decay mainly as a result of grime buildup rather than rot. Further studies of additional systems, additional design patterns, and additional grime and rot measures are required.

Construct validity refers to the meaningfulness of measurements. To validate construct validity you must show that the measurements are consistent with an empirical relation system. An empirical relation system is an intuitive ordering of the data in terms of the attributes of interest. The dependent variables in this study are grime and rot. Since we find grime to be the dominant reason for decay, and further, modular grime to be the most dominant cause of grime, we measure this dependent variable using the number of relationships that patterns develop. We can say, for example, that the Visitor pattern appears to suffer from more decay and grime than the Singleton pattern in the JRefactory system, and thus consequences are higher. We can make similar observations in the other systems studied; however, a more comprehensive validation requires the study of more instances of these patterns in other systems need. We also only mined for intentional patterns whose names were clearly present as strings or sub-strings in the participating classes of the realization. Some pattern realizations may not bear any identifiable string as part of their name, which may explain why we find grime (as opposed to rot) to be the dominant form of decay. Mining for unintentional patterns is beyond the scope of this study. When calculating test requirement growth as a result of the development of harmful relationships, we aggregated associations, aggregations, and composition. While this reduces the precision of the counts, it does not conflict the empirical relation system.

Finally, content validity refers to the adequate representation of the content. In order for this study to capture the notions of decay, grime, and rot further, we need to investigate additional independent variables of modular, class, and organizational measures. Finer granularity of measures and improved metrics are necessary. We have used manual verification of patterns and customized queries written in the .QL language to identify harmful grime, however; this is subjective and better automated means are necessary to clearly distinguish between added logical features and grime. We have also not investigated the potential roles that a class can have when participating in multiple design patterns (Khomh et al. 2009; McNatt and Bieman 2001). Additionally, when mining the eXist system, we included a pre-release version in the study, and we also observed a period of 3 years between two releases. This makes eXist a less than ideal subject for content selection and poses a construct validity threat. Additional systems need to be investigated.

9 Conclusions

It is not possible to stop the aging and deterioration of design pattern realizations. Evidence suggests that as design patterns age, the realizations of patterns remain and modular grime builds up. This exploratory multiple case study showed that decay is due mainly to grime rather than rot. We have defined new terms for understanding and quantifying decay, rot, and grime buildup in software design patterns, we have shown, through a multiple case study using three open-source systems, that modular grime buildup does occur as design patterns evolve, and we have shown that grime buildup can have negative and adverse consequences on the testability and the adaptability of design patterns. Clearly, alternative solutions to design patterns could be used to achieve similar functionality, but if a pattern is introduced in an early stage of the system, then we can only conjecture that the designer had this in mind for some reason. This study only measured how patterns decay over time. We did not measure "global system quality," but if the evolution of design patterns were to be used as a partial factor to measure global system quality, then we would venture to say that if design patterns decay, then the overall quality of a system also suffers.

We have carefully selected measures that provided an intuitive idea about what is happening to design patterns as they evolve, and we have chosen these measures to help us quantify our definitions of class, modular, and organizational grime. We followed a strict protocol (Yin 2003) to gather statistics on all pattern realizations under study in order to help assess the consequences of decay. We found no significant evidence of class or organizational grime; however, we did find evidence that grime buildup was mostly due to the increased coupling observed in the classes that participate in the realizations of design patterns, that is, modular grime.

We evaluated twelve propositions and in general found no evidence of rot, and significant evidence of modular grime buildup. We observed realizations of five design patterns in JRefactory, four design patterns in ArgoUML, and four design patterns in eXist. The grime buildup was mostly modular. We observed modular grime buildup in JRefactory and eXist. To a lesser extent, we observed this in ArgoUML. In JRefactory and eXist, we observed that realizations continued to gather modular grime buildup at a steady rate; however, ArgoUML appeared to go through periods of refactoring that lowered modular grime before seeing slight increases again. Class and organizational grime levels were not significant. Testability and adaptability of design patterns were important quality attributes. We found that modular grime buildup affects testability to a higher degree than adaptability. A high pattern fan-out (efferent) count indicates dependencies on other modules, and a high fan-in (afferent) count is indicative of a pattern with more responsibility. Growth in the *instability* measure, which tracks the ratio of efferent to afferent coupling, indicated that efferent coupling is increasing faster than afferent coupling. The effect is a decrease in testability, since covering all the new relationships in the evolving patterns requires additional test cases. Adaptability was evaluated by studying the relationship that exists between changes to lines of code and the buildup of coupling. Spearman's correlation coefficients and multivariate regression showed that a relationship exists for most realizations in JRefactory and eXist. ArgoUML did not reveal a relationship. While the existence of a relationship between changes to LOC and coupling metrics does not indicate causality, this was surprising considering the small sample sizes.

We studied three real world open-source systems: JRefactory, ArgoUML, and eXist. Clearly additional systems and design patterns must be studied. Additionally, we must increase the number of pattern realizations studied and the total number of releases.

Our characterization of decay and grime showed how grime builds up around design patterns. We identified various forms of grime. The removal of grime, as it appears, can potentially control some aspects of software maintenance costs, and improve adaptability and test effectiveness of systems. Developing refactoring techniques to contain grime is a natural progression of this research.

References

Altova Umodel. (2006). Altova. http://www.altova.com. Used 2008, 2009.

eXist Database. (2008). http://exist-db.org. Accessed 2008.

JDepends Open Source Software. (2008; 2009). http://www.clarkware.com/software/JDepend.html. Used 2008, 2009.

JRefactory Open Source Software. (2008). http://jrefactory.sourceforge.net. Accessed 2008.

PatternSeeker Tool. (2008). Colorado State University. Accessed 2008.

Semmle Query Technology. (2008; 2009). http://semmle.com. Used 2008, 2009.

ArgoUML. (2008). http://argouml.tigris.org. Accessed 2008.

Arisholm, E. (2006). Empirical assessment of the impact of structural properties on the changeability of object-oriented software. *Information and Software Technology, 48*(2006), 1046–1055.

Arisholm, E., Briand, L. C., & Foyen, A. (2004). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering, 30*(8), 491–506.

Arisholm, E., & Sjoberg, D. (2000). Towards a framework for empirical assessment of changeability decay. *The Journal of Systems and Software, 53*(1), 3–14.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering, 22*(10), 751–761.

Baudry, B., & Sunye, G. (2004). Improving the testability of UML CLASS DIAGRAMS. In *First international workshop on testability assesment, 2004, IWoTA 2004, proccedings*, November 2004, pp. 70–80.

Baudry, B., Traon, Y., & Sunye, G. (2002). Testability analysis of a UML class diagram. In *Software metrics symposium*, Ottawa, Canada, June 2002, pp. 54–63.

Baudry, B., Traon, Y., Sunye, G., & Jezequel, J. M. (2001). Towards a safe use of design patterns to improve OO software testability. In *Proceedings of the 12th international symposium on software reliability engineering, ISSRE'01*, p. 324.

Baudry, B., Traon, Y., Sunye, G., & Jezequel, J. M. (2003). Measuring and improving design patterns testability. In *9th international software metrics symposium,* September 2003, pp. 50–59.

Belady, L. A., & Lehman, M. M. (1971). Programming system dynamics, or the meta-dynamics of systems in maintenance and growth. Technical Report. IBM Thomas J. Watson Research Center.

Bieman, J. M., Straw, G., Wang, H., Munger, P. W., & Alexander, R. (2003). Design patterns and change proness: An examination of five evolving systems. In *Proceedings of ninth international software metrics symposium (metrics 2003)*, pp. 40–49.

Binder, R. V. (2000). Testing object oriented systems. Models, patterns, and tools. Reading: Addison-Wesley Publishers.

Briand, L. C., Devanbu, P., & Melo, W. (1997). An investigation into coupling measures for C++. In *Proceedings of the international conference of software engineering, ICSE'97*, Boston, MA.

Briand, L. C., Wust, J., & Lounis, H. (1999). Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE international conference on software maintenance*, p. 475.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kin, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., & Zazworka, N. (2010). Managing technical debt in software reliant systems. FoSER 2010, November 7–8, Santa Fe, New Mexico, USA.

Brown, W. J., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). *Anti patterns. Refactoring software, architectures, and projects in crisis*. New York: Wiley.

Cain, J. W., & McCrindle, R. J. (2002). An investigation into the effects of code coupling on team dynamics and productivity. In *Proceedings of the 26th annual international computer software and applications conference*, *COMPSAC '02*.

Chidamber, S. R., & Kemerer, C. F. (1991). Towards a metrics suite for object-oriented design. In *Proceedings: OOPSLA 1991*, pp. 197–211.

Cunningham, W. (1992). The WyCash portfolio management system. OOPSLA'92 Experience Report, 1992.

Design Pattern Finder (2008; 2009). http://www.codeplex.com/DesignPatternFinder. Used 2008, 2009.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering, 27*(1), 1–12.

Fenton, N. E., & Pfleeger, S. L. (1996). *Software metrics: A rigorous and practical approach*. London: PWS, Computer Press.

France, R. B., Kim, D. K., Song, E., & Ghosh S. (2002). Metarole-based modeling language (RBML) specification V1.0.

France, R. B., Kim, D. K, Song, E., & Ghosh, S. (2004). A UML-based pattern specification technique. *IEEE Transactions On Software Engineering, 30*(3), 193–206.

Freeman, E., & Freeman, E. (2004). *Head first design patterns*. Sebastopol, CA: O'Reilly Media.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object oriented software*. Reading, MA: Addison-Wesley.

Gilb, T. (1988). *Principles of software engineering management*. England: Addison Wesley.

Godfrey, M. W., & Tu, Q. (2000). Evolution in open source software: A case study. In *Proceedings of the 2000 international conference on software maintenance (ICSM 2000)*. San Jose, California, October 2000.

Guéhéneuc, Y. G., & Albin-Amiot, H. (2001). Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the 39th international conference and exhibition technology of object oriented languages and systems*, pp. 296–305.

Guo, Y., Seaman, C., Zazworka, N., & Shull, F. (2010). Domain-specific tailoring of code smells: An empirical study. ICSE'10, May 2–8, Cape-Town, South Africa.

Henry, S., & Kafura, D. (1984). The evaluation of software systems' structure using quantitative software metrics. *Software: Practice and Experience, 14*, 561–573. doi:10.1002/spe.4380140606.

Izurieta, C., & Bieman, J. M. (2007). How software designs decay: A pilot study of pattern evolution. In *1st ACM-IEEE international symposium on empirical software engineering and measurement, ESEM'07*, Madrid, Spain, September 2007.

Izurieta, C., & Bieman, J. M. (2008). Testing consequences of grime buildup in object oriented design patterns. In *1st ACM-IEEE international conference on software testing, verification and validation, ICST'08,* Lillehamer, Norway, April 2008.

JavaNCSS. (2008; 2009). http://www.kclee.de/clemens/java/javancss. Used 2008, 2009.

Khomh, F., Guéhéneuc, Y. G., & Antoniol, G. (2009). Playing roles in design patterns: An empirical descriptive and analytic study. In *Proceedings of the 25th IEEE international conference on software maintenance (ICSM)*, September 20–26, 2009, Edmonton, Alberta, Canada. IEEE Computer Society Press.

Kim, D. K. (2004). A meta-modeling approach to specifying patterns. Colorado State University PhD Dissertation, June 21, 2004.

Kim, D. K., & Shen, W. (2008). Evaluating pattern conformance of UHL models: A divide-and conquer approach and case studies. *Software Quality Journal, 16*(3), 329–359.

Kouskouras, K. G., Chatzigeorgiou, A., & Stephanides, G. (2008). Facilitating software extension with design patterns and aspect-oriented programming. *Journal of Systems and Software, 81*, 1725–1737.

Lehman, M. M. (1980). Program life cycles and laws of software evolution. *IEEE Special Issue on Software Engineering, 68*(9), 1060–1076.

Lehman, M. M., Perry, D. E., & Ramil, J. F. (1998). Implications of evolution metrics on software maintenance. In *Proceedings of the 1998 international conference on software maintenance (ICSM'98)*, November 1998, Bathesda, Maryland.

Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of Systems and Software, 23*, 111–122.

Martin, R. C. (2002). *Agile software development, principles, patterns, and practices*. Englewood Cliffs, NJ: Prentice Hall.

McNatt, W. B., & Bieman, J. M. (2001). Coupling of design patterns: common practices and their benefits. In *Proceedings of the 25th computer software and applications conference*. IEEE Computer Society Press, October 2001, pp. 574–579.

Offutt, J., & Amman, P. (2008). *Introduction to software testing* (p. 27). Cambridge: Cambridge University Press.

Ott, R. L., & Longnecker, M. (2001). *An introduction to statistical methods and data analysis* (5th ed.). Pacific Grove, CA: Duxbury.

Parnas, D. L. (1994). Software ageing. Invited plenary talk. In *16th international conference (ICSE'94)*, pp. 279–287, May 1994.

Pree, W. (1994). Meta patterns-a means for capturing the essentials of reusable object oriented design. *Lecture Notes in Computer Science, 821*, 150–162.

Rombach, H. D. (1990). Design measurement: Some lessons learned. *IEEE Software, 7*, 17–25.

Schull, F., Melo, W., & Basili, V. (1996). An inductive method for discovering design patterns from object oriented software systems. Technical Report UMCP-CSD CS-TR-3597 or UMIACS-TR-96-10, University of Maryland, Computer Science Department.

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010). Building empirical support for automated code smell detection. ESEM'10, September 16–17, 2010, Bolzano-Bozen, Italy.

Van Gurp, J., & Bosch, J. (2002). Design erosion: Problems and causes. *Journal of Systems and Software, 61*, 105–119.

Yin, R. K. (2003). *Case study research: Design & methods* (3rd ed.). Thousand Oaks: Sage Books.

Author Biographies

Clemente Izurieta is an assistant professor of computer science at Montana State University. Dr. Izurieta has over 16 years of experience developing and architecting commercial software systems for Hewlett Packard Co. and Intel. His work involves empirical research and the evolution of software designs, as well as studies in ecological modeling and cyber infrastructures.

James M. Bieman is a professor of computer science at Colorado State University. His work is focused on the evaluation and improvement of software design quality. In particular, he is studying whether early design decisions match demands for change as systems evolve.