DISSERTATION

DECAY AND GRIME BUILDUP IN EVOLVING OBJECT ORIENTED DESIGN PATTERNS

Submitted by

Clemente Izurieta

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2009

COLORADO STATE UNIVERSITY

April 29, 2009

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

Committee on Graduate Work

Sudipto Ghosh

Committee Member: Dr. Sudipto Ghosh

Ross McConnell

Committee Member: Dr. Ross McConnell

Committee Member: Dr. Alexander Hulpke

Adviser: Dr. James M. Bieman

Department Head: Dr. L. Darrell Whitley

ABSTRACT OF DISSERTATION

DECAY AND GRIME BUILDUP IN EVOLVING OBJECT ORIENTED DESIGN PATTERNS

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

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

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

ACKNOWLEDGEMENT

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

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

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

DEDICATION

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

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

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

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

Design patterns provide an initial point of reference that can be extracted from a design and whose evolution can be followed. It is a common belief that software designs decay over time and we want to determine the extent of decay as it pertains to design patterns. We show that as a consequence, the adaptability and testability of designs deteriorate, and that as systems built with design patterns evolve, added non-pattern code, or "grime", affects pattern participants of individual design patterns.

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

1.1 Approach

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

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

1.2 Contribution

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

2. BACKGROUND

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

2.1 Software Evolution

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

The study of software evolution began in earnest in the late 1960s. Lehman [59] introduced the study of software evolution. Belady and Lehman [10] first studied the evolution of the OS/360 system. The data produced by this early study prompted Lehman to propose the first three laws of software evolution. There are now eight laws which continue to be revised. Lehman conducted numerous studies on commercial E-type systems, where an E-type system is described by Scacchi [77] as "an application embedded in its setting of use." In other words, there is feedback between an E-type system and its operating environment causing continued evolution of the software system. Lehman's feedback ontology comes from the domain of control systems, which is traditionally tied to the physical sciences rather than software engineering. The feedback ontology's use to model software evolution does not take into account ontologies found in fields such as social networks, cultural demography, and lately communities of developers such as those found in the open source movement. The second and seventh laws are directly related to software decay. Lehman's second law of evolution—*Increasing Complexity* —states that as a system evolves, so does its complexity as measured by the size of the system. Lehman's seventh law of evolution—*Declining Quality* —describes how E-type systems, unless rigorously maintained, are perceived as having declining quality.

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

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

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

Many prior studies examine software evolution with a focus on overall size measurements of the subject systems. For example, Godfrey and Tu [38] studied evolution in terms of the size of the Linux operating system. They "expected to find that Linux was growing more slowly as it got bigger and more complex." They found that certain sub-systems are growing at a super linear rate, which contradicted results from commercial systems. Super linear growth was only found in driver sub-systems of the development releases. Another study by Izurieta and Bieman [49] found no evidence of super-linear growth, even in the driver sub-systems. The latter study was carried out on stable releases of Linux and FreeBSD.

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

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

Mattsson and Bosch [65] observe the evolution of Object Oriented Frameworks (OOF). Frameworks provide a base for building applications of a similar domain. They study three frameworks from commercial and open source domains. Their studies examine three methods that allow management to make better decisions based on the evolution of frameworks. They identify change prone modules.

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

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

2.2 Maintenance and Decay

Evolution is directly tied to changes to a system that occur over time. Belady and Lehman [10] assert that it is impossible to inject new code to older systems without introducing new defects or faults, which represents decay. Software evolution involves changes to designs, architectural specifications, and code as a result of fault repairs, new or changing requirements for additional functionality, improved performance, improved storage and memory usage, etc. These changes, which are implemented under time and resource constraints, contribute to the decay of a system. Additionally, these changes affect the test effectiveness, adaptability, reliability, and performance of a system. Fenton and Pfleeger [30] classify changes to software into various categories, but design decay is specifically related to the adaptive and corrective categories. These types of changes are dependent on external environment changes and bug reports, hence our interest in finding characterizations of negative changes (decay) to a system.

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

Figure 2.1 from Izurieta and Bieman [49] displays the inverse growth functions for various stable releases of the Linux operating system. The growth functions are plotted per release, and clearly show evidence to support Turski's claim.

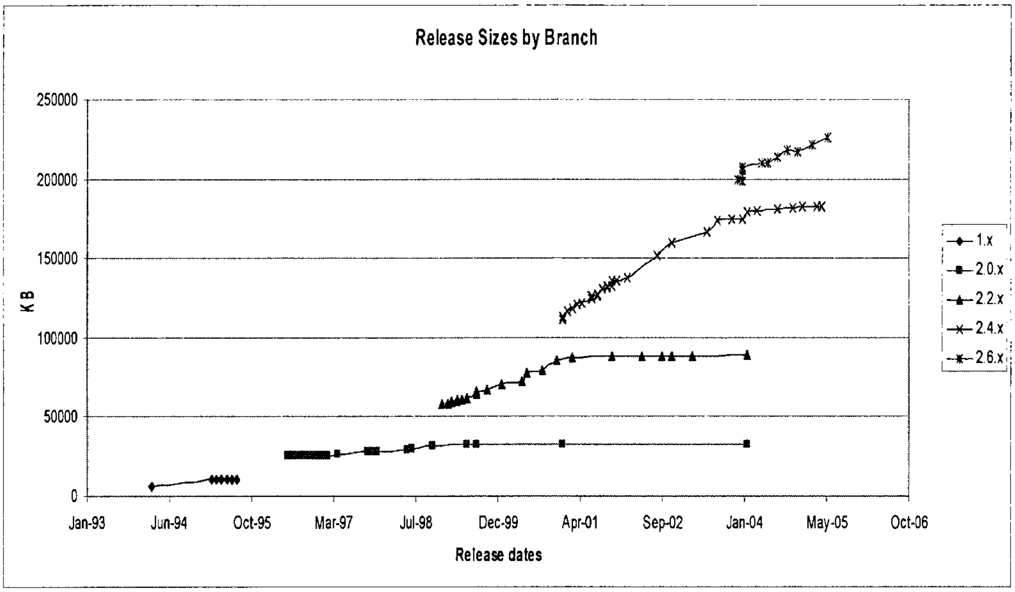


Figure 2.1 Linux Release sizes by development branch

A system may incur higher cost due to reasons other than decay. Higher psychological complexity may indeed be due to truly complex problems and their solutions. Developers regularly make tradeoff decisions between efficiency and complexity when designing software solutions. A classical area in computer science is algorithms, where sometimes a linear time algorithm may be more efficient but harder to maintain. The maintenance [45] associated with it may be too high compared to a slower implementation that is easier to comprehend and has lower psychological complexity.

Symptoms of code decay [31] include bloated code (too many if statements or nested loops, return statements in many sections of a function, etc.), hacks to fix specific defects, ownership by too many developers, lack of comments, and high coupling, among others. Causes of decay include short release cycles of software, imprecise requirements, poor architecture, novice developers, lack of necessary development and process tools (change management), etc. Decay leads to higher maintenance costs [75], [29].

Parnas [71] uses an analogy between software systems and medicine to describe software aging. He uses the term *software geriatrics*, which equates refactoring to major surgery, applies the notion of second opinions, and describes the cost associated with preventative measures. His thesis is that we cannot expect to slow decay as long as complex systems are built by developers without adequate training. We opine that professional software engineers with the experience and necessary skills to build such systems are necessary.

Eick et al. [29] use a number of generic code decay indices (CDIs) to analyze the change history of a telephone switching system. Statistical tests are carried out on the indices, and prediction models are analyzed. Although Eick et. al [29] describe such indices as indicative of decay, they more aptly indicate change and do not demonstrate that change is negative. Examples of Eick's CDIs include: number of deltas, lines added or deleted as part of a change, the number of developers implementing a change, the historical number of changes in a given time interval, the frequency of changes, 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.

In order to further understand design decay, we must find a way to accurately measure and characterize it in software systems. The body of research in software decay, specifically as it relates to object oriented systems is small. We propose a model for studying decay, especially decay in the context of object oriented design patterns.

2.3 Pattern Specification Languages

Design patterns have become pervasive in the designs of complex software systems. Properly communicating designs require techniques that can adequately represent design patterns in a clear and understandable manner. This section provides background on some design pattern languages studied. It is a summary from Izurieta and Biery [50]. In order to clearly understand how grime buildup was affecting realizations of design patterns we needed to select a pattern language that would serve as a formal reference point to evaluate design pattern realizations.

2.3.1 RBML

RBML is the Meta Role Based Modeling Language, which is defined in terms of a specialization of the UML metamodel. While some methods for formal specifications of design patterns require sophisticated mathematical skills, RBML is an extension of UML, already a de-facto standard in the software industry. 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. "The pattern specifications created by the technique are metamodels that characterize UML design models of pattern solutions. A pattern's metamodel is obtained by specializing the UML metamodel" [32]. 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 represent the dynamic aspects of the pattern. For in depth descriptions of RBML see Kim [56], and France et al. [33].

Figure 2.2 from France et al. [32], displays an example of a part of the Observer design pattern that structurally conforms to its SPS in RBML. Each class in the realization maps to an RBML role in the specification. Class *Kiln* maps to the class role *Subject*, while class *TempObs* maps to the class role *Observer*. Similarly, each association in the pattern realization maps to an association in the RBML specification. Association *obsTemp* maps to the association role *Observes*. Essential methods and fields are also checked for compliance. Finally, class and association multiplicities are validated against the specification to check for violations.

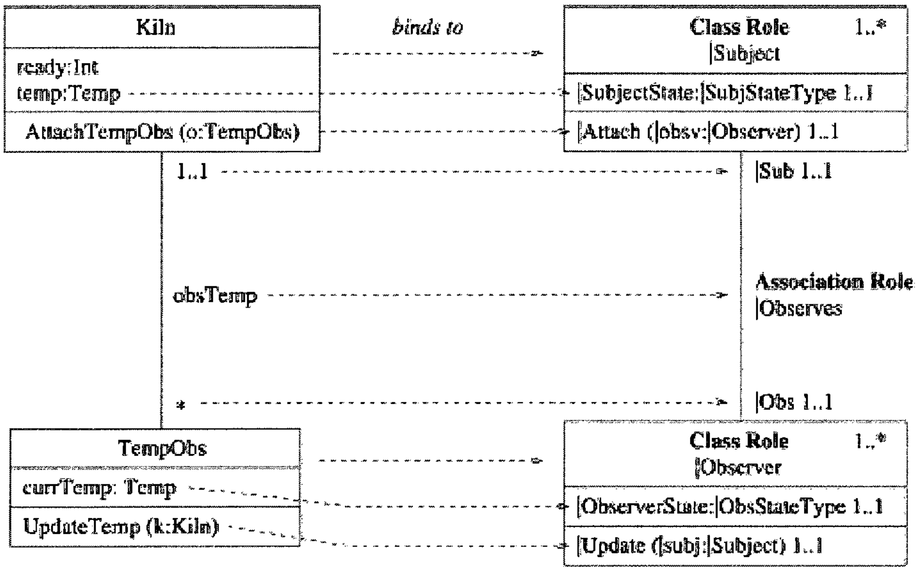


Figure 2.2 A structurally conforming class diagram and its SPS

Establishing structural conformance of a pattern realization to its SPS consists of the following steps, as specified by France et al. [33], and Kim [56]:

a) Bind UML classes to SPS roles.

b) Check compliance with classifier roles realization multiplicities.

c) For every UML class bound to an SPS role:

i) Check that all metamodel constraints are satisfied.

ii) Check that feature roles (structural and behavioral) in the UML class bound to an SPS role satisfy multiplicities.

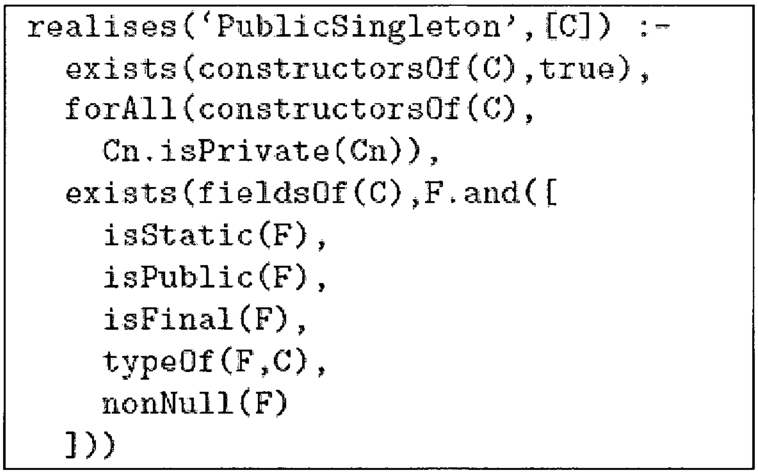
iii) Check that all mandatory features specified in the SPS are present in the UML class diagram.

d) Establish conformance of relationship roles.

By establishing structural conformance to an SPS, we can check that the syntactic properties expressed by the SPS are met. A pattern also has semantic properties that are expressed via *constraint templates* in RBML. Semantic properties are constraints placed on the behavioral aspects of a pattern. *"Tools that mechanically discharge most proof obligations are not likely to appear in the near future."* [32], although formal specification languages such as Z [51] do provide help in this area if the constraints of RBML can be adequately captured. Checking semantic constraints requires that proof obligations are dismissed by asserting that the operations supported by the class conform to behavioral feature roles.

2.3.2 Spine

SPINE [13] is a prolog like language that allows for the specification of design patterns. The underlying design behind SPINE allows patterns to be specified in terms of constraints of their implementation in Java. Design patterns are specified using functions and predicates; which are supported with some existential quantifiers such as *for all*, and *exists*. The language and these constructs allow patterns to be specified in terms of behavior and structure. Figure 2.3 shows a simple definition of the *Singleton* pattern.



```
realises('PublicSingleton',[C]) :-
  exists(constructorsOf(C),true),
  forAll(constructorsOf(C),
    Cn.isPrivate(Cn)),
  exists(fieldsOf(C),F.and([
    isStatic(F),
    isPublic(F),
    isFinal(F),
    typeOf(F,C),
    nonNull(F)
  ]))
```

Figure 2.3 SPINE definition of the Singleton pattern [13]

The definition includes all the items that are essential to define a Singleton pattern. For example, all fields described by the class cannot be modified by anyone, including the methods within the class itself. All constructors in the class are also private, etc. SPINE is coupled with HEDGEHOG [14], a proof engine that processes pattern specifications written in SPINE, and then determines if such a specification can be matched to a selected set of classes in Java. HEDGEHOG builds a proof tree and processes declarative constraints to verify the realization of a design pattern. A number of static predicates are defined by HEDGEHOG to check the essential relationships that exist between classes of design patterns. Additionally, HEDGEHOG provides support for semantic predicates that need to be checked to see if the realization of a pattern conforms to the intended definition. Blewitt et al. [13] note that while some semantic problems are undecidable, most semantics that need to be checked in design pattern do not exhibit such traits and are for the most part manageable.

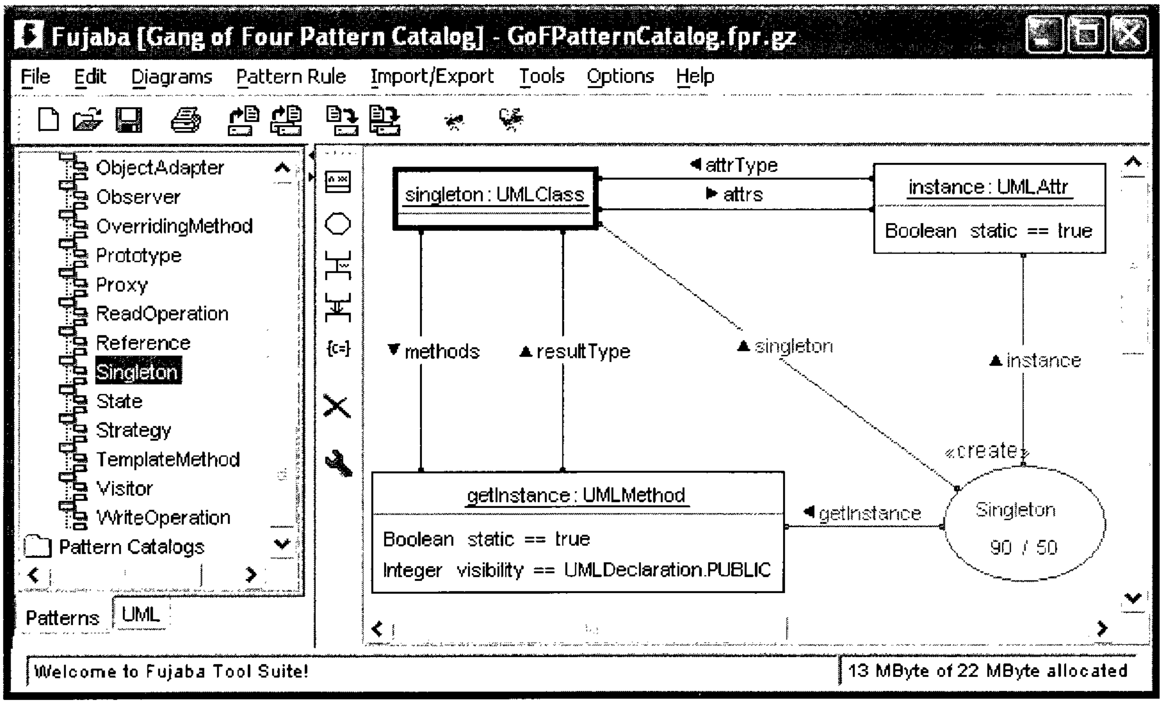
Results of experiments performed by Blewitt et al. [13] show that of the twenty-four patterns specified by Gamma et al. [36], a total of 7 of them cannot be represented with SPINE. These patterns were Builder, Façade, Chain of Responsibility, Command, Interpreter, Mediator, and Memento. Defining these patterns in SPINE produces either a *narrow* definition or a *vague* definition that would match too many false positives in the former or too many false negatives in the latter case.

SPINE and HEDGEHOG together provide a framework to specify, detect and model design patterns allowing the user to specify essential complexities of the patterns while keeping a relatively simple syntax. The lack of a visual aspect to the language may hinder more complex modeling.

2.3.3 FUJABA

The FUJABA (From UML to Java And Back Again) Tool Suite is a Computer Aided Software Engineering (CASE) tool developed by researchers at the Software Engineering Group at the University of Paderborn in Germany. FUJABA provides an extensive set of capabilities for the creation of UML models, story diagrams (described as by the creators as UML behavior diagrams), and graph transformations used for software design [35]. Source code generation from models, reverse engineering of existing source code to diagrams, and pattern detection are provided for FUJABA users.

The pattern detection capabilities included within FUJABA are unique from many comparable solutions. While many other pattern inference tools attempt to pinpoint patterns with great precision, the creators of the pattern specification and inference tools used within FUJABA take a different approach. As evident in the GoF collection of patterns, pattern definitions are intentionally nondescript. The lack of a formal "recipe" for patterns allows developers flexibility in implementation details. While implementations of a pattern will have certain similarities, attempting to locate patterns embedded within source code based on a generic "cookie-cutter" approach may lead to high false positive pattern detection rates and limit the usefulness of a detection tool. In order to provide more meaningful results, the tool associates two "fuzzy values" with the results from the pattern detection. These fuzzy values represent the believed likelihood that the pattern was found and a threshold that assists the user in helping determine whether the likelihood value implies the determination of the pattern should be accepted.



Welcome to Fujaba Tool Suite!

13 MByte of 22 MByte allocated

Figure 2.4 FUJABA Rule for the Singleton pattern [80]

Pattern detection begins with the creation of abstract syntax graphs (ASGs) from the source code in the requested code base that is generated by the Java Compiler Compiler (JavaCC) [68]. Once the graphs have been assembled, a unique "bottom up, top down" approach is used for pattern comparison. The algorithm will begin in the "bottom up" state of the algorithm, analyzing portions of the ASG against a collection of predefined pattern rules that are added to a priority queue. It is believed that this process helps quickly narrow the possible pattern candidates [68]. Pattern rules are defined in

FUJABA as graph transformation rules. These rules are broken up into sub-pattern rules that represent known aspects of a given pattern. An example of a pattern rule is shown above for the Singleton pattern in Figure 2.4. As a rule is successfully applied (annotations are added to the graph) in the "bottom up" state, subsequent rules for a specified pattern will be run. Once a rule is encountered where annotations on the graph necessary to proceed are missing and the current rule cannot be run in the "bottom up" state, a state change is made to "top down" [68]. The "top down" portion of the search attempts to apply rules more specific to the pattern. During the whole process, the fuzzy values associated with the likelihood of the pattern occurrence and the threshold for pattern acceptance is being continually updated. The final fuzzy value for the likelihood is no greater than the lowest sub-pattern likelihood value encountered during the process [69].

FUJABA provides some additional user capabilities. The inference process used is actually semi-automatic [68]. A user performing the reverse engineering may interact with the system during the process to view the intermediate results and manually intervene if necessary. Users may also specify their own pattern rules (beyond the GoF support already packaged) by creating graph transformation rules (graphs using specified notation) in the GUI and importing the newly created pattern rules.

2.3.4 PINOT + MUSCAT

The Pattern INference and recOvery Tool (PINOT) was created by software engineering researchers at the University of California-Davis in order to "reverse engineer" existing source code to detect software patterns. PINOT supports detection of nearly all "Gang of Four" (GoF) patterns and is limited to Java source code.

PINOT adds an embedded pattern detection engine inside the open source Java compiler Jikes. The pattern detection engine leverages many aspects of the compiler, namely the abstract syntax tree and symbol tables in order to help analyze potential patterns. The creators of PINOT partitioned the patterns under analysis into two categories: those classified by structural characteristics ("structure-driven patterns") and those classified by behavioral traits ("behavior-driven patterns"). Examples of structure-driven patterns include Facade, Template Method, and Proxy [36]. Examples of behavior-driven patterns include Singleton, Decorator, and Strategy [36].

The detection algorithm begins by searching for classes that exhibit traits determined to be effective in locating a pattern. This allows the ability to quickly narrow the classes under analysis. Once a candidate group is formed, structural analysis is performed to identify structure-driven patterns and candidates for behavioral analysis. The structural analysis searches inter-class relationships and structures relevant to known patterns, and runs algorithms tailored to match each structure-driven pattern and potential behavior-driven patterns. Further behavioral analysis is applied to candidates identified as possible pattern participants by ensuring that the class(es) under study perform the correct function under specified behavior. This is verified through the use of data-flow analysis on the Abstract Syntax Tree (AST) generated by the compiler to create a control-flow graph between elements identified as "basic blocks".

In a head-to-head comparison, PINOT exhibited advantages in performance and detection accuracy against two other pattern detection applications, FUJABA and HEDGEHOG [14]. While PINOT exhibited advantages in speed and accuracy under given test sets, the application does contain limitations. The first is that the tool cannot provide pattern inference information on incomplete source collections, unlike FUJABA [80]. While this ability may not seem useful from a reverse engineering standpoint, it may prove useful in efforts to refactor existing (but not complete) code to leverage known patterns or in analyzing applications that leverage third-party libraries where the source code is not available. Another limitation of PINOT is that it only detects common implementations of a given pattern. Any variation or deviation in source code from an expected pattern solution may not be properly categorized as the pattern. Perhaps the largest limitation of PINOT is that it is constrained to detection of GoF patterns. Support for custom patterns or other commonly used patterns would require modifications to the PINOT source code.

Realizing that support for GoF patterns only limited the possible scope of the tool, the creators of PINOT created another tool for use with PINOT that allow for additional extensibility. The Minimal UML SpecifiCATion (MUSCAT) Language allows users to define their own patterns through the use of a subset of the UML, thereby providing the ability to detect these patterns through a "customized PINOT." Within MUSCAT, only class diagrams are used because the ability to relate elements of a diagram used to convey structural information (e.g. Class Diagram) back to a diagram used to represent behavior (e.g. Activity Diagram) would require additional annotation [79].

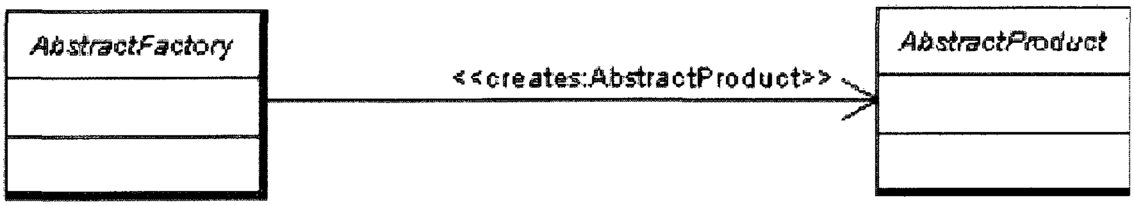


Figure 2.5 MUSCAT Representation of the Abstract Factory Pattern [79]

A subset of notations used for UML class diagrams are leveraged by MUSCAT. Supported notations include: aggregation, inheritance, bidirectional association, dependency association, and directed associations [79]. MUSCAT also specifies a defined set of custom UML stereotypes referred to as "pattern rules." Pattern rules are used to define characteristics necessary to express the behavior associated with a given pattern. Rules are as generic as <<creates:type>> and <<!creates:type> that signify a class creates or does not create a given type, or as specific as <<singleton>> and <<facade>> where a class embodies a Singleton or Facade pattern. An example of the Abstract Factory pattern specified in MUSCAT is shown in Figure 2.5.

After the appropriate diagram(s) have been created using the MUSCAT notation, the pattern definitions specified within the diagram may be added to PINOT in order to provide future detection. In order to do this, the diagrams must be saved as XMI. XMI is a commonly used format for storing UML diagrams in a vendor agnostic format. The XMI representation of the diagram(s) is passed to a python script that creates C++ code defining the elements of the pattern(s) required for detection that conforms to the PINOT API. The generated code is compiled and linked with PINOT to create a customized version of PINOT that will be able to detect the pattern(s).

While MUSCAT does add a degree of extensibility to the PINOT application, the ability to define new patterns is constrained by the need to depict the pattern as a class diagram (exclusively) and the support provided for given pattern rules.

2.3.5 LayOM

LayOM is the layered object model language with explicit support for specifying design patterns. The code specified in LayOM is supported by an environment that translates it into C++. Objects are encapsulated by layers, which represent different levels of abstraction. To communicate with an object, a message must pass through all the layers, which can represent different levels of functionality. Figure 2.6 illustrates the concept.

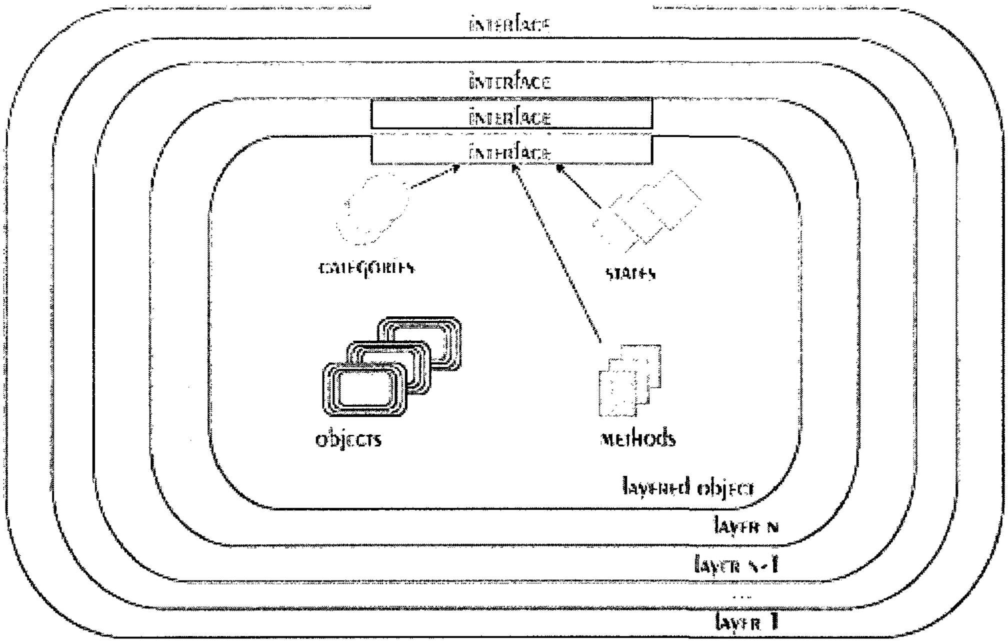


Figure 2.6 A layered object model [16]

At the center lie the concrete objects which have concrete states. States can be abstracted at higher levels to form additional simpler dimensions. Categories are also a characteristic of objects that can be used to restrict the functionality afforded to different clients of the object. The idea is to use layers as higher abstraction layers to represent design patterns, where each layer can define either the structural, behavioral, or application relationships between objects that participate in the design pattern. Bosch [16] has created a number of layers that can be superimposed on top of traditional C++ objects to represent the functionality of various design patterns. The goal of these layers are primarily to add the ability to not just extend patterns, which is the intended method for changing patterns, but also to have an ability to reuse the layers. Design patterns only provide general guidelines to implement a solution to a problem, leaving details to be application specific. In order to reuse a design pattern, the layers created in LayOM must be generic enough, yet provide a sophisticated level of essential complexity inherent in every design pattern. LayOM is a textual language, and its translation into C++ modules, allows the translated units to be included into other executable modules. Additional work is necessary to make sure that the C++ representation of the design pattern fits an application correctly. Figure 2.7 is an example of a simple layer created for the Façade pattern.

```
class FacadeExample
   layers
      face : Facade(forward mess1, mess2 to PartO1, forward mess3 to PartO2);
      PartO1 : PartOf(ClassOfO1);
      PartO2 : PartOf(ClassOfO2);
   ...
end; // class FacadeExample
```

Figure 2.7 LayOM layer for the Façade pattern [16]

The ability of LayOM to model concepts is poor due to its complex textual description. The strict syntax checking of the translator allows for easier detection and specification techniques. Both, behavioral and static aspects of design patterns can be represented with layers.

2.3.6 LePus

The LanguagE for Pattern Uniform Specification (LePUS) is an object-oriented modeling language that was devised by researchers at the University of Essex in the United Kingdom. LePUS is intended to be coupled with Class-Z, a formal language that is derived from Z [51]. The combination of LePUS and Class-Z provides the ability to combine a visual representation specified in LePUS with a formally defined textual description in Class-Z.

The creators of LePUS and Class-Z characterize it as "a subset of first-order predicate calculus" [26]. As a result, models created within LePUS/Class-Z are limited to depicting a particular state. No support is provided for higher-order logic or component interaction diagrams (e.g. Sequence Diagrams or Collaboration Diagrams) included within the UML. The language is built upon the use of terms and formulas. A term is an entity associated with a given type and dimension [84]. Native types are provided within the language for CLASS (classes), SIGNATURE (of a method), and HIERARCHY (a subtype of a CLASS collection). Another type is provided for METHOD, described as the "superimposition of a signature term over a class term" [84]. A dimension may be zero or one, where zero dimension terms are for single entities and one dimension terms are used for collections. Additional support is provided for unary (property), binary (e.g. aggregation, creation), and transitive relations, as well as a custom set of predicates. Each visual construct used within LePUS has a corresponding textual equivalent in Class-Z.

As LePUS was created, care was given for design pattern specification support. The use of terms, formulas, and the native type and predicate constructs are everything needed to specify all of the GoF patterns. A library of the GoF design patterns represented in LePUS and Class-Z is available online [27]. An example of the Abstract Factory Pattern in LePUS and Class-Z is shown below in Figures 2.8 and 2.9.

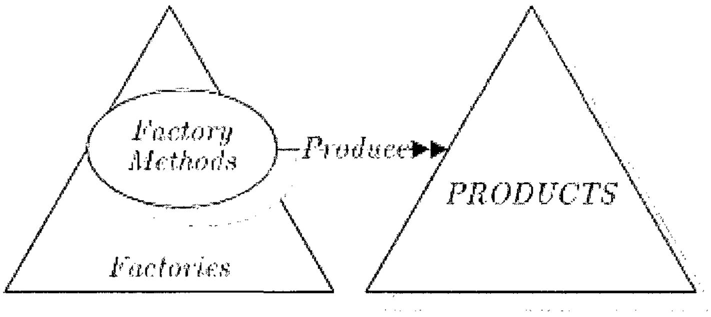
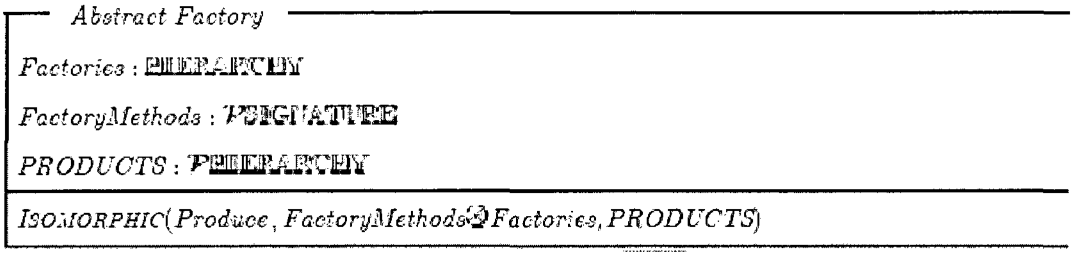


Figure 2.8 The Abstract Factory Pattern in LePUS Notation [28]


Abstract Factory

Factories : HIERARCHY

FactoryMethods : ℙSIGNATURE

PRODUCTS : ℙHIERARCHY

ISOMORPHIC(*Produce*, *FactoryMethods*⊗*Factories*, *PRODUCTS*)

Figure 2.9 The Abstract Factory Pattern in accompanying Class-Z Notation [28]

One of the weaknesses of the LePUS approach is that there is a steep learning curve to use it. While starting from scratch allowed the language designers extensibility to build a language around the requirements of their domain (object-oriented design models and pattern specifications), other pattern languages are based on the widely-adopted Unified Modeling Language (UML). Experienced object-oriented software designers would likely be familiar with the notation used within a UML-based language. Adoption of LePUS and Class-Z would require knowledge in both the unique modeling notation used and formal methods.

Unlike all of the other pattern languages described in this paper, it is important to note that pattern detection capabilities are not provided by LePUS, Class-Z, or tools tailored for LePUS and Class-Z modeling. The use of LePUS and Class-Z is intended for forward engineering purposes only.

2.3.7 DeMIMA

In this system, Gueheneuc and Antoniol [41] use a multilayered approach to identify possible design motifs. They identify micro-architectures in source code that are similar to design motifs. Micro-architectures refer to the structural characteristics of possible design patterns, and are composed of classes and their relationships. The term design motif is used instead of design pattern because patterns also encompass information not readily available from their identification. For example, intended use, motivation, and consequences are traits that cannot be observed in design patterns. Micro-architectures can match up with more than one motif.

The motif identification process is divided into three tasks. The first task is automated and consists of matching up a candidate micro-architecture found in source code to a design motif that is represented with a UML-like class diagram. The second and third tasks consist of contextualizing and comprehending a micro-architecture in its setting of use and are dependent on a maintainer's level of experience. DeMIMA assists with the first task. It provides a multi-layered approach to match micro-architectures with design motifs by characterizing the constituents of a design motif using a meta-model. The first layer uses the *Pattern and Abstract-level Description Language* (PADL) to describe relationships and build a UML-like model of the code. The language provides the ability to distinguish between association, use, aggregation, and composition relationships between classes. The original work to recover these relationships [42] provides the ability to recognize unidirectional binary relationships, which is a necessary property required by the recognition algorithms in the second layer of DeMIMA. In the third layer, DeMIMA looks for micro-architectures in the second layer that may match a design motif. To identify micro-architectures, the model is transformed into a constraints systems and explanation based programming is used to find the best match.

Gueheneuc and Antoniol apply DeMIMA to five open source systems and thirty three industrial components. On average they observed thirty-four percent precision with 100 percent recall for the open source systems. Precision (# true design motifs found / # micro-architectures found) can be improved by adding more constraints that reduce the possible micro-architecture matches. Additionally, some design motifs, such as the Factory pattern have low precision because many micro-architectures found in the source code can match its structure. The identification algorithms would require additional semantic knowledge to distinguish between false positives and actual patterns.

2.3.8 Pattern Language Comparison

The creation and use of pattern languages is an emerging field. While the Gang-of-Four (GoF) patterns have fueled wide adoption and interest in design patterns, there are still challenges in providing a complete pattern specification language that provides comprehensive support for modeling, specification, and pattern detection. Support for these functions varies between each of the languages evaluated within this research due to the scope, purpose, and original intent of their creators.

While none of these languages are ideal for every desired pattern language purpose, the languages evaluated in this paper are among the best available options today. When choosing between possible options, users must evaluate language capabilities against their needs. Each offers adequate specification capabilities. Principal differences between the languages are in style, modeling, and detection. The most effective languages for modeling were RBML and FUJABA. RBML offered the best alternative for specifying the *blue print* that is used to compare the realizations of design patterns in the code to make sure no violations occur.

2.4 Adaptability and Testability of Software Designs

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, etc. Also, establishing a relationship does not necessarily imply causality, making this a difficult problem.

Adaptability has many definitions, but in general we define adaptability of a design pattern as follows:

- the effort required by a developer to make changes to a pattern,
- the ability of the pattern to accommodate changes in its environment (changeability), and
- The ease with which a design pattern can be pulled out from its current environment and reused in a different setting.

The ISO standard for software engineering [46] based on studies by Lientz and Swanson [62] proposes different categories for software maintenance. In particular, *adaptive maintenance* describes *"the modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment."* Clearly if the evolution of design patterns exhibit evidence of decay, rot, or grime accumulation as defined in this research, then all aspects of adaptability suffer. This includes not only usability, but further code modifications.

The environment of a pattern realization is its setting of use. If a pattern realization evolves in the intended way, then it can be extended as defined when changes in the environment occur, thus it adapts to the environment. Typical changes in the environment involve the addition of new concrete classes. By definition, the pattern has the ability to accommodate such changes and the effort required is minimal. *"Designs are more extensible when they are independent of implementation details, allowing them to adapt to new implementations without internal modification or breaking their existing contracts."* [53]

Many software metrics have been studied in procedural paradigms. Rombach [74] shows how some of these measures can predict maintainability (an external quality attribute) in a system. Metrics for object oriented systems first appeared in the early 1990s. Chidamer and Kemerer [22] 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 [61] validate a relationship between Chidamer and Kemerer's metrics and maintenance effort as measured by the number of changes made to classes.

Significant empirical evidence links the negative effects of coupling on adaptability and maintenance of systems. Arisholm and Sjoberg [2] focus on measuring *changeability decay*; the increased effort required to implement changes in object-oriented systems. Subsequent work by Arisholm [3] shows that there are important relationships between structural attributes of object-oriented systems and development effort. Arisholm's primary goal is to assess the extent to which structural properties of object-oriented systems affect changeability. He investigates static structural attributes as well as *change profile* measures. The latter depend on the former, but are weighed by the proportion of change experienced by the corresponding structural measure. Arisholm states 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."

Most studies do not have accurate access to changeability data. Arisholm's study was set up from the beginning to have developers log every change in the system under study. He collected the number of hours spent in analysis, design, coding, testing, and documentation. He used multiple linear regression on change effort and determined that change profile measures were the most significant independent variables when predicting changeability. The structural measures focus on coupling and size. Other measures related to inheritance and cohesion were not used. Additional work by Arisholm et al. [4], finds evidence linking export (efferent) coupling to change proneness, the dependent variable measured as SLOC (source lines of code). Import (afferent) coupling does not play a significant role (beyond size measures) when explaining changes is SLOC.

Basili et al. [5] use the probability of fault detection as a surrogate measure for fault proneness. The goal of their observational study was to assess Chidamer 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. 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. [17] show that coupling measures are useful indicators of quality in object oriented designs. Their study also uses Chidamer 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 [20] 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. [18] 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.

Testability of object oriented designs has been extensively studied. Binder [12] suggests that at its most abstract, tests should demonstrate the relationships that must hold for a system under test. The design of object oriented systems is driven in large part by the relationships of the objects and classes that make up the system. Evaluating a full design can be daunting. However, by focusing on the design patterns that make up the system we can gain a better understanding at a localized level.

Fenton and Pfleeger [30] propose using the *Test Effectiveness Ratio* (a measure of code coverage) as a measure for knowing the "extent to which the test cases satisfy a particular testing strategy." The measure is defined as the number of paths exercised at least once divided by the total number of paths. The denominator of the ratio is a finite subset of all possible paths (an infinite number); such as linearly independent paths, visit-each-loop once paths, simple paths, branch coverage, or statement coverage. They also suggest the *minimum number of test cases* needed to satisfy a strategy for a given specification. They equate a test case with a finite path through a flow graph. The number of test cases helps identify the amount of time needed to test, and also with planning strategies. Both of these measures are presented in the context of procedural designs.

Work by Hamlet and Voas [45] and Voas and Miller [85] propose methods for improving reliability of systems by improving the testability of systems. The crux of their idea is that highly testable systems uncover problems and failures early, thus producing a highly reliable system.

Tsai et al. [82] categorizes design patterns into two distinct groups, static and dynamic. Static patterns are typically used where changes to the design are not anticipated. The Singleton pattern is an example of a static pattern. Dynamic patterns allow for extensibility either at runtime or compile time, and new functionality is achieved via polymorphic constructs. Examples of dynamic patterns include the Visitor pattern and the State pattern. Both static and dynamic realizations of design patterns are studied.

To evaluate the consequences that decay and grime buildup has on testability, we select evaluation criteria described in section 4.4.1. We build on the equations proposed by Binder [12] and look for the formation of anti-patterns in code.

2.5 Summary

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 [3], [17], [5] are available that investigate the relationship that may exist between independent object oriented measures and the negative effects they may have on design quality.

This dissertation focuses not just on quantifying object oriented decay, but takes us a step further towards 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.

3. DECAY, ROT AND GRIME DEFINITIONS

To measure decay, rot and grime we must first characterize them. Preliminary research [47] revealed that prior definitions of decay [71], [29] do not quite describe what was observed in design pattern realizations. Data mining and manual inspections of code showed that realizations of patterns tend to remain in place as the system evolves. The structure of a design pattern represented by its UML class diagram does not tend to break. Rather, the phenomenon observed revealed that the realization of patterns were becoming obscured by additions of software artifacts that were not part of the intended ways in which patterns were meant to be extended. The following definitions are a result of these observations. In sections 3.1, 3.2 and 3.3 we define the notions of decay, rot and grime as they pertain to design patterns. In section 3.4 we provide an example and show the process for verifying a pattern, and in section 3.5 we provide detailed descriptions of the measures used in our observational case study.

3.1 Decay

We define *decay* as the deterioration of the internal structure of system designs. 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, number of methods, etc. In general, accurately measuring decay is very difficult to do, because predefined quality measures vary for individual designs. Decay is most apparent when the time required to make changes in a software system increases, regardless of the available resources. In other words, decay lowers the adaptability of the system leading to further atrophy and aging. 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.

3.2 Design Pattern Rot

By focusing on micro-architectures of designs we can examine well-formed structures against design quality violations much more accurately. Thus, we define *Design pattern rot* as the deterioration 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. The structural integrity of a design pattern realization is determined by systematically checking its classifiers (classes, interfaces, etc.) and associations against its formal RBML specification. A single *core deviation* from the formal specification stops the realization from representing said pattern. For example, the absence of an intended 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 such association as a result of a lack of understanding of design patterns, or because the pattern may not have been documented.

We have observed near-instances of patterns. A *near-instance* is a close match to a pattern RBML specification that violates some requirements. For example, figure 3.1

displays a near-instance of the Observer pattern, where all elements are bound to an RBML classifier with the exception of the observer *"ProgressIndicator"*. This class cannot be bound to any concrete or abstract observer. The figure represents a distorted view, or near instance of a design pattern. A change that turns a pattern into a near-instance represents a case of pattern rot.

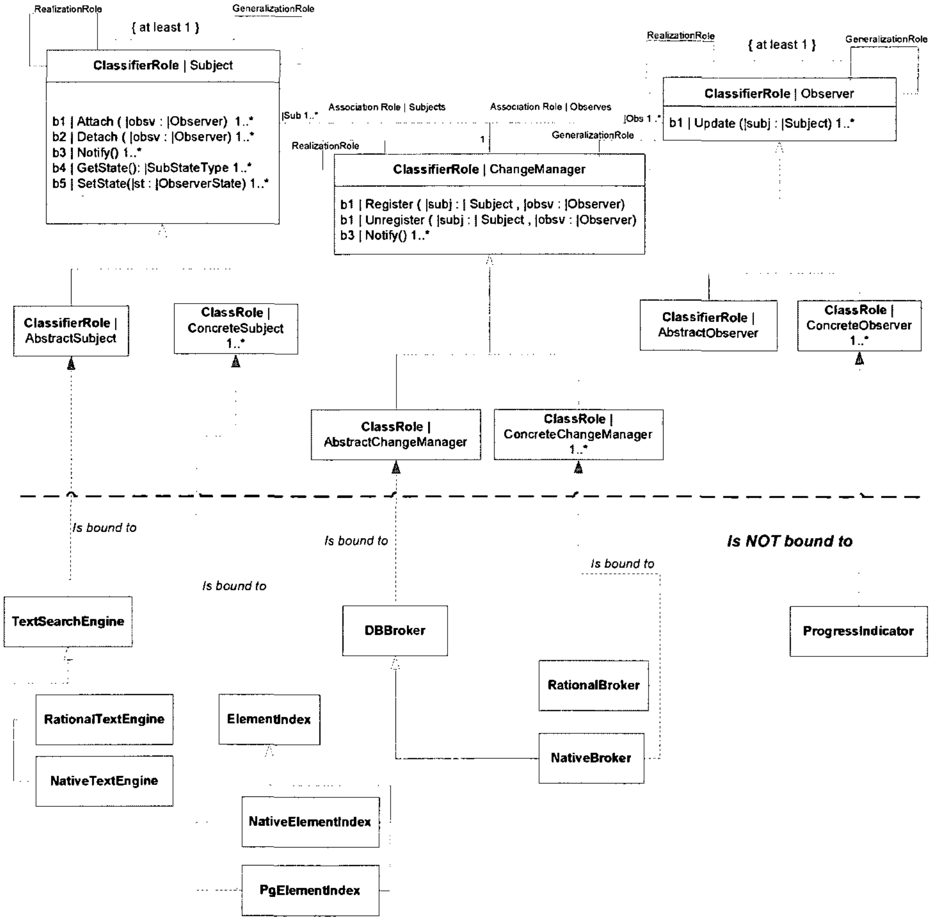

RealizationRole
GeneralizationRole
{ at least 1 }
ClassifierRole | Subject
b1 | Attach (|obsv : |Observer) 1..*
b2 | Detach (|obsv : |Observer) 1..*
b3 | Notify() 1..*
b4 | GetState(): |SubStateType 1..*
b5 | SetState(|st : |ObserverState) 1..*
|Sub 1..*
Association Role | Subjects
Association Role | Observes
|Obs 1 .*
1
RealizationRole
GeneralizationRole
ClassifierRole | ChangeManager
b1 | Register (|subj : | Subject , |obsv : |Observer)
b1 | Unregister (|subj : | Subject , |obsv : |Observer)
b3 | Notify() 1..*
RealizationRole
{ at least 1 }
GeneralizationRole
ClassifierRole | Observer
b1 | Update (|subj : |Subject) 1..*
ClassifierRole | AbstractSubject
ClassRole | ConcreteSubject 1..*
ClassifierRole | AbstractObserver
ClassRole | ConcreteObserver 1..*
ClassRole | AbstractChangeManager
ClassRole | ConcreteChangeManager 1..*
Is bound to
Is bound to
Is bound to
Is bound to
Is NOT bound to
TextSearchEngine
RationalTextEngine
NativeTextEngine
ElementIndex
NativeElementIndex
PgElementIndex
DBBroker
RationalBroker
NativeBroker
ProgressIndicator

Figure 3.1 A near-instance of the Observer pattern

3.3 Design Pattern Grime

If design patterns do not show signs of pattern rot (a form of decay) as defined, then we want to understand how evolution is having an impact on them. *Design pattern* *grime* is a form of decay that does not break the structural integrity of a pattern; instead, it is the buildup of unrelated artifacts in classes that play roles in a design pattern realization. Unrelated artifacts do not contribute to the intended role of a design pattern.

We have identified different forms of grime. When a pattern realization is instantiated for the first time, all its classes will contain the necessary elements to make it conformant to its SPS. Elements include the necessary attributes and methods. If increases in such elements are detected as the system evolves while the pattern maintains its structural integrity, then these elements must be unnecessary from a design pattern point of view. *Class grime* is associated with the classes that play a role in the design pattern and grime is indicated by increases in the number of methods of the class and the number of public attributes.

Grime can also be observed in the environment surrounding the realization of a pattern in the form of associations that are unrelated to how a pattern is meant to be extended. Figure 3.2 provides the visual intuition of this type of grime buildup. The pattern instance on the right hand side is shown with associations that develop over a period of time to other classes in the surrounding environment. The number of these couplings obscures the realization of the pattern. Grime can also be observed within the classes that participate in the realization of a pattern. Non-conformant associations can develop between classes. *Modular grime* is indicated by increases in the coupling of the pattern as a whole by tracking the number of relationships (generalizations, associations, dependencies) pattern classes have with external classes.

When patterns become part of a design they are implemented as part of a Java package and are typically distributed as a number of Java 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. *Organizational grime* refers to the distribution and organization of the files and namespaces that make up a pattern.

Grime is relative to the role that a design pattern plays. What is considered grime from a design pattern point of view may be adequate functionality from a different design perspective. Grime buildup occurs as long as the changes that a pattern undergoes do not violate the constraints necessary for the pattern to remain compliant with its formal model definition. The pattern is said to maintain its structural integrity.

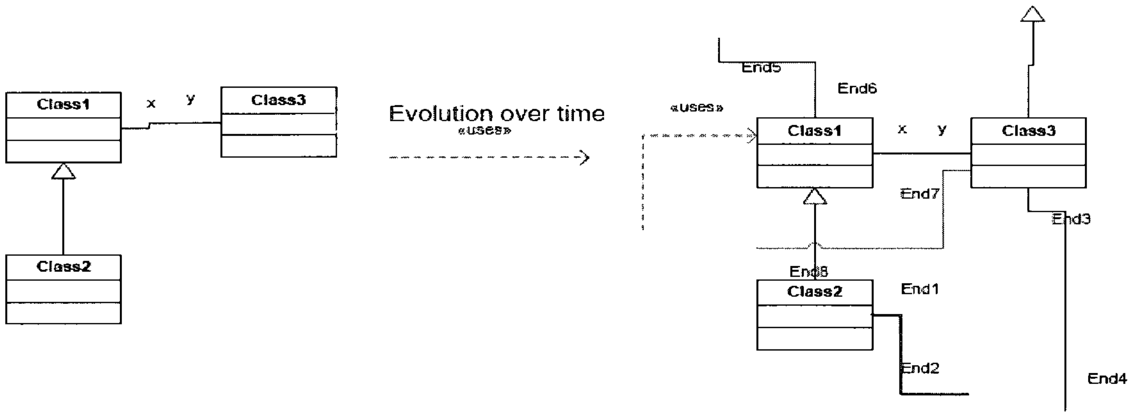


Figure 3.2 A visual intuition of grime buildup. Relationships represent the grime that has accumulated as the pattern evolves

3.3.1 Types of Grime

To quantify decay and grime, we first characterize it. Results from a pilot project [46] identified distinct types of pattern rot and grime buildup in object oriented design patterns. Figure 3.3 depicts the landscape as observed from early results.

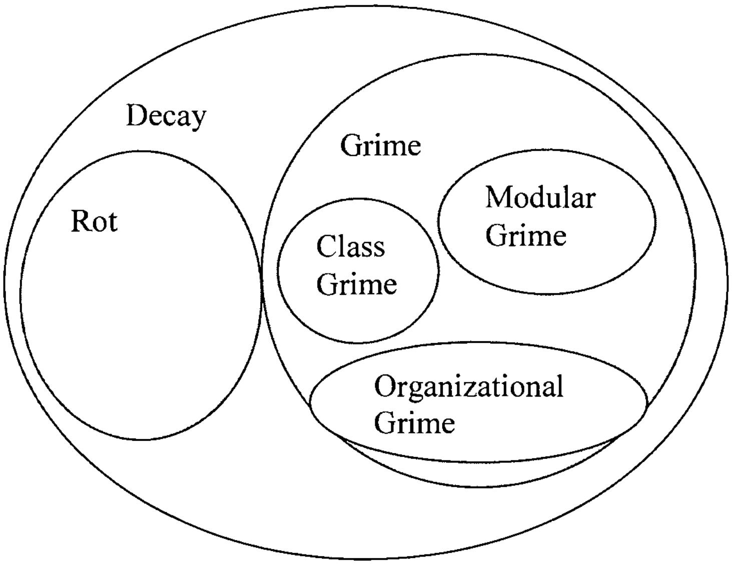


Figure 3.3 The landscape of design pattern rot and grime. Pattern rot and grime are mutually exclusive

Any change with negative effects on testability or adaptability of a design represents decay. Design pattern rot destroys essential elements of the design pattern realization and indicates deterioration of the structural integrity of a design pattern. A single change to a pattern realization can break its conformance with its SPS. In other words, the realization ceases to be a pattern. Although preliminary results indicate that design pattern rot is rare, we have evidence to suggest that buildup of design pattern grime occurs frequently. These forms of grime are depicted by the white bubbles included inside the grime bubble. Each type of grime obscures the realization of the pattern. 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. The following sections provide a definition of each form of grime buildup. In section 3.4 of this document we describe the measurements used to track each form of grime.

3.3.1.1 Modular Grime

Modular grime buildup of a pattern is manifested through its number of dependencies with classes that do not play a role in the design pattern. An increase in coupling with classes (possibly from other patterns), indicates deterioration of modularity and is a symptom of grime buildup. This form of grime also points to the deterioration of the surrounding environment of the design pattern. As dependencies increase, it is reasonable to expect that the system becomes harder to extend and the testability and adaptability of the pattern becomes restricted. Additionally, a higher number of dependencies decrease the comprehensibility of the pattern. Even though the design pattern realization remains as the system evolves, it becomes obscured. The effort required to extract the realization from 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 SPS 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. A growing number of relationships forces classes to become tightly coupled, thus reducing maintainability. For example, Izurieta and Bieman [47] find instances of classes that belong to the Visitor pattern, that later develop inheritance associations from external interface classes. While this is not illegal, it breaks the intent in which the pattern was meant to be extended.

3.3.1.2 Class Grime

Class grime occurs in object oriented systems regardless of whether classes belong to a design pattern or not. We can use well known measures such as depth of a class in the inheritance hierarchy, number of ancestors, number of public attributes, etc. If the inheritance depth of a structure becomes large, then dynamic binding faults can become hard to find. Public attributes that are directly accessible can cause encapsulation problems. Graves et al. [39] found that the number of times code has been changed is a better predictor of the number of faults than the code size. They develop a measure, "number of deltas", which tracks the number of changes of a software module over its entire change history. 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 rather than rot. For example, once a realization of a pattern is verified, then we know it encapsulates the necessary class attributes, however an increase in class attributes is an indication of class grime. We expect that class measurements should remain stable for classes that belong to design patterns and this would be in line with the rationale of how design patterns are meant to evolve, in that they should only be expanded through concrete classes and not through modification to existing classes. However, studies by Bieman et al. [11] show that classes that participate in patterns are just as prone to changes as those that do not participate in patterns, thus providing strong evidence for this form of grime buildup.

3.3.1.3 Organizational Grime

Organizational grime buildup refers to the namespace (such as Java packages) and physical (files and directories) organization of design patterns. The distribution and organization of the packages that make up a pattern tend to change over time and can clearly cause maintenance efforts to increase. The number of physical files that contain the actual implementation of patterns will grow if additional concrete classes are added. The case studies in this research are all Java open source systems, and in Java, all public classes must be in their own files. The addition of a few or no new files (and classes) to an evolving pattern realization along with measurable increases in attributes, LOC, or methods indicates that existing classes are being modified rather than extended via new concrete classes. Grime buildup can occur when the pattern realizations are not being extended via concrete classes.

We observed organizational grime buildup of some patterns in a pilot study [47]. The study tracked 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. Grime occurs when a design pattern is spread across packages, or when the relationships between packages increase. A consequence is increased maintenance effort. 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. We used JDepend [53], design pattern finder [24], and custom scripts (available in appendix B) to gather metrics.

3.4 Structural Verification of Pattern Realizations

Informal pattern definitions, such as those in Gamma et al. [36], are not sufficient for detecting decay. A precise specification is necessary in order to compare realizations found in designs against it. More importantly, the specification must be usable in a practical sense. Based on our investigation of pattern languages, we selected the Meta Role Based Modeling Language (RBML) [33], which is defined in terms of a specialization of the UML metamodel.

To understand how realizations of design patterns decay or accumulate grime over time, we compare the UML diagram of a design pattern realization against its RBML specification as described in section 2.3.1. A threat to the validity of this study is the manual process used to verify that pattern realizations are conformant. Automating these tasks is beyond the scope of this work.

The following example illustrates the deterioration of the environment surrounding a pattern (*grime buildup*) over a period of time, while the structure of the pattern itself remains intact and shows no signs of rot. Figure 3.4 from [56] displays the RBML Structural Pattern Specification (SPS) of the Visitor pattern. We demonstrate that the Visitor realizations found in JRefactory [54] are in structural conformance with their SPS from their initial release studied (V.2.6.12) to the current release (V.2 9.19). Structural changes observed in the last two releases do not affect conformance with the pattern's SPS and are thus considered grime buildup from the perspective of the design pattern.

Observed changes to pattern realizations that do not affect the structural integrity of a pattern can also be benign, and may be the result of necessary changes to the existing design, or the addition of new functionality. However; from the perspective of the intended design we classify those changes as grime if they are not in accordance with the extensibility principles of the design pattern. Whilst necessary, if they do not conform to intended guidelines they obscure the realization of the design pattern.

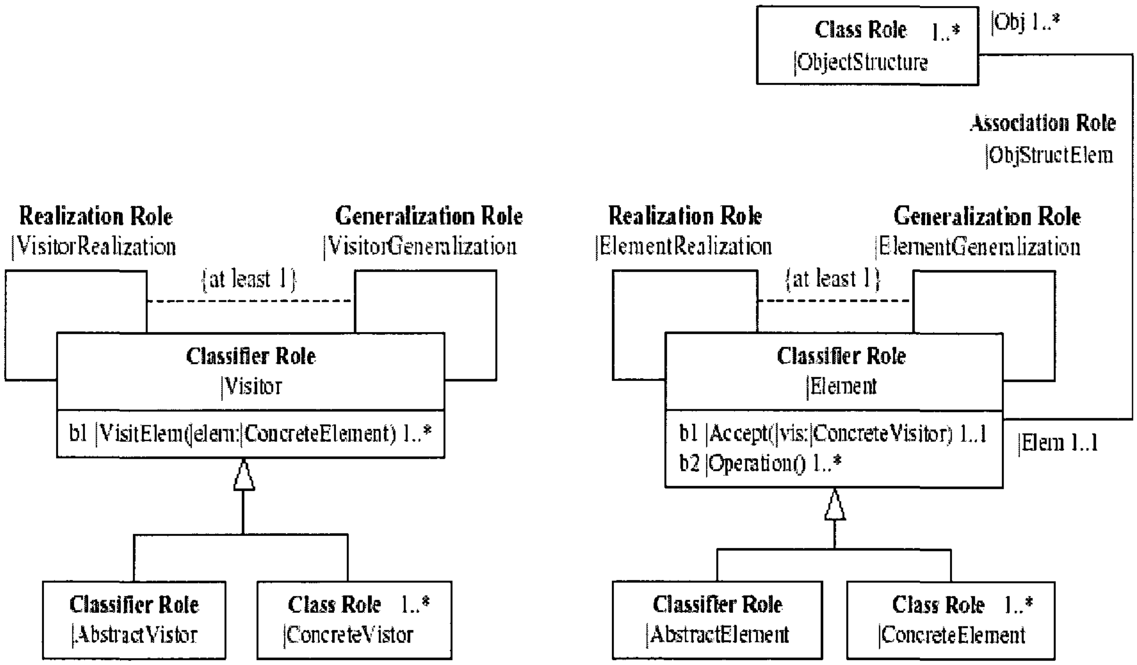


Figure 3.4 A Visitor SPS [56]

We focus on the *visitor hierarchy* of the Visitor pattern. The subject hierarchy (also known as element hierarchy) is similar in that there are no observed evolutionary changes that affect the structural integrity of the design pattern. We find that all versions conform structurally to the Visitor SPS and show no sign of core deviations from the SPS. Figure 3.5 displays the structurally conforming version 2.6.12. For the sake of brevity we have omitted feature roles (operations or methods); however we have found that all classes of the pattern instance conform.

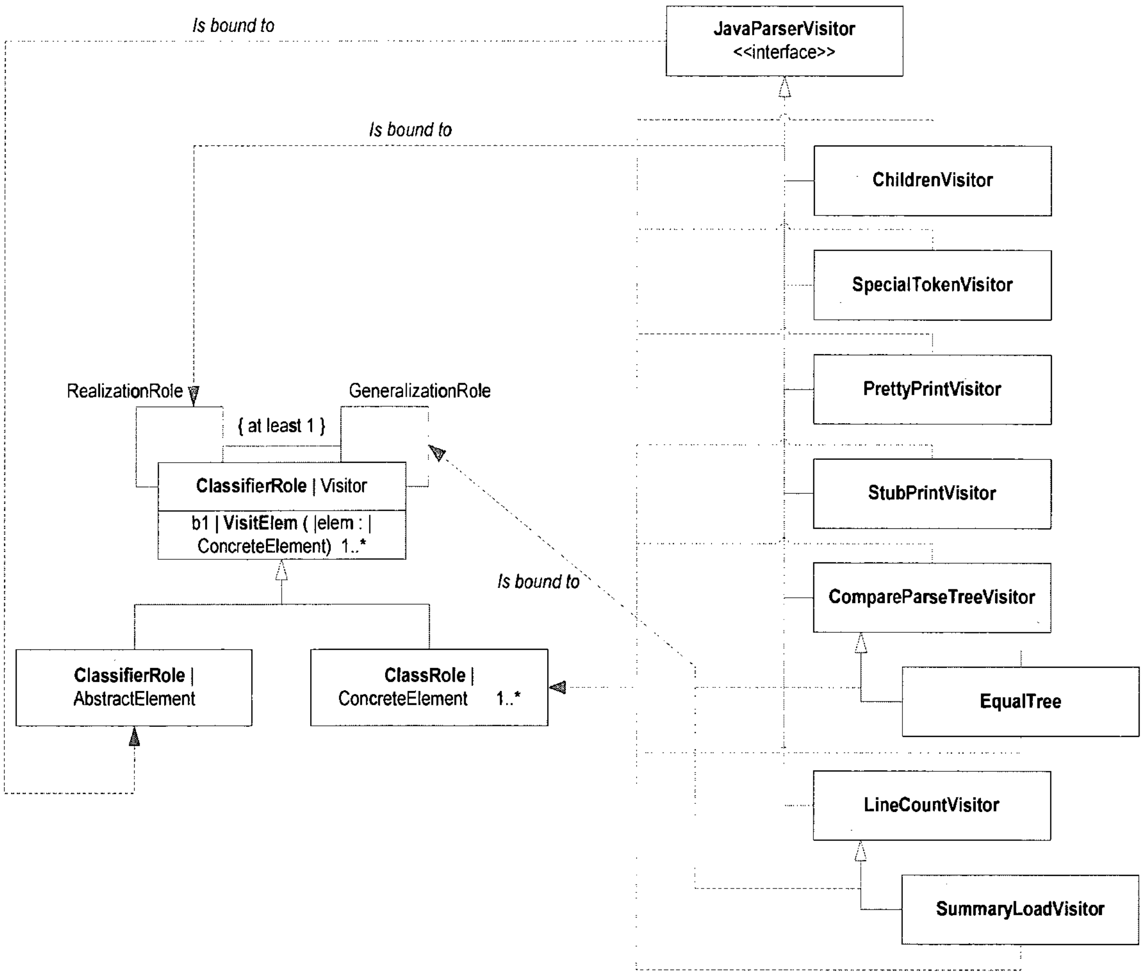


Figure 3.5 A structurally conforming visitor class diagram of a realization of the Visitor pattern in JRefactory version 2.6.12

Pattern realizations in versions 2.6.12 through 2.8.00 exhibit full conformance, and we find no deviations from the SPS of the pattern. In versions 2.9.00 and 2.9.19, shown in figure 3.6, we observe that the Visitor realization has evolved to include additional classes and additional relationships; however they do not represent violations of SPS.

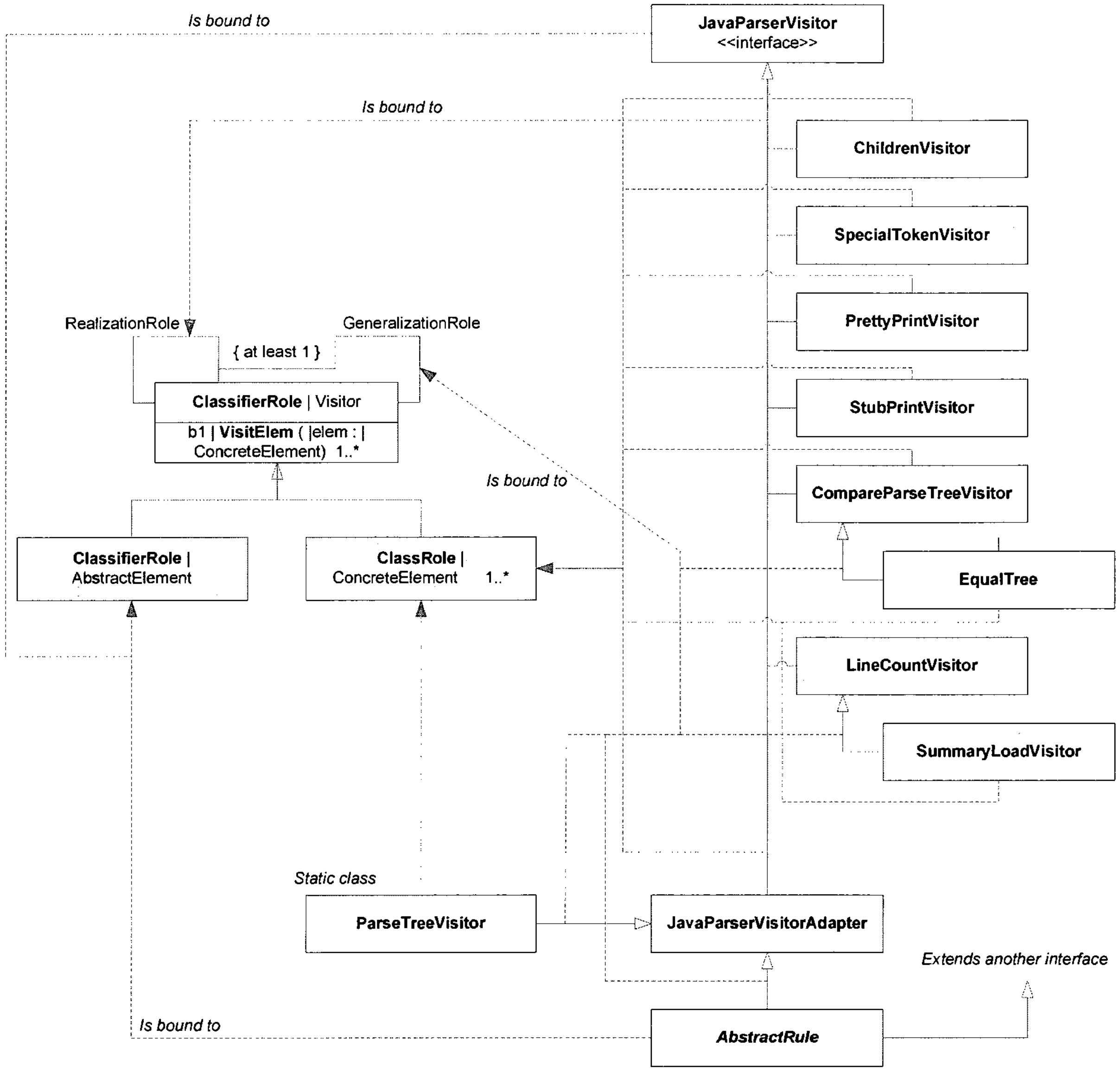


Figure 3.6 A structurally conforming visitor class diagram of a realization of the Visitor pattern in JRefactory version 2.9.19

While the realization of the Visitor pattern does conform to the SPS, the meta-model constraints may not be satisfied if the SPS is strict. A strict SPS specifies many constraints that must be satisfied by a given realization, some of which may not be necessary. For example, an examiner may chose to reject a visitor pattern realization if the concrete classes of the subject hierarchy name the standard *"accept"* methods with another name such as “receive”. If the SPS is too lenient, then any structure that resembles a visitor realization would match. Thus, it is up to the examiner of a pattern to decide what constitutes an acceptable strictness level for matching a design pattern. If the SPS is strict, then we find two *possible* grime buildup examples in the Visitor SPS. The first case of grime buildup could occur with the *ParseTreeVisitor* class. While the class is indeed concrete, and thus bound to the ***ClassRole*** | *ConcreteElement* role in the SPS, the class is also a *static class*. The second case occurs with the UML class *AbstractRule*. In this case, the class also inherits from an external interface, and we thus question the ability to bind this class to the ***ClassifierRole*** | *AbstractElement* role in the SPS.

As said, when checking conformity to an SPS, the results are dependent on the strictness of the RBML model that characterizes the design pattern. A level of strictness must be present in the RBML, for otherwise all UML diagrams would conform. As we define the SPS of various patterns in our research, we strive to include the essential artifacts that are necessary to model a design pattern. In our example, both deviations were introduced in the second to last version of the software studied. If we use a lenient SPS, then the generalizations of the *AbstractRule* class in the UML model are not considered core violations, and can be accounted for as follows. The generalization of the *JavaParserVisitorAdapter* class is structurally conformant with the SPS, and the second realization to an external interface is considered *grime buildup* because it is not part of the SPS that we check the realization against.

3.5 Measurements

This section describes the measurements used to evaluate the hypotheses and the effects that grime buildup has on testability and adaptability of systems.

3.5.1 Modular Grime measurements

Modular grime buildup of a pattern is manifested through increases in the number of dependencies in a pattern. An increase in coupling with classes (possibly with other patterns), indicates deterioration of modularity and is a symptom of grime buildup. This form of grime also indicates the deterioration of the surrounding environment of the design pattern. As dependencies increase, the system becomes harder to extend and the testability of the pattern is restricted. Additionally, a higher number of dependencies can potentially decrease the comprehensibility and thus adaptability of the pattern. Even though the design pattern realization remains as the system evolves, it becomes obscured. An obscured realization is much harder to maintain because developers need to uncover the pattern in order to fully understand the consequences of possible changes to members of the pattern.

3.5.1.1 Relationship counts

Various relationships such as associations, use-dependencies, realizations, and generalizations increase as a result of modular grime buildup. Relationship count increases contribute to obscuring the pattern realization as it evolves, because the pattern structure is not as clear in a class diagram. Some relationships that occur as a result of modular grime are harder to modify than others, thus contributing to an increased effort necessary for a developer to adapt or test a pattern. This accidental complexity that is now embedded in the pattern realization makes it potentially harder for the pattern to accommodate valid extensions. We use Design Pattern Finder [24] to get an aggregate relationship count for every pattern under observation.

3.5.1.2 Afferent coupling of a design pattern (fan in)

We use *afferent coupling* [64] of a pattern realization C_a, as an indicator of modular grime. Afferent coupling represents a pattern's incoming dependencies. A consequence of higher C_a levels may mean that the pattern realization is less adaptable because there is a greater risk of affecting dependent modules when changes occur. Cain and McCrindle [20] refer to this measure as the "responsibility" of a class. Afferent coupling can also be described as the fan-in of a pattern. Afferent coupling is measured by counting a pattern's incoming edges from unidirectional associations to its participants. It is a subset of the relationship counts described previously. As modular grime buildup occurs, the value of the C_a of a given design pattern is likely to increase, potentially reducing its adaptability.

3.5.1.3 Efferent coupling of a design pattern (fan out)

We also use *efferent coupling* [64] of a pattern realization C_e, as another indicator of modular grime. Efferent coupling represents a pattern's dependencies on external classes that may or may not belong to other patterns. Cain and McCrindle [20] refer to this measure as the "instability" of a class. Outgoing dependencies can include inheritance, interface implementation, parameter types, and other temporal dependencies such as variable types if they are part of a method's local variables. Efferent coupling is a subset of the relationship counts described previously. An increasing count of outgoing relationships indicates that the pattern is growing its dependencies, thus potentially making its testability much harder because of the added dependencies.

3.5.2 Class Grime measurements

We use well known measures such as the number of public methods and number of attributes that participate in realizations of design patterns. Any increases in such measures beyond the essential as defined by the design pattern's RBML specification are considered grime from a pattern realization's perspective.

3.5.2.1 Number of Public Methods

The number of public methods refers to 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, but it may be considered grime from the perspective of the pattern if it is not essential to its functionality as a pattern.

3.5.2.2 Number of Attributes

The number of attributes refers to 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 considered 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 we measure the number of attributes. An increase in the number of attributes is an indication that functionality is being added. Some functionality may indeed be needed, but it may be considered grime from the perspective of the pattern if it is not essential to its functionality as a pattern.

3.5.3 Organizational Grime measurements

Organizational grime buildup refers to the namespace (such as Java packages) and physical (files and directories) organization of design patterns. The distribution and organization of the packages that make up a pattern tend to change over time and can clearly cause maintenance efforts to increase.

3.5.3.1 Afferent Coupling of a package

We use *afferent coupling* of a pattern realization's package; C_a, as an indicator of organizational grime. With organizational grime, we focus on the dependencies developed at a package level where design pattern realizations are present. This is similar to afferent coupling at a class level; however we take the abstraction a level higher to examine what happens at an organizational level.

3.5.3.2 Files containing the implementation

It is important to count the number of files containing the implementation of the various design pattern realizations. A constant number of implementation files while the pattern realizations continue to evolve and develop grime, indicates that changes and code are being added to existing files rather than being modularized, thus contributing to grime buildup.

3.5.3.3 Packages containing the implementation

The numbers of Java packages that participate in the implementation of design patterns give us another perspective into the organizational aspects of the implementation of the patterns. We track this measure, and expect that the number of packages that are in place at the beginning of a project will remain as is with very little change. Other measures track the way the packages are organized. Pattern organization is beyond the scope of this research. Studies by Booch et al. [15] for example, recommend namespace hierarchies no deeper than three levels, where namespace inheritance works just as with classes, where specializations inherit more general attributes. A specialized package can be used anywhere its parent can.

4. OBSERVATIONAL STUDY METHODOLOGY

The goal of the observational study is to find empirical evidence of the accumulation of pattern grime and possible pattern rot in design pattern realizations. In sections 4.1 and 4.2 we describe three open source systems used in this study and the tools used to mine the data. In section 4.3 we present sixteen hypotheses to be evaluated based on observations made to the three open source systems. Section 4.4 provides a detailed methodology for verifying pattern realizations against the RBML specification, and we identify the consequences that grime may have on different aspects of design adaptability and testability. To quantify consequences we select various surrogate measures.

4.1 Systems Studied

To study the consequences of grime, we have chosen three Open Source systems. Commercially developed systems were considered, but given the constraints associated with obtaining permission to mine and publish findings, we decided to focus on open source systems only. Since most of the tools available for data mining operate on Java systems, this was an essential criterion for selection. We chose two development tools and one database system implemented using the Java language. Given the number of releases, their usage statistics, and the number of years under development [79], we deemed the two development systems to be widely used and successful. The database system eXist, was not as successful as its counterparts in the database space.

4.1.1 JRefactory

JRefactory is written in the Java language and is available through SourceForge.net. JRefactory supports many refactoring operations in a system, and automatically updates the java source files as appropriate. Typical refactorings supported include repackaging, removing empty classes, moving fields and methods up and down an inheritance hierarchy, and extracting methods. We studied versions 2.6.12, 2.6.38, 2.7.05, 2.8.00, 2.9.00, and 2.9.19. These releases represent the evolution of the software over a period of almost four years. JRefactory also supports many integrated development environments, including jEdit, NetBeans, JBuilder, and Ant.

4.1.2 ArgoUML

ArgoUML 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 five years.

4.1.3 eXist

eXist is an XML database management system and stores data according to the XML data model. eXist is designed to support many web technology standards including XQuery, XSLT, various HTTP interfaces (Soap, xmlrpc, rest), and other XML specific database interfaces such as XUpdate, XMLDB, etc. We studied versions 0.8, 0.8.1, 0.8beta, 0.9, 0.9.2, and 1.0b1. They represent development for approximately 6 years.

4.2 Tools

Various tools are used to mine for data and understand the structure of the systems studied. Early tools were complemented by scripts written by the author. As better tools were found that supported the types of information we were mining for, we made a transition to them and less scripting was required. The following sections describe the tools used in this research.

4.2.1 Eclipse

Eclipse [25] is a widely used Open Source development IDE. It is an extensible framework with support for numerous plug-ins that extend the capabilities of the system. Eclipse is used in this research as a platform for the analysis of all source code under study. Eclipse provides us with an easy to use GUI that helps us organize and navigate the code. Additionally, it provides the capability to search for data and use readily available plug-ins to help with data mining.

4.2.2 JDepend

JDepend [53] is freely downloadable, and generates design quality metrics for the source code trees under study. A limitation of JDepend is that it "does not currently support the calculation of Ca (fan-in) and Ce (fan-out) in terms of the number of classes inside a package that have afferent or efferent couplings to classes inside other packages. Rather, JDepend calculates Ca and Ce strictly in terms of the number of packages with which a package has afferent or efferent couplings, based on the collective analysis of all imported packages." Thus, this tool helps our research in terms of gathering data that contributes to organizational grime.

4.2.3 JavaNCSS

This is a command line utility that was used early in our research, but was later replaced by more powerful tools. It is designed for gathering code metrics for the Java programming language. JavaNCSS [52] allows the user to gather such metrics a McCabe's CCN (Cyclomatic Complexity Number), NCSS (Non commenting Source Statements), and average values.

4.2.4 SemmleCode

Semmle is a UK company based in Oxford that develops and distributes .QL, an object oriented query language for structured data. Queries are constructed using the .QL language to compute specific measures, check for coding conventions, and find bugs. .QL has a syntax that is similar to SQL, and is implemented as an Eclipse plug-in named SemmleCode [78], which can be used to query any Java project. We use .QL to create queries that are suitable for analyzing realizations of various design patterns.

4.2.5 Pattern Seeker

Pattern Seeker [72] is a tool developed at CSU to help with the identification of design pattern instances in code. Pattern seeker is a command line interface based tool that searches for specific strings that may indicate intentional patterns. We use Pattern Seeker to help identify possible patterns, however manual checking is necessary to verify possible hits.

4.2.6 Design Pattern Finder

Design Pattern Finder [24] is a GUI based tool very similar to Pattern Seeker to help with identification of design pattern instances in code. It is a Windows application that searches source code directories for Gang of Four patterns. It works with .php, .java,

.vb, and .cs files. Results of searches can be saved to text files, and some patterns are configurable through xml files.

4.2.7 AltovaUML

AltovaUML [1] is a commercial suite of tools. It is a UML design tool with the added functionality to produce UML diagrams from code. When possible matches for design patterns are found, we reverse engineer the code using AltovaUML to create a UML diagram that can be checked against the RBML specification of a design pattern. This allows us to identify possible deviations of a pattern realization against the necessary elements described by the RBML.

4.3 Hypotheses

The following set of null hypotheses is tested throughout the observational case study. Hypotheses concerning class, modular, and organizational grime have been postulated. Additionally, hypotheses regarding the consequences of grime buildup on testability and adaptability are also tested.

Decay, Rot, and Grime Buildup

$H_{1,0}$: There is inconsequential *pattern rot* in design pattern realizations. The number of *core deviations* is minor as a system evolves.

$H_{2,0}$: There is inconsequential *grime buildup* in design pattern realizations. The amount of grime buildup is minor as a system evolves.

$H_{3,0}$: The number of pattern realizations do not increase as the system evolves over time.

Class Grime Buildup

$H_{4,0}$: The increases in the number of public methods in a class that belongs to a pattern, is inconsequential.

$H_{5,0}$: The increases in public attributes (fields) in a class that belongs to a pattern, is inconsequential.

$H_{6,0}$: The increases in the sizes of classes that belong to pattern realizations is inconsequential.

Modular Grime Buildup

$H_{7,0}$: The afferent coupling C_a (fan-in) level increases of pattern realizations over time are inconsequential.

$H_{8,0}$: The efferent coupling C_e (fan-out) level increases of pattern realizations over time are inconsequential.

Organizational Grime Buildup

$H_{9,0}$: The total number of packages that participate in the implementation of a design pattern remains constant over time.

$H_{10,0}$: The total number of physical files that make up the implementation of design patterns remains constant throughout the evolution of the system.

$H_{11,0}$: The package level afferent coupling C_a (fan-in) of packages containing pattern realizations is inconsequential.

Consequences of Grime Buildup

$H_{12,0}$: There is no correlation between changes in LOC and design pattern grime buildup.

$H_{13,0}$: The adaptability of design patterns, measured by the Instability ratio of Ce / (Ca + Ce) tends to remain the same as patterns evolve.

$H_{14,0}$: The adaptability of design patterns, measured through its Abstractness value **A**, is inconsequential.

$H_{15,0}$: Grime buildup has a higher impact on the testability than the adaptability of design patterns.

$H_{16,0}$: The minimal number of test requirements necessary to maintain test effectiveness remains constant as grime buildup increases.

4.4 Evaluating Hypotheses, Testability and Adaptability

To evaluate the hypotheses and the effects of grime buildup on the external attributes of testability and adaptability of design pattern realizations, we track the evolution of the Factory, Adapter, Singleton, State, Iterator, Proxy, and Visitor patterns in every system under investigation. The RBML for each pattern can be found in appendix A. For each realization we perform the following steps:

1. Create the RBML pattern specification. RBML and the Unified Modeling Language (UML) are used to specify selected pattern structures. The RBML is used to describe the essential classifiers and relationships necessary for a pattern realization to be in compliance with the intended design. Pattern specifications are verified by field experts.
2. Use *Pattern Seeker* and *Design Pattern Finder* to mine every version of software under study. These tools will provide coarse statistics to help identify the total number of realizations of a given pattern. Additionally, these tools help identify the number of references to such patterns.
3. Use *AltovaUML* to reverse engineer the UML diagram of pattern realizations to check for conformance against the RBML specification. Checking for conformance is a manual process which could be automated [57].
4. Mine all versions of software using available tools, and develop queries in the *.QL SemmleCode* language to capture all measurements specified in section 3.4 for the participants of the realizations of the pattern under study. Each query is designed specifically for each pattern. All statistics are gathered and graphed. Because it is very difficult to gather statistics for each individual pattern realization, the statistics gathered are an aggregate of all realizations of a given design pattern.
5. Analyze observations for the pattern realizations.
6. Evaluate hypotheses.

The process of counting relationships that form as a result of grime buildup was automated (step four), however manual intervention was still required to distinguish between relationships that are not part of the intended role of the pattern, and those that extend the pattern in intended ways (step three).

4.4.1 Testability

We use two methods to evaluate testability:

1. Look for the appearance of design anti-patterns, and
2. Examine the effects of increases in relationships on test requirements.

4.4.1.1 Anti-patterns

The consequence of grime increase may manifest itself as anti-patterns in a design. Anti-patterns can make testability efforts unmanageable and can quickly render tests ineffective. This is especially true with dynamic patterns, where inheritance hierarchies can grow unbounded, causing the potential number of paths that need to be tested to grow very fast.

To evaluate testability, we look for empirical evidence of the emergence of testing anti-patterns in designs. An anti-pattern *"describes a commonly occurring solution to a problem that generates decidedly negative consequences."* [19] Anti-patterns develop as a result of increased coupling.

To track the development of testing anti-patterns we follow the evolution of various realizations of the Visitor, State, and Singleton patterns over a period of four years in the JRefactory [54] open source system. We mine for different types of anti-patterns, but in particular, we look for empirical evidence of anti-patterns described by the work of Baudry et al [6], [7], [8], and [9]. They describe two anti-patterns (testing conflicts) that weaken design patterns. The effects of these anti-patterns are further compounded by inheritance hierarchies and polymorphism. Figure 4.1 shows the *concurrent-use-relationship*, where two paths exist from A to C. Class A has a transitive use-path through B and B' to C. This scenario is described as an anti-pattern because A can change the state of C through one path, and read C from another path. Thus, consistency needs to be maintained. Maintaining consistency can become difficult, especially when multiple paths exist through a polymorphic hierarchy. As the number of relationships in design patterns grow, some design pattern realizations are likely to develop this form of anti-pattern.

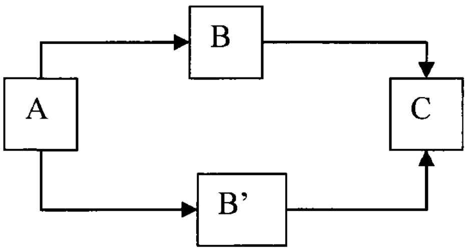


Figure 4.1 Concurrent-Use-Relationship

The second anti-pattern is called *self-usage*. Figure 4.2 displays its structure. Self-usage identifies potential self referential loops in the design, which must be tested for potential infinite loops. Self references can occur at a single class level or through multiple transitive class paths.

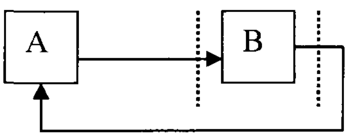


Figure 4.2 Self-Use-Relationship

In addition to the anti-patterns described by Baudry et al., we also look for additional anti-patterns as described by Brown et al. [19] that develop as a result of grime buildup.

We look for evidence of the *lava flow* anti-pattern, where some occurrences of code remain unchanged through the lifecycle of a product. As the rest of the system evolves to conform to a new operational domain, this realization of a design patterns lies dormant. Test cases and requirements of the dormant realization of the design pattern may still be viable at a unit level, however when tested at a system level, this pattern may not work correctly because the environment and the other code in the system around it have changed. Dormant code, if not checked early, can lead to further deterioration of the system and testing requirements because new developers do not want to remove code that is not understood.

We also look for evidence of the *swiss army knife* anti-pattern. In a swiss army knife, classes implement too many methods. They exhibit a constant increase in methods that may not have anything to do with the original intent of the class in the design pattern, or a sudden implementation of methods for added new interfaces.

4.4.1.2 Testing requirements

The second method used to evaluate testability examines the effects of increases in relationships (associations, realizations, and dependencies) that develop as a result of grime buildup on test requirements.

Class relationships are subject to many kinds of faults which must be tested. Examples of faults include wrong multiplicities, which as a result can generate missing or erroneous links between classes, errors in the creation or deletion of the runtime objects that must satisfy the constraints specified in the UML, etc. In the case of a binary association between classes, there exist four possible combinations that must be tested, Binder [12]. For each combination an *accept* and a *reject* test case is necessary, thus yielding eight possible scenarios. In the case of n-*ary* associations there exist $8n$ possible scenarios that must be tested. The formula $8n$ covers the basic boundary conditions, but an additional constant number of tests can be added to cover typical scenarios that are found from operational profiles. Thus, we express the minimum number of n-ary association test requirements necessary using the linear model $A(n, k) = 8nk + c, c>=0$. The variable k indicates the total number of such relationships found in the release. The consequences of not testing such combinations increase the fault proneness of the system.

In the case of aggregation, which is a special kind of association, there exists a relationship between the whole and its parts, however the lifetime of either is independent. Since aggregation is a kind of association, $A(n, k)$ already covers the multiplicity tests, however additional test cases are necessary to cover the test requirement for independent creation and destruction of the whole and each of its parts. We express the minimum number of test requirements as $AG(n, k) = A(n, k) + 4nk$.

Composition is a kind of relationship that does require testing for the transitive property. Composition is also a kind of association where there exists a relationship between a whole and its parts, but one where the part is created and destroyed alongside the whole. In other words, the lifetime of a part is dependent on the whole. Since $A(n, k)$ already covers the multiplicity tests, we express the minimum number of test requirements as $C(n, k) = A(n, k) + 2nk$. The last term of this equation covers the sequential creation and destruction of the whole and its parts.

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

Finally, the dependency relationship is fully application dependent and the number of test requirements to cover such temporal relationships varies. Thus, $D(n, k) = c, c>=0$.

The equations (referenced in section 6.4) are numbered as follows:

I. $A(n, k) = 8nk + c, c>=0$

II. $AG(n, k) = A(n, k) + 4nk$

III. $C(n, k) = A(n, k) + 2nk$

IV. $G(n, k) = 2nk$

V. $D(n, k) = c, c>=0$

As systems evolve, new relationships develop between classes. Added relationships represent added test requirements. These relationships may or may not have been intended in the original design. More often than not, such relationships are the consequence of modular grime buildup. Without the necessary updates to the testing suite of such systems, the possibility of faults, as expressed by the equations, grows.

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

The efferent coupling C_e (fan-out) metric is also used to evaluate the testability of design pattern realizations. Efferent coupling is measured at the package level as well as at the pattern realization level. We observe what happens to the package that contains the realization of the pattern under study by measuring its efferent coupling. Further, we also measure the efferent coupling of the pattern realization itself.

4.4.2 Adaptability

When grime buildup occurs, the effort required making changes potentially increases, and refactoring may be necessary in order for the pattern to accommodate further changes. It is also clear that detection of the pattern realization is made much harder because the realizations become obscured. Additionally, the fault proneness and ripple effects that result from the accumulation of modular grime buildup are well known [2], [5], and [17].

Adaptability is a non-functional attribute of software, and measures have been proposed by Gilb [37] to track the total number of changes to a system module, or to estimate the amount of time spent on a module [30]. These measures focus on overall designs however, rather than design patterns. In addition, these measures are not easily available in many Open Source systems, and it is not feasible to mine for such information. We use measures described in section 4.4 concerned with modular grime, class grime, and organizational grime that affect adaptability of design patterns.

We observe what happens to grime measures C_a (afferent coupling), and C_e (efferent coupling), as a result of code changes (ΔLOC) made to participant classes in design patterns. If we observe that the value of afferent coupling increases, then the pattern's responsibility increases, which implies that the pattern is less adaptable as more software artifacts depend on the pattern. Intuitively, it is much harder for a developer to make changes to the design pattern without affecting related classes. Increases in efferent coupling make a pattern unstable as the pattern becomes dependent on other classes. Thus, growth observed in the ΔLOC in existing design pattern participants can be indicative of concerns that affect adaptability if modular grime measures also grow. However, not all changes in lines of code are necessarily grimy and some patterns may exhibit changes in LOC without corresponding increases in grime measures. We also use the instability measure on all pattern realizations studied to evaluate a pattern's ability to accommodate change. A value of zero or one approaching zero indicates a pattern that is hard to modify.

4.4.2.1 The Relationship between changes in LOC and grime counts

Adaptable design patterns require less maintenance and are easier to comprehend. If design patterns are extended using agreed upon techniques then changes experienced in the number of lines of code that participate in design patterns will not contribute to grime buildup. Intuitively, classes that experience high number of changes require more maintenance. Thus, we use the number of changes in lines of code (ΔLOC) of participant design pattern classes to see if they are related to grime buildup. Changes in lines of code can either be additions or deletions.

We perform statistical analyses designed to determine if ΔLOC are indicative of grime buildup. Correlation and multivariate regression analyses are performed using SAS statistical software [76]. Correlation coefficients are calculated using the Spearman method as opposed to Pearson's correlation because the sample sizes are small and non-parametric techniques are necessary. Regression analysis is used to evaluate any possible structural relationship between ΔLOC and grime measures; it is not used to predict dependent variables. Since grime buildup occurs as a result of changes in lines of code, we cannot speculate that ΔLOC can be predicted by grime measures; however we can use regression to find a relationship that is statistically significant.

The simple linear regression model assumes the following [70]:

- The expected errors for each independent variable all have a value of zero: $E(\varepsilon_i) = 0$ for all i.
- The errors all have the same variance: $\mathrm{Var}(\varepsilon_i) = \sigma^2_\varepsilon$
- The errors are independent of each other.
- The errors are all normally distributed; ε_i are all normally distributed for all i.

These assumptions are a threat to validity; however we use them consistent with empirical research studies in software engineering. The multivariate regression model is written as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \varepsilon$$

where the y denotes the dependent variable ΔLOC, and the x_i denote the independent variables. In general, β_j where $j \neq 0$, represents the expected change in y for a unit increase in x_i while holding all other xs constant. The parameter β_0 is the y-intercept.

4.4.2.2 Instability of a design pattern

Instability [64], [66], [67] is a measure of how hard it is to change a software artifact without changing other artifacts. This measure is used as a surrogate for adaptability of design patterns. The value is a ratio of its fan out to the total number of relationships.

$$\text{Instability} = Ce / (Ca + Ce)$$

A value of one represents high instability (unstable) and it is representative of new artifacts that have dependencies on already established artifacts. Brittle design patterns have high values, but are easier to adapt because the number of incoming dependencies is small. In other words, the pattern can be modified without significant consequence. A value of zero represents a stable artifact, and it is typical of artifacts that have been in place since early design of the software. These artifacts are hard to modify because many artifacts depend on it. It is much harder to adapt a design pattern with a low Instability value.

Other notions of stability refer to parts of a design that show how characteristics tend to remain the same. Kelly [55] observes that "good software designs aim to stabilize parts of the software". The metric we calculate for design pattern realizations, namely *instability*, may have a constant, and hence "stable" value that is close to one, however this would be indicative of a young pattern. The instability metric is computed for every design pattern studied.

4.4.2.3 Abstractness of a design pattern

The *abstractness* (**A**) [64] of a design pattern can be measured as the ratio of abstract classes (including interfaces) to the total number of participating classes. The range of this measure is [0..1], with 0 indicating that the pattern realization contains all concrete classes, and 1 indicating all abstract classes. Values approaching one indicate that the pattern is potentially more adaptable through extensibility. In order to understand how grime affects abstractness we need to set an expected value of abstractness for every design pattern studied. We can do this by referencing the RBML of a design pattern. This value may indeed be a range of accepted abstractness. As grime buildup occurs, classes that belong to a design pattern develop relationships with classes that do not play a role in the pattern as specified by the RBML. The "external" classes and/or interfaces contribute to changing the abstractness value of the pattern.

5. RESULTS

Table 5.1 lists all the systems and versions of software studied. 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 take 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.

Table 5.1 Software versions and release dates studied

JRefactory		**ArgoUML**		**eXist**	
Version	Release date	Version	Release Date	Version	Release Date
2.6.12	1/2001	0.10.1	10/2002	0.8beta	6/2002
2.6.38	2/2002	0.12.0	8/2003	0.8	8/2002
2.7.05	7/2003	0.14.0	12/2003	0.8.1	8/2002
2.8.00	8/2003	0.16.0	7/2004	0.9	1/2003
2.9.00	10/2003	0.18.1	4/2005	0.9.2	8/2003
2.9.19	5/2004	0.20.0	2/2006	1.0b1	10/2006
		0.22.0	8/2006		
		0.24.0	2/2007		

At a high level we gather statistics for many design patterns, however when observing trends at the realization level of individual patterns, we track the patterns described in table 5.2.

Table 5.2 Design patterns studied for each system

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

We use queries written in the .QL language, described in section 4.2, and customized scripts to generate the measurements. We graph the results against calendar dates. Since we are studying the evolution of software patterns over time, using calendar dates is a necessary and implied time scale. 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. [60] suggest the latter approach in the study of the OS/360 system; however evolution studies of the Linux Operating System by Godfrey and Tu [38] suggest using calendar dates.

5.1 Modular Grime Results

Modular grime is measured by counting the number of relationships that design pattern realizations develop as they evolve. We observe how the overall number of references to design pattern realizations change. References describe what is occurring in the environment surrounding the pattern realization. An increase in these numbers suggests that pattern realizations are being used, thus changes in pattern realizations affect other parts of the system, and thus its adaptability. The total number of references is a superset of afferent coupling. Reference counts also include definitions used in generic types, two way dependencies, and global definitions. References are obtained by using the Pattern Design Finder tool, which has less accuracy than other tools. For example, this tool accounts for references found in comments which are not valid but the tool has no way of discriminating between them.

The ability to differentiate counts for every realization of a design pattern is cost and time prohibitive, thus we aggregate the measures for all realizations of individual patterns studied. Relationship counts for afferent and efferent couplings are obtained using the .QL language, and are only gathered for the patterns shown in Table 5.2. We only focus on classes that participate in a realization of a pattern and (to save on computational time) whose relationship count equals or exceeds five.

Intentional Adapter patterns are typically named with classes that contain the "adapter" string as a suffix. There is no other way to really identify such patterns because the *adaptee* is typically named with its original name. This makes the search for Adapter patterns difficult.

The Adapter pattern as described by Gamma et al. [36] has two common implementations. The first implementation is the *class adapter* version, where public inheritance is used to obtain the interface from the *target* class, and private inheritance is used to obtain the implementation from the *adaptee* class. Clearly this is not feasible in Java environments where multiple inheritance is not supported. An alternative implementation in Java is the realization of an interface that uses the "implements" keyword, and an extension of the adaptee via regular inheritance using the "extends" keyword. A second approach is the *object adapter* version, where object composition is used to combine classes with different interfaces.

We also note that the Java language provides a number of built in Adapters. Most of these adapters are found as part of the Windows [86] events interfaces to help simplify bookkeeping. Examples include *WindowAdapter, ContainerAdapter, FocusAdapter, MouseMotionAdapter, KeyAdapter,* and *MouseAdapter.* We do not gather statistics for built in adapters. In some cases we find adapters that extend Windows built in adapters, and we do gather statistic for these patterns because they are clearly intentional and beyond what the Windows built-ins provide.

We also calculate the Instability of design pattern realizations. Instability ranges between the values of zero and one and it is dependent on the afferent and efferent results. Section 4.4.2.2 provides a description of this measure.

5.1.1Reference Counts and Afferent Coupling

Reference counts are less accurate than the afferent coupling measures gathered. We use *Design Pattern Finder*, which takes into account a much less specialized set of rules when counting and thus the counts are higher. Reference counts are gathered for all possible realizations of design pattern realizations that the tool "believes" it has found. In all three systems we see a slight and steady increase in all design pattern realizations. Notable exceptions are the reference counts in the ArgoUML system where pattern realizations tend to follow a constant path of evolution.

Figures 5.1 and 5.2 display the reference count results for JRefactory. Figure 5.2 is the same as 5.1 without the numbers for the State and Visitor patterns. Figure 5.2 helps us visualize the results for patterns with smaller counts.

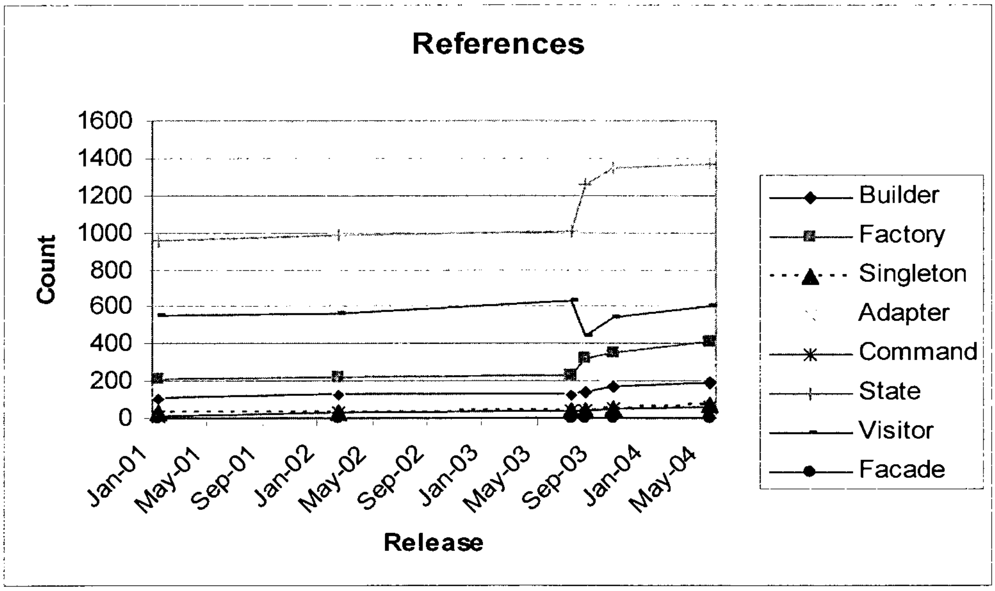


Figure 5.1 JRefactory reference counts to all possible realizations of various design patterns

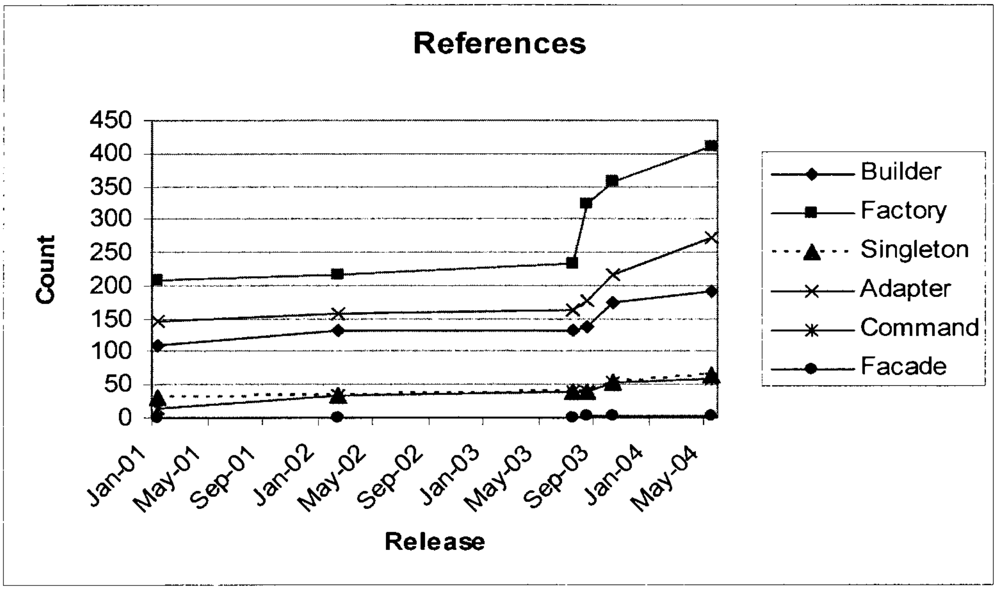


Figure 5.2 Figure 5.1 minus the State and Visitor patterns

JRefactory sees significant increases in reference counts beginning in August 2003 (release 2.8.00). This is observed in most design patterns and is an indication of increased usage. The Visitor pattern shown in figure 5.1 however, exhibits a slight decrease after the August 2003 release, and never experiences reference count increases similar to those encountered by other patterns.

Figures 5.3 and 5.4 display the reference count results for ArgoUML, and eXist respectively. In ArgoUML's case, with the exception of the Façade pattern, reference counts tend to remain constant through all releases of the software. The observed increase in reference counts for the Façade pattern is significant, and begins in December 2003 (release 0.14.0) and continues until July 2004 (release 0.16.0) before exhibiting a much lower rate of growth. eXist's reference counts increase at a steady rate until August 2003 (release 0.9.2) before they begin to increase more significantly for all patterns studied except the Proxy pattern. It is important to note that the release distance between versions 0.9.2 and 1.0b1, measured in calendar months, is thirty eight months. All five previous releases were separated by at most seven months. Thus, the overall growth observed in reference counts appears to be consistent with previously observed growth rates.

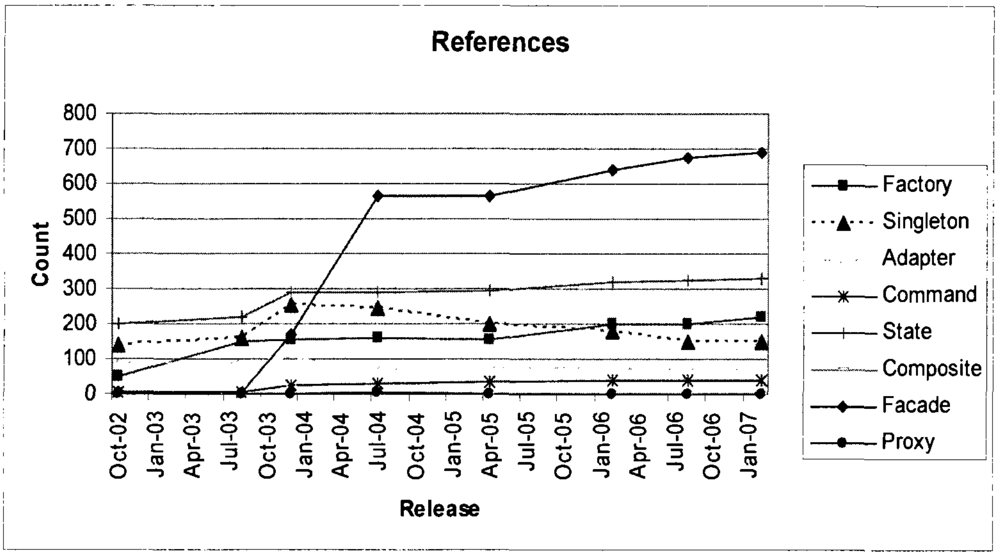


Figure 5.3 ArgoUML reference counts to all possible realizations of various design patterns

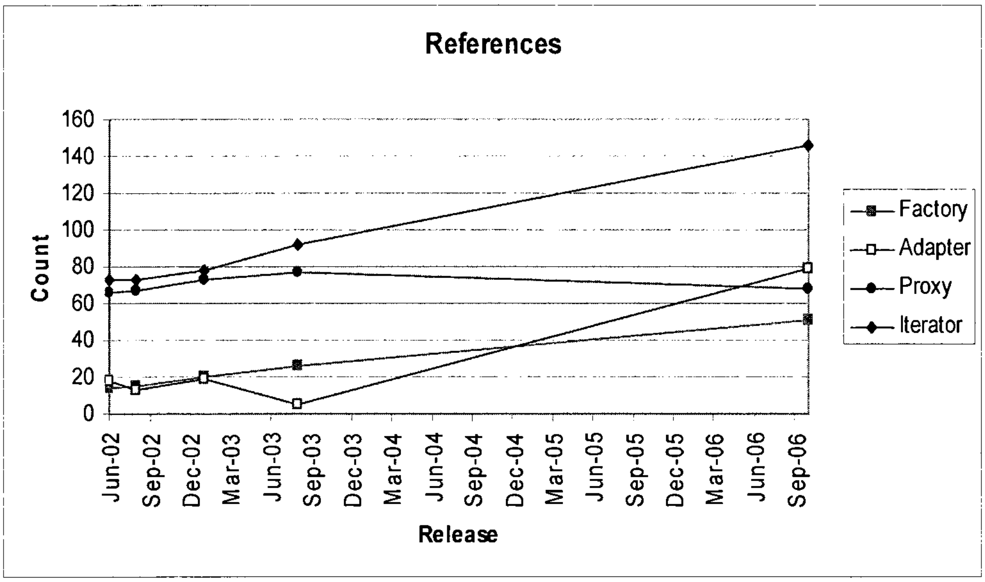


Figure 5.4 eXist reference counts to all possible realizations of various design patterns

Afferent coupling metrics are calculated and displayed in figures 5.5 through 5.8. We find realizations of the Adapter and Factory patterns in all systems studied. Adapter pattern realizations show evidence of afferent coupling growth in both JRefactory and eXist; however ArgoUML's realizations experience 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 tend to remain constant.

Figure 5.5 and 5.6 display the afferent coupling results for JRefactory. Figure 5.6 is the same as 5.5 minus the numbers for the Visitor pattern. Figure 5.6 helps us visualize the results for patterns with smaller counts.

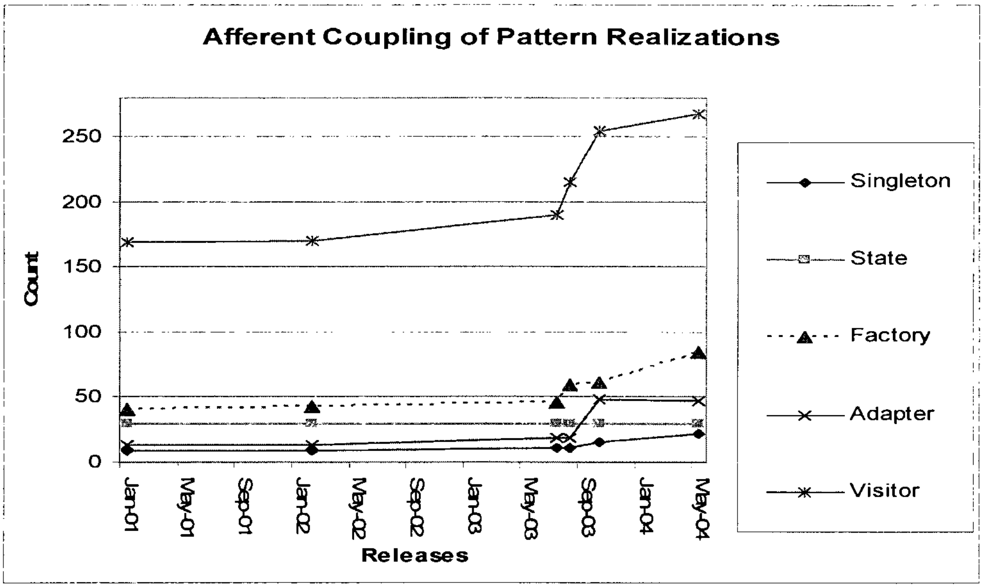


Figure 5.5 JRefactory afferent coupling counts to all possible realizations of various design patterns

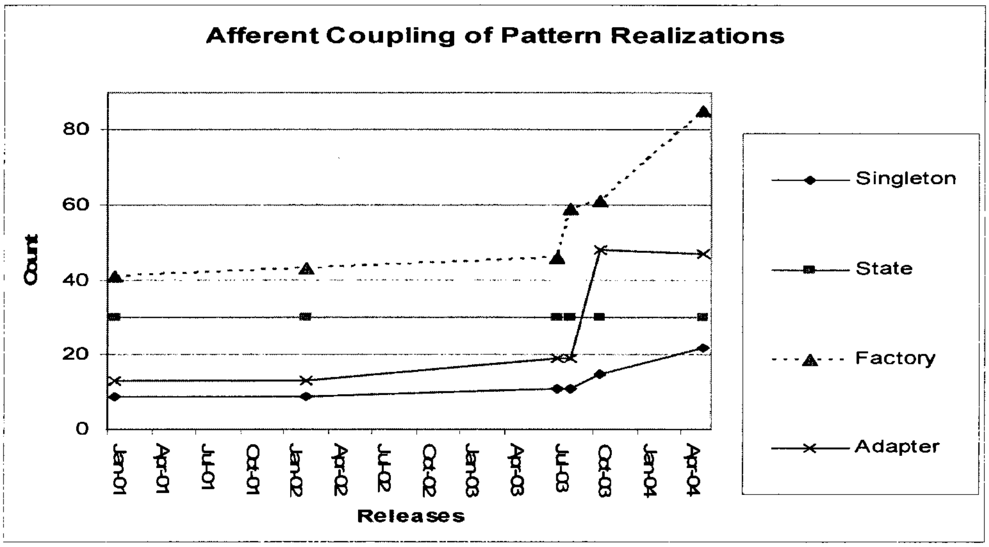


Figure 5.6 Figure 5.5 minus the Visitor pattern counts

Figures 5.7 and 5.8 display the afferent coupling count results for ArgoUML, and eXist respectively.

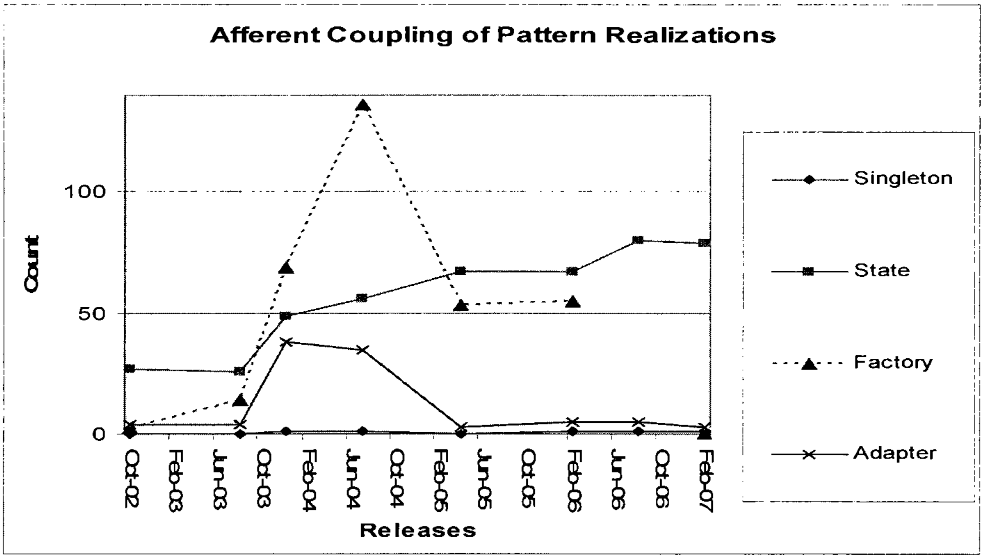


Figure 5.7 ArgoUML afferent coupling counts to all possible realizations of various design patterns

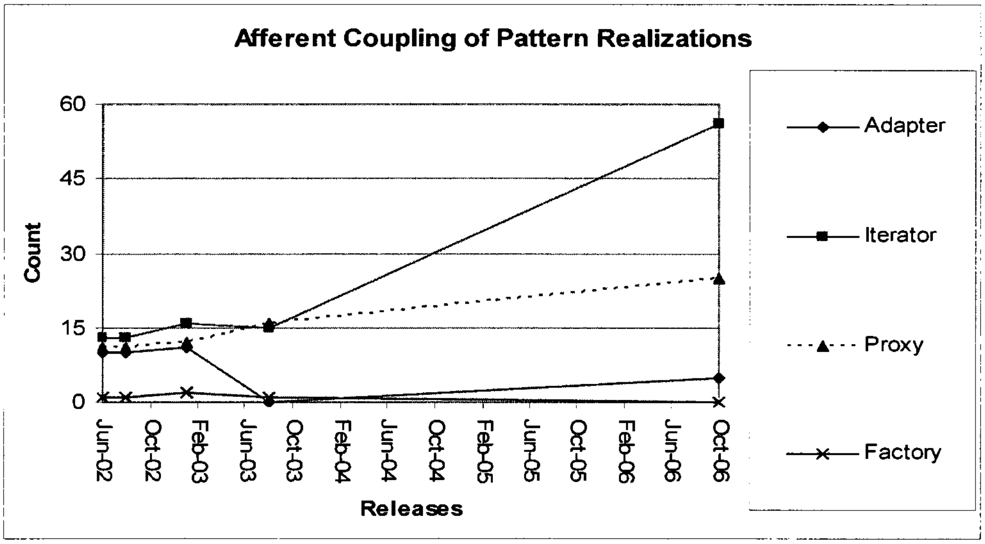


Figure 5.8 eXist afferent coupling counts to all possible realizations of various design patterns

Factory pattern realizations also show a tendency to grow; however unlike JRefactory where growth of realization increases significantly, in the ArgoUML source code we find that the first four releases implement all realizations of the pattern using classes. Beginning in April 2005 (release 0.18.1), most of the Factory pattern classes are converted to Java interfaces. The patterns retain their names, but they are now interfaces instead of classes. We had to modify our pattern recognition query in order to gather statistics of the now converted interfaces. The drop in afferent coupling between July 2004 (release 0.16.0) and April 2005 (release 0.18.1) is due to the UmlFactory class. The class originally had a C_a count of fourty-four and when separated into an interface-implementation combination its C_a count dropped to three, thus reducing 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. We thus do not provide those numbers in our results because they would not make sense. The interfaces for the factories remain, however their implementations are gone. The reference counts for the Factory pattern in the ArgoUML system still see a slight increase after February 2006 in versions 0.22.0, and 0.24.0, however these are references to the existing Factory realizations. The implementations of the factory classes stopped with version 0.20.0. eXist' s realizations of the Factory pattern tend to remain constant. We do not find evidence of the Factory pattern realizations in the last version of eXist studied.

We find realizations of the Singleton pattern in JRefactory and ArgoUML. Only a single realization is available in the latter and its afferent level counts remain stable. JRefactory's afferent level counts show constant values until August 2003 (release 2.8.00) when C_a levels climb sharply.

State pattern realizations were found in both JRefactory and ArgoUML's code. The evolution of these realizations follows distinct paths. The State pattern in JRefactory yielded three instances. All three instances never changed (evolved) through all the releases even though the number of references to the pattern realizations did increase. This indicates that the pattern realizations are not dead code and they indeed continue to be used. ArgoUML's realizations of the State pattern show a steady growth in its afferent coupling levels. We do not find any major refactoring efforts after close inspection of the code.

The only viable realizations of the Visitor pattern were found in JRefactory. Similarly, we only find the Proxy and Iterator patterns in eXist. In all three cases we clearly see very defined and steady increases in afferent coupling counts.

5.1.2 Efferent Coupling Counts

Figures 5.9 and 5.10 display the efferent coupling results for JRefactory. Figure 5.10 is the same as 5.9 minus the numbers for the Visitor pattern. Figure 5.10 helps us visualize the results for patterns with smaller counts.

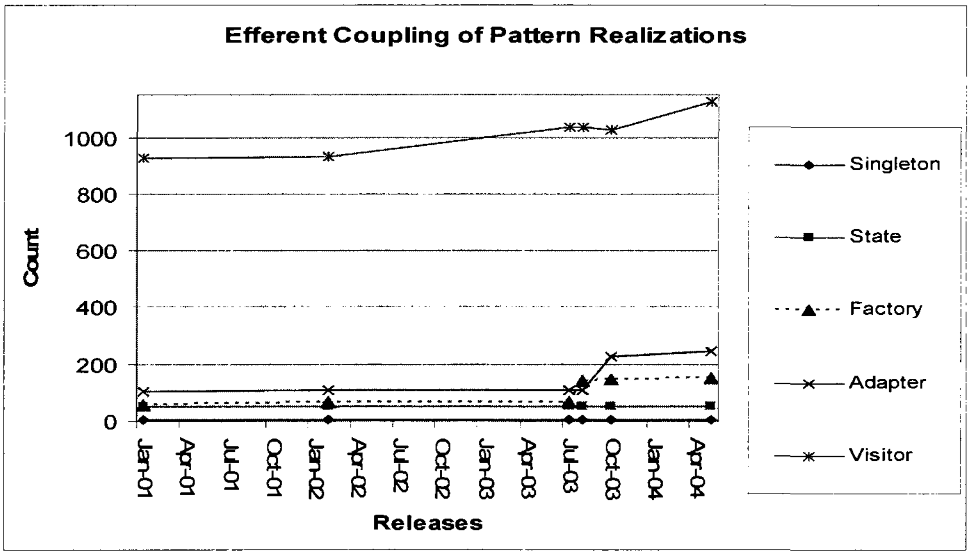


Figure 5.9 JRefactory efferent coupling counts to all possible realizations of various design patterns

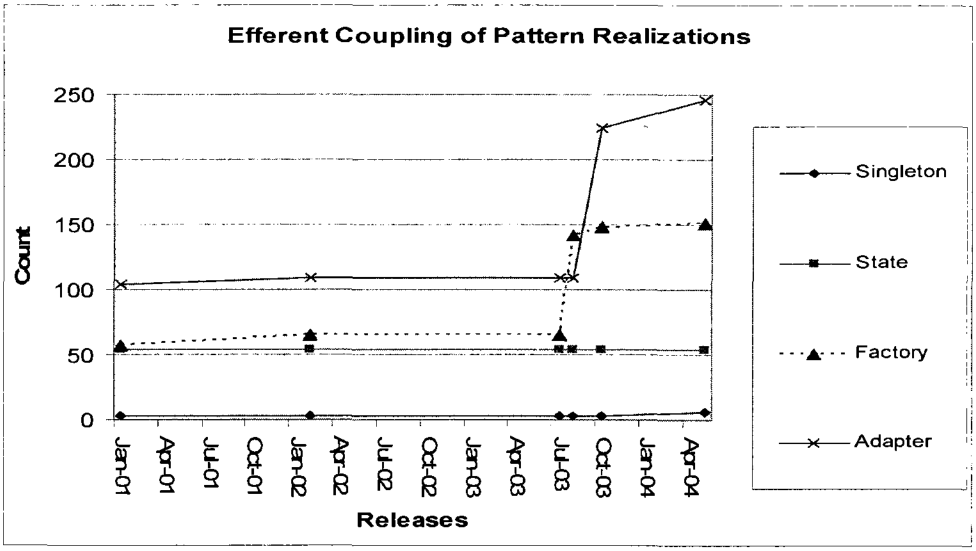


Figure 5.10 Figure 5.9 minus the Visitor pattern counts

Figures 5.11 and 5.12 display the efferent coupling count results for ArgoUML, and eXist respectively.

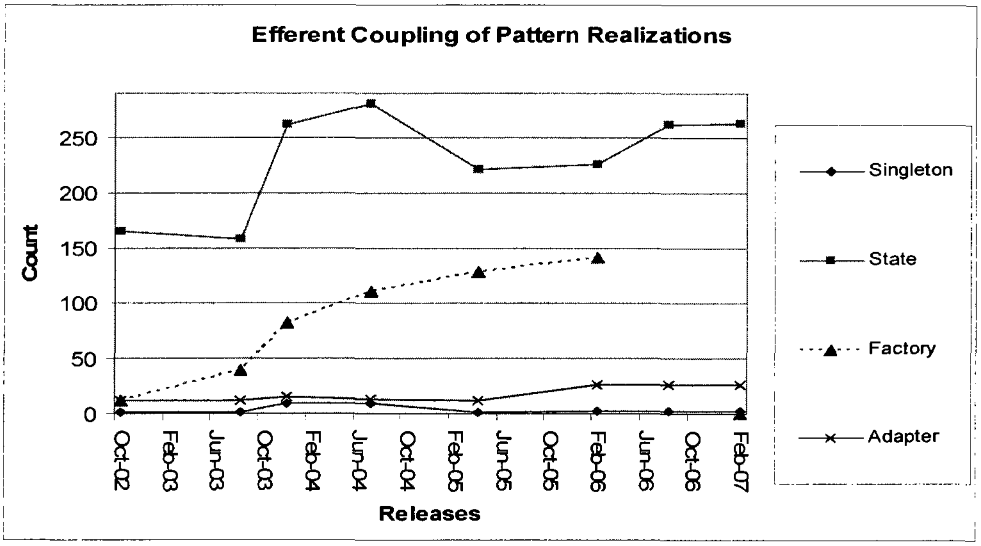


Figure 5.11 ArgoUML efferent coupling counts to all possible realizations of various design patterns

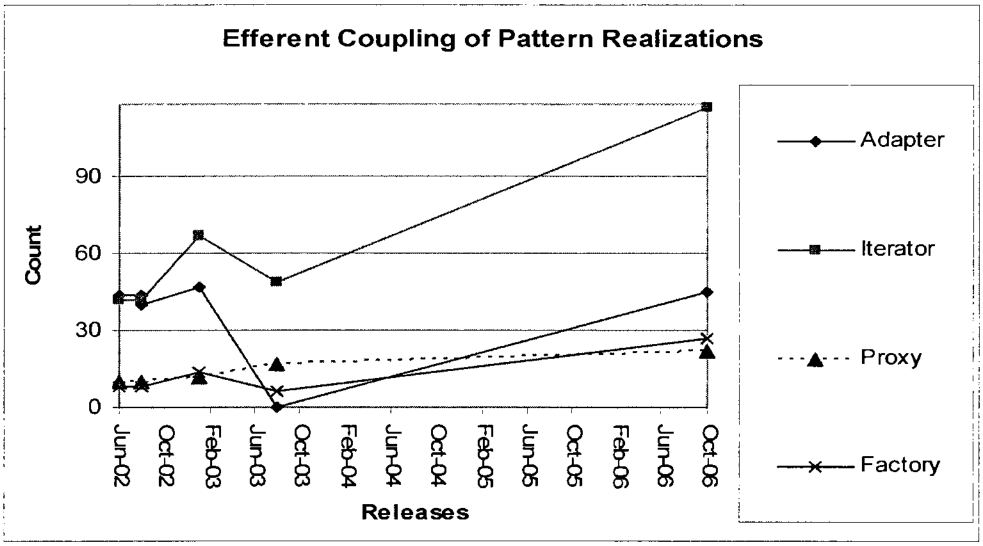


Figure 5.12 eXist efferent coupling counts to all possible realizations of various design patterns

We first focus on realizations of the Adapter and Factory patterns; which are found in all systems studied and shown in figures 5.9 through 5.12. All realizations in all systems studied show steady growth in efferent coupling. In the case of JRefactory, we see a marked increase after the July 2003 version (release 2.7.05), and in the eXist database system we observe a marked increase after the August 2003 version (release 0.9.2). The realizations observed in the ArgoUML system shows more controlled growth in the case of the Factory pattern and an almost constant or bated growth in the case of the Adapter pattern. 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.

The Singleton pattern is found in JRefactory (figure 5.10) and ArgoUML (figure 5.11). As expected, C_e levels remain constant for both systems. In JRefactory we observe 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. The Singleton pattern tends to remain unchanged once implemented and this is consistent with the results obtained.

As already described, the State pattern realizations remain unchanged throughout their lifecycles in the JRefactory system. Figure 5.10 shows the efferent coupling count. Both C_a and C_e remain unchanged, however the reference counts do increase as the pattern ages. In ArgoUML we observe moderate and subdued growth.

The Visitor pattern in JRefactory (figure 5.9) experiences continual and consistent growth throughout its lifecycle as do realizations of the Proxy and Iterator patterns in the eXist database (figure 5.12).

5.2 Class Grime Results

Class grime is measured by focusing on the classes rather than the relationships that form part of a design pattern. Measurements are 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. In rare occasions we observe patterns extended via generalizations.

5.2.1 Class Size

All systems studied JRefactory, ArgoUML, and eXist exhibit growth in total lines of code (LOC) for every design pattern realization. JRefactory exhibits a sudden jump after July 2003; however the other systems tend to show a more consistent rate of growth. The Singleton pattern realizations found in JRefactory and ArgoUML show little change. It appears that once in place, the latter tends to remain as-is.

Figures 5.13 and 5.14 display the results for JRefactory. Figure 5.14 is the same as 5.13 minus the numbers for the Visitor pattern. Figure 5.14 helps us visualize the results for patterns with smaller counts.

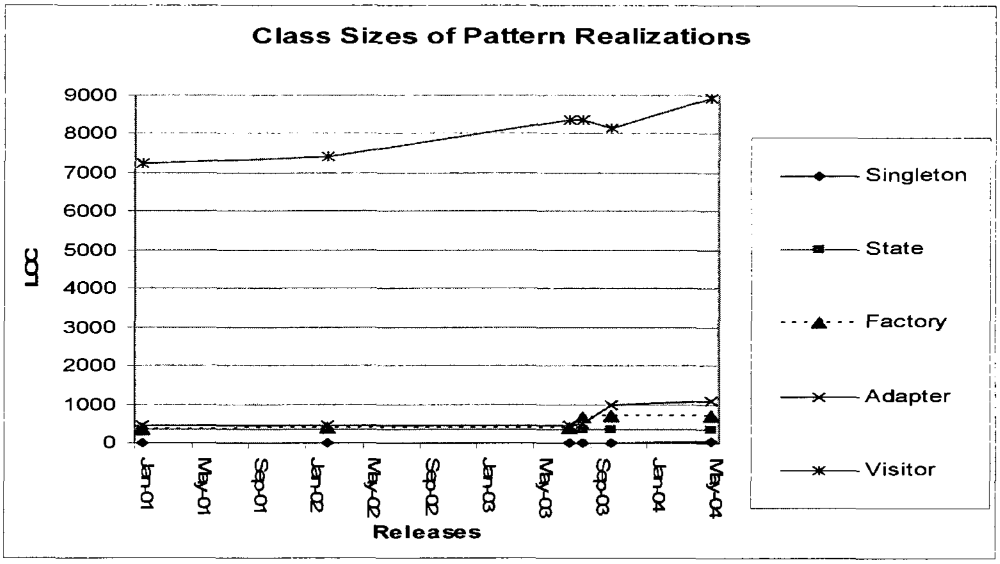


Figure 5.13 JRefactory class sizes in LOC of all possible realizations of various design patterns

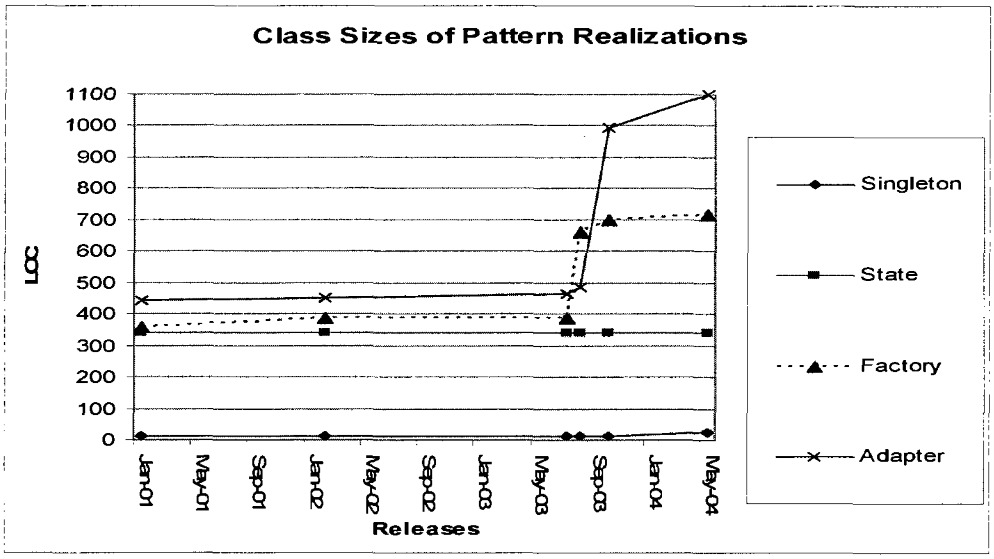


Figure 5.14 JRefactory class sizes in LOC of all possible realizations of various design patterns minus the Visitor pattern

Figures 5.15 and 5.16 display the LOC count results for ArgoUML, and eXist respectively.

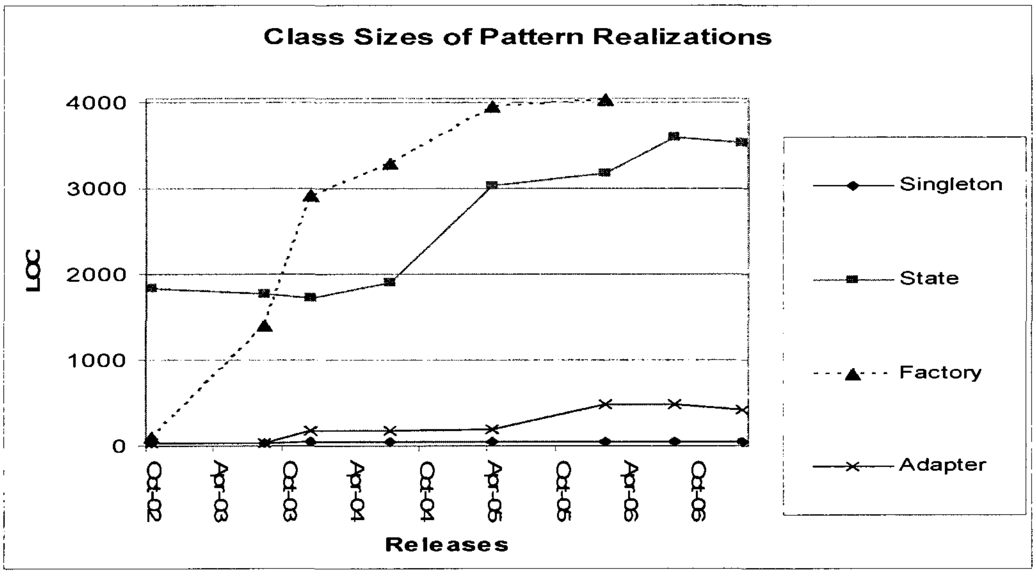


Figure 5.15 ArgoUML class sizes in LOC of all possible realizations of various design patterns

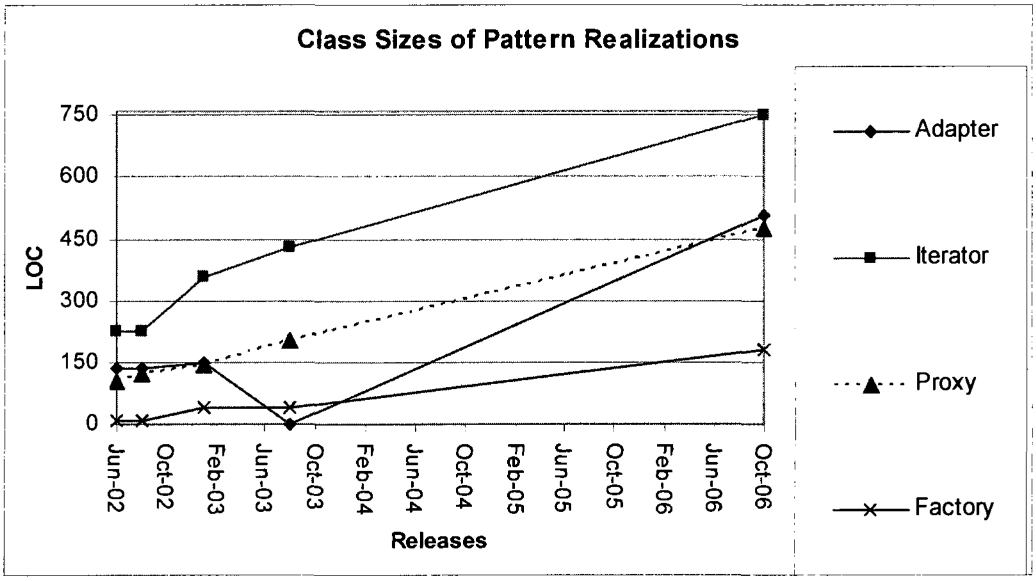


Figure 5.16 eXist class sizes in LOC of all possible realizations of various design patterns

5.2.2 Number of Attributes

With the exception of only four design patterns, all realizations across all systems studied exhibit little change in terms of growth of additional attributes. This is a positive result because it tells us that no new state, beyond what is expected as a responsibility of a design pattern, is being added as the pattern evolves. In JRefactory we see modest growth in the Adapter and Visitor patterns. In ArgoUML we observe a steady growth in the State pattern, and in eXist we see a sharp increase in the last version of the Iterator pattern.

Figures 5.17, 5.18, and 5.19 show the respective results for the total number of attributes measure of JRefactory, ArgoUML and eXist systems.

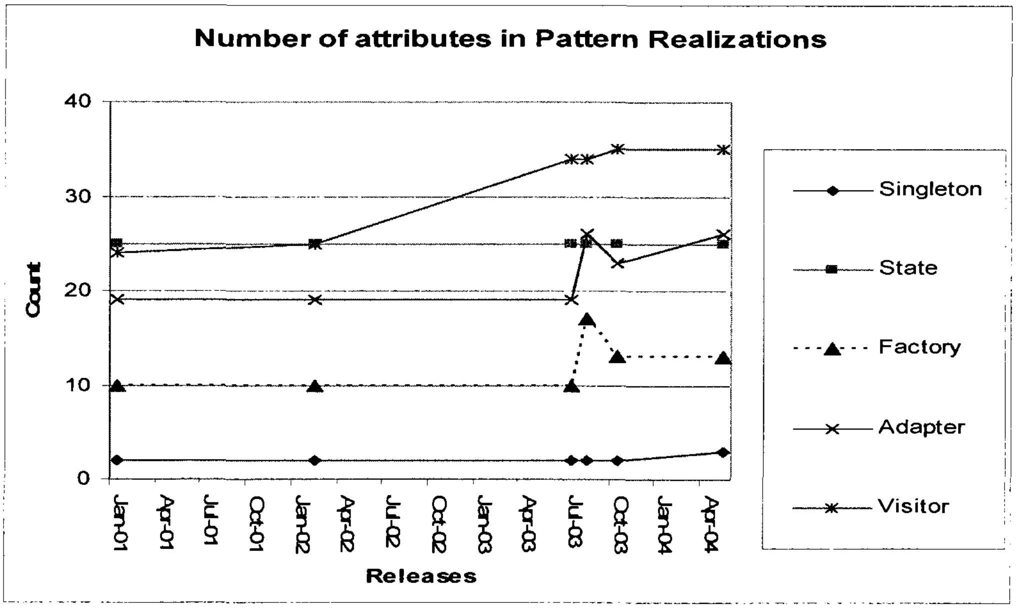


Figure 5.17 JRefactory number of attributes of all possible realizations of various design patterns

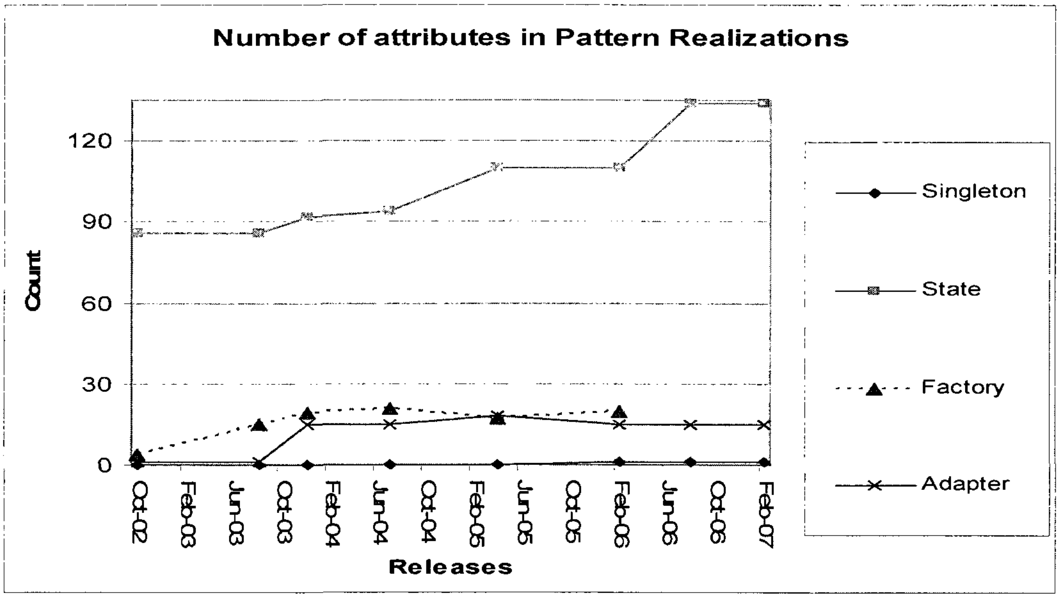


Figure 5.18 ArgoUML number of attributes of all possible realizations of various design patterns

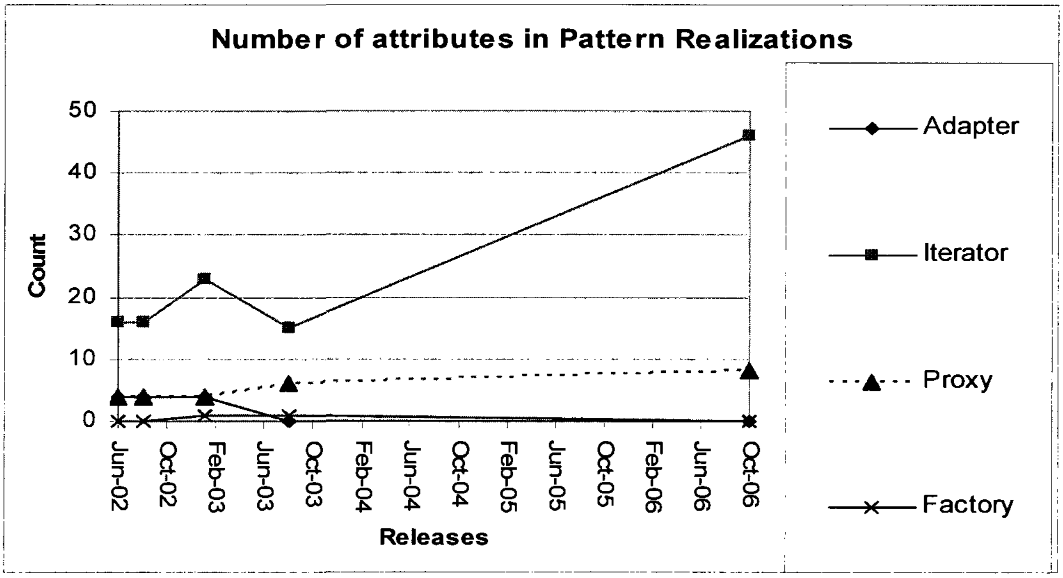


Figure 5.19 eXist number of attributes of all possible realizations of various design patterns

5.2.3 Number of Methods

Similar to the number of attributes observations, we discern a modest growth for the total number of methods in the Adapter and Visitor patterns in the JRefactory system.

In ArgoUML we only observe a sharp increase in the total number of methods for the Factory pattern, and in the eXist system we observe modest increases in the Iterator pattern, and a very sharp increase in the Adapter pattern.

Figure 5.20 and 5.21 display the results for JRefactory. Figure 5.21 is the same as 5.20 minus the numbers for the Visitor pattern. Figure 5.21 helps us visualize the results for patterns with smaller counts.

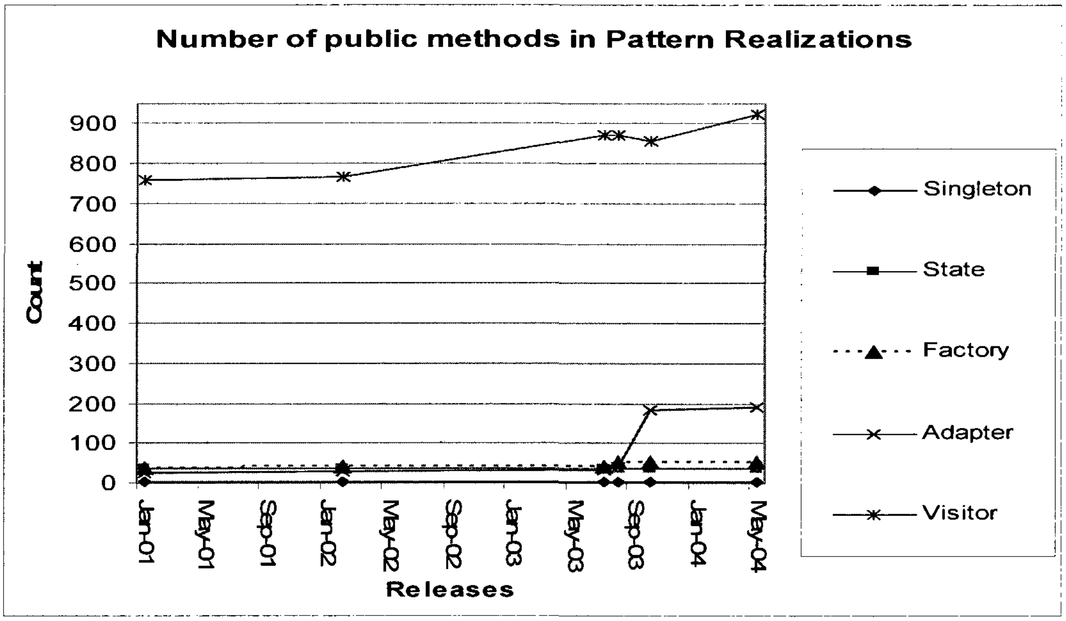


Figure 5.20 JRefactory number of public methods of all possible realizations of various design patterns

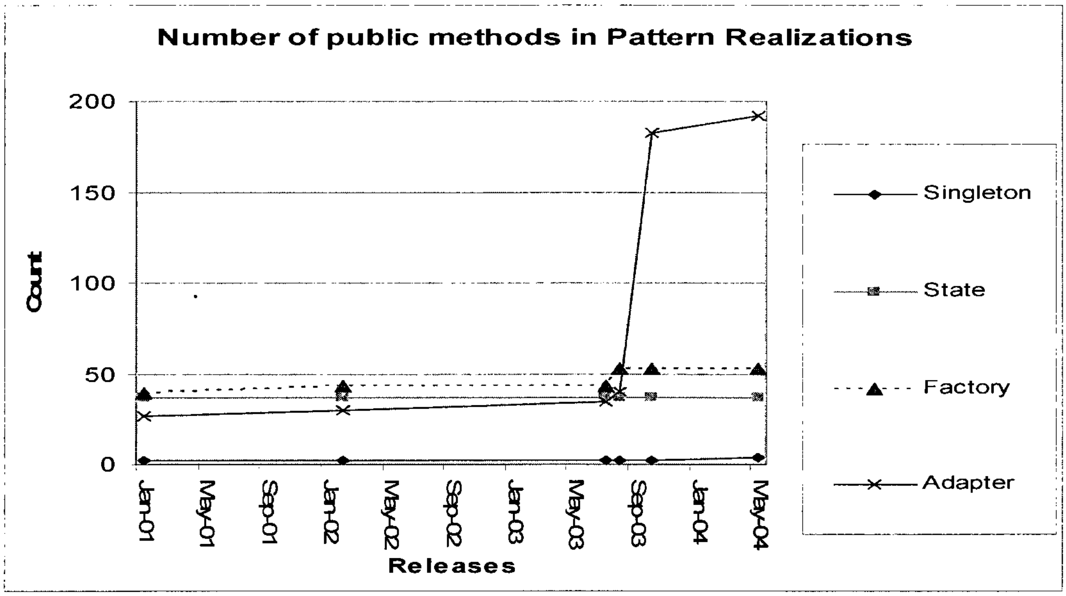


Figure 5.21 JRefactory number of public methods of all possible realizations of various design patterns minus the Visitor pattern

Figures 5.22 and 5.23 show the respective results for the total number of public methods measure of the ArgoUML and eXist systems.

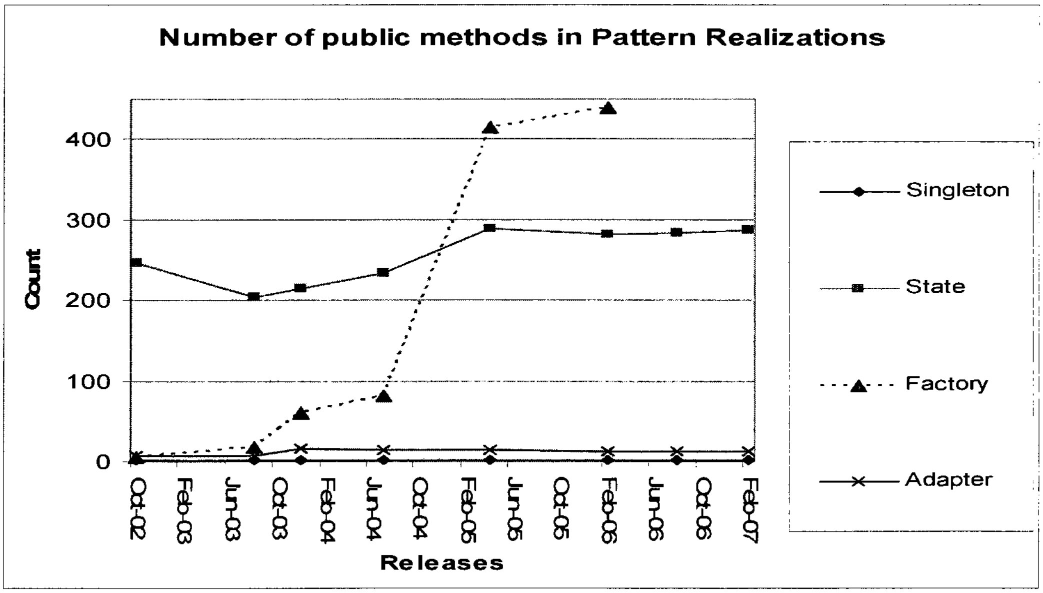


Figure 5.22 ArgoUML number of public methods of all possible realizations of various design patterns

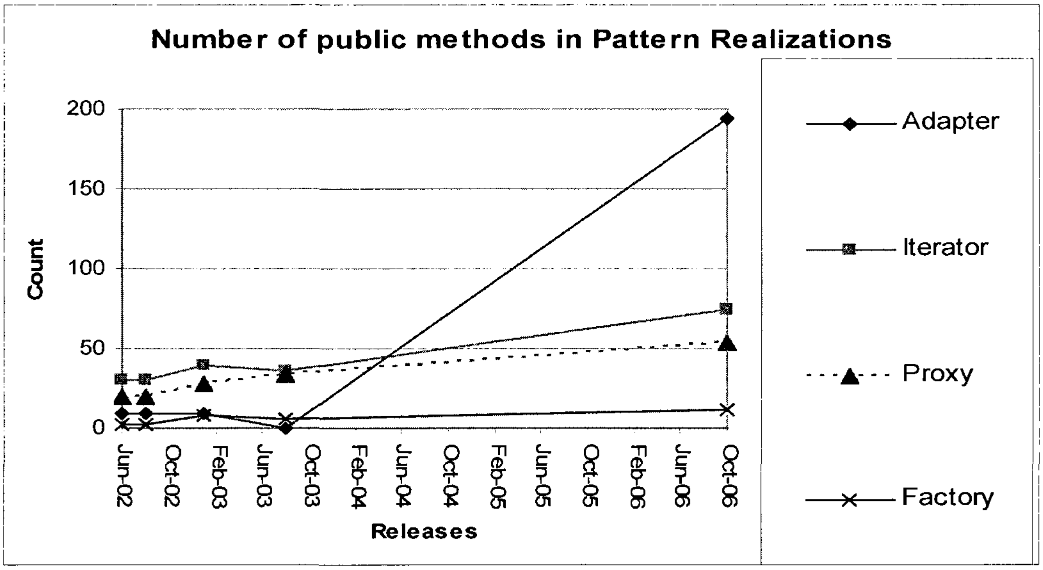


Figure 5.23 eXist number of public methods of all possible realizations of various design patterns

5.3 Organizational Grime Results

Organizational grime gives us a different perspective into the evolution of software systems. We observe 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. In addition to observing sizing measures at the package and file levels, we also measure afferent coupling changes. Afferent coupling is a good indicator of whether the existing package is being referenced and imported by other packages. Efferent coupling measures at a package level were not collected as they do not provide compelling data.

5.3.1 Afferent Coupling Counts at a Package Level

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 is used to gather afferent coupling measurements which operate at this level.

Figure 5.24 displays the afferent coupling of the packages that participate in the implementation of design patterns of JRefactory.

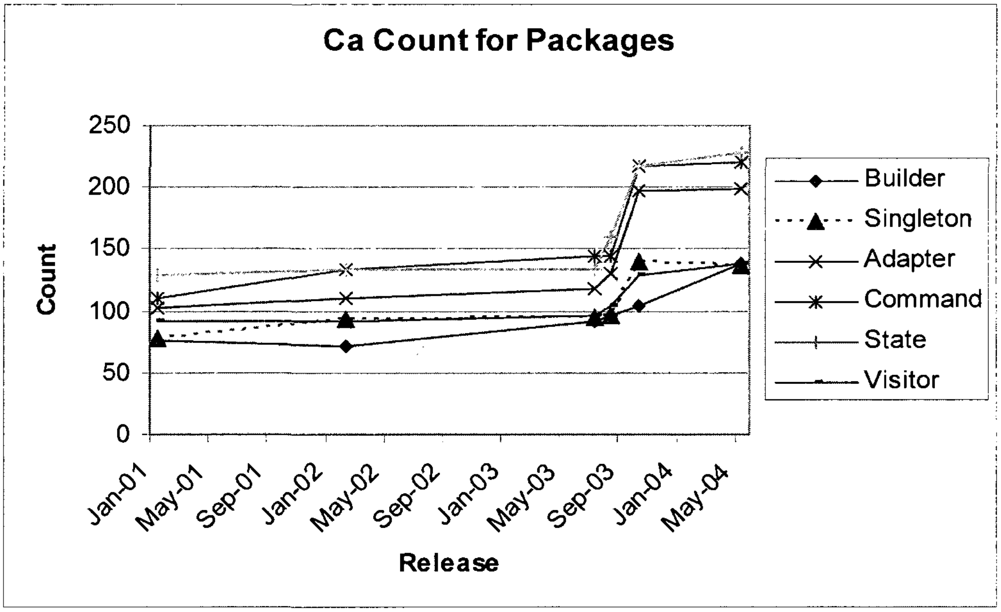


Figure 5.24 Afferent Coupling of packages in JRefactory that contain the realization of design patterns

Figures 5.25 and 5.26 show the results for the afferent coupling of Java packages in ArgoUML. We use two graphs to reduce clutter.

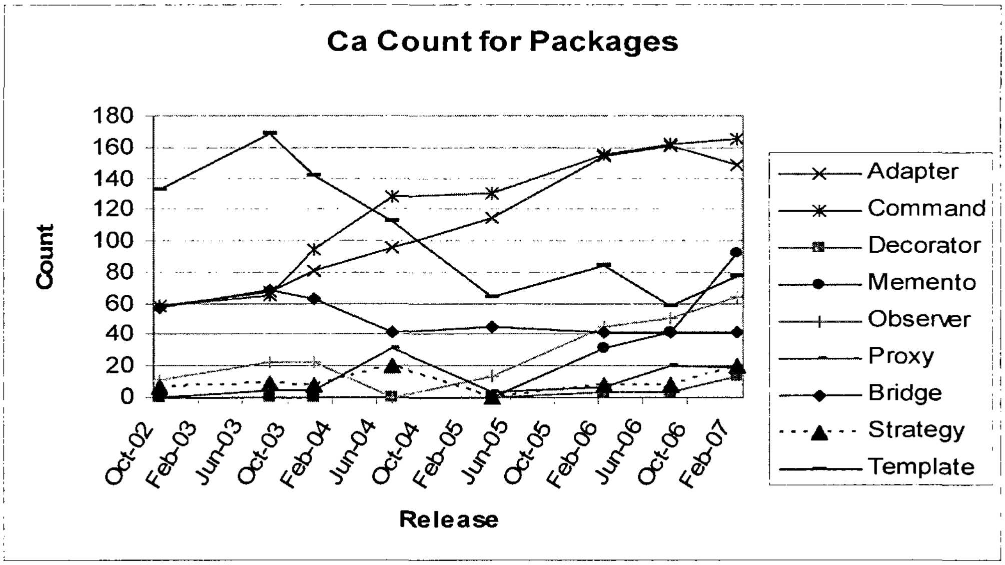


Figure 5.25 Afferent Coupling of packages in ArgoUML that contain the realization of design patterns

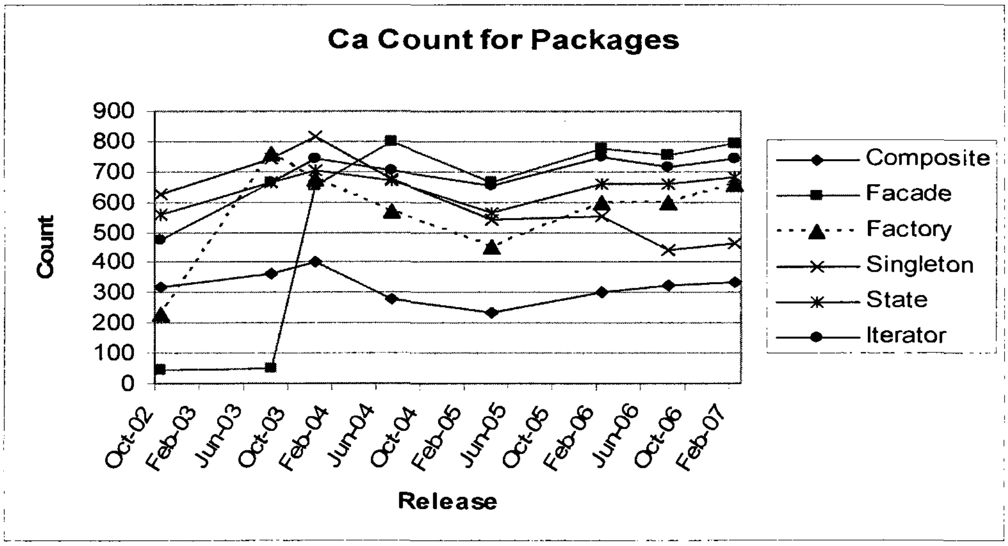


Figure 5.26 Afferent Coupling of packages in ArgoUML that contain the realization of design patterns

Figure 5.27 displays the afferent coupling of the packages that participate in the implementation of design patterns of eXist.

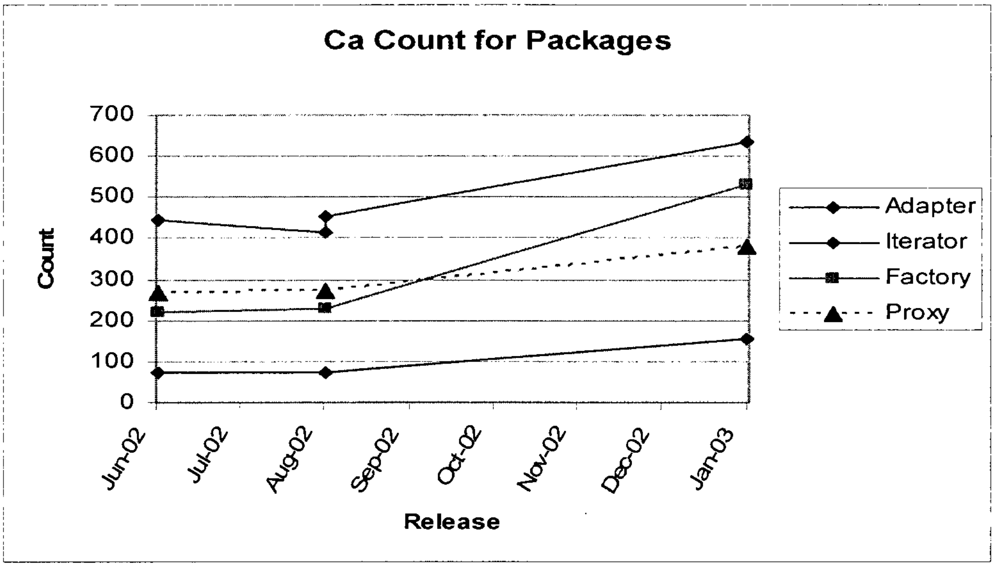


Figure 5.27 Afferent Coupling of packages in eXist that contain the realization of design patterns

5.3.2 Number of Files that Participate in the Implementation

We use *Pattern Seeker* and *Design Pattern Finder* to gather file statistics for various design patterns. The total number of files that participate in the implementation of design pattern realizations exhibit a constant value in the cases of JRefactory and ArgoUML. The State pattern in the latter does show steady growth, however this appears to be an exception. In the case of eXist, we observe a steady growth after August 2003.

Figure 5.28 displays the total number of files that participate in the realization of various design patterns of JRefactory.

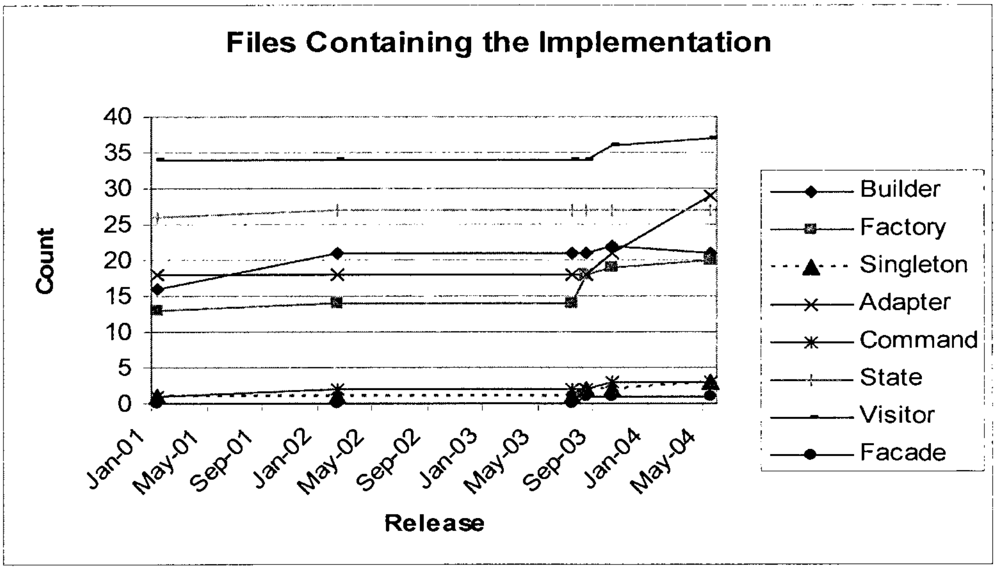


Figure 5.28 JRefactory number of files participating in the realization of design patterns

Figures 5.29 and 5.30 show the results for the total number of files measure of ArgoUML. Figure 5.30 is the same as 5.29 minus the numbers for the State pattern. Figure 5.30 helps us visualize the results for patterns with smaller counts.

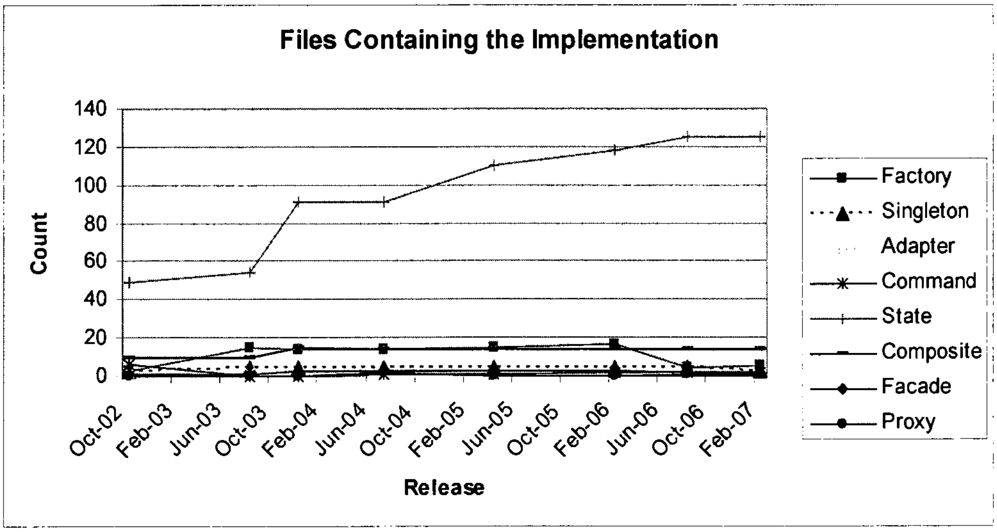


Figure 5.29 ArgoUML number of files participating in the realization of design patterns

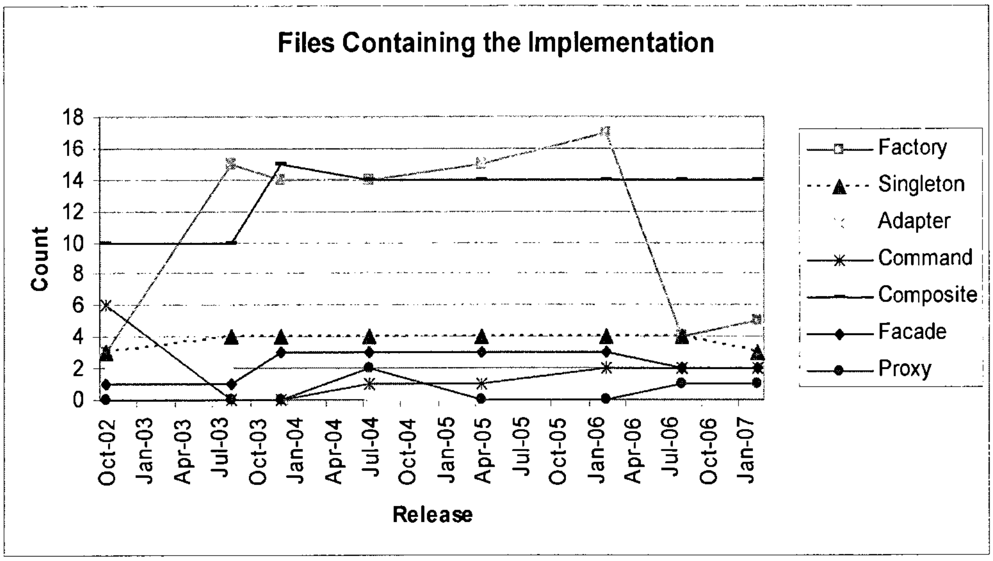


Figure 5.30 ArgoUML number of files participating in the realization of design patterns minus the State pattern

Figure 5.31 displays the total number of files that participate in the realization of various design patterns of eXist.

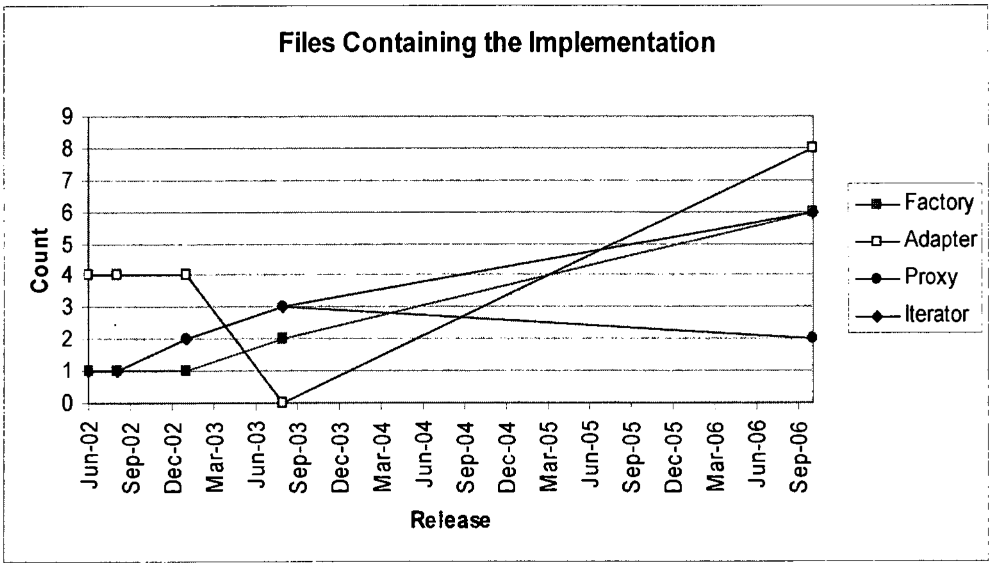


Figure 5.31 eXist number of files participating in the realization of design patterns

5.3.3 Number of Java Packages that Participate in the Implementation

At a package level we can get a different perspective regarding the evolution and grime buildup of design pattern realizations. Rather than focusing on files, we focus on Java packages that participate in realizations of design patterns. Measures at a package level are considered organizational.

The total number of packages that participate in the implementation of various design pattern realizations remain very steady for ArgoUML. In the case of eXist, the number of Java packages shows a tendency to grow over time, and in the case of JRefactory we observe steady numbers until the July 2003 release when a marked increase is observed until October 2003. The total number of packages settles again into a constant value after such date.

Figure 5.32 displays the total number of packages that participate in the realization of various design patterns of JRefactory.

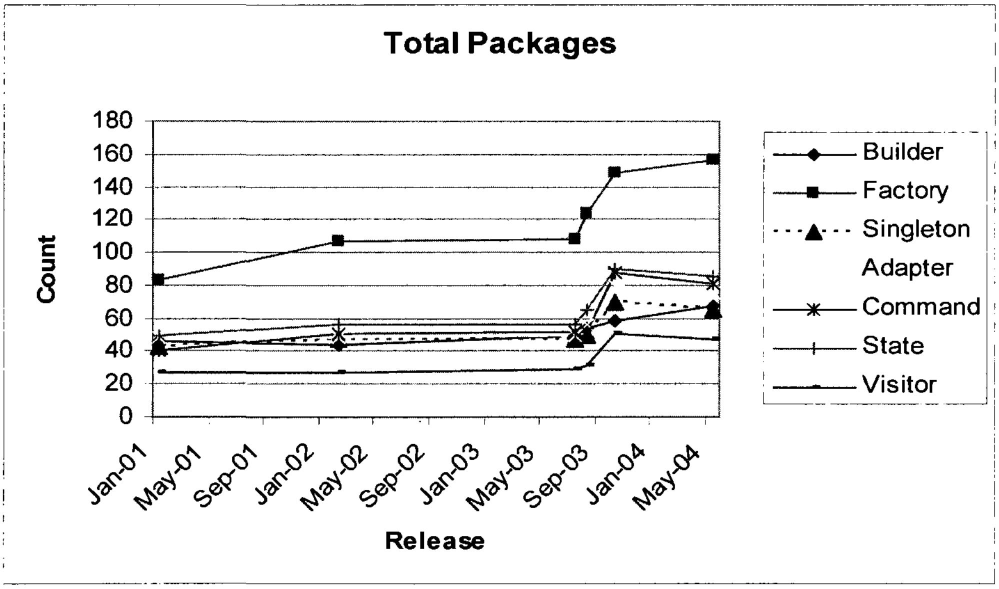


Figure 5.32 JRefactory number of packages participating in the realization of design patterns

Figures 5.33 and 5.34 show the results for the total number of Java packages measure of ArgoUML. We use two graphs to reduce clutter.

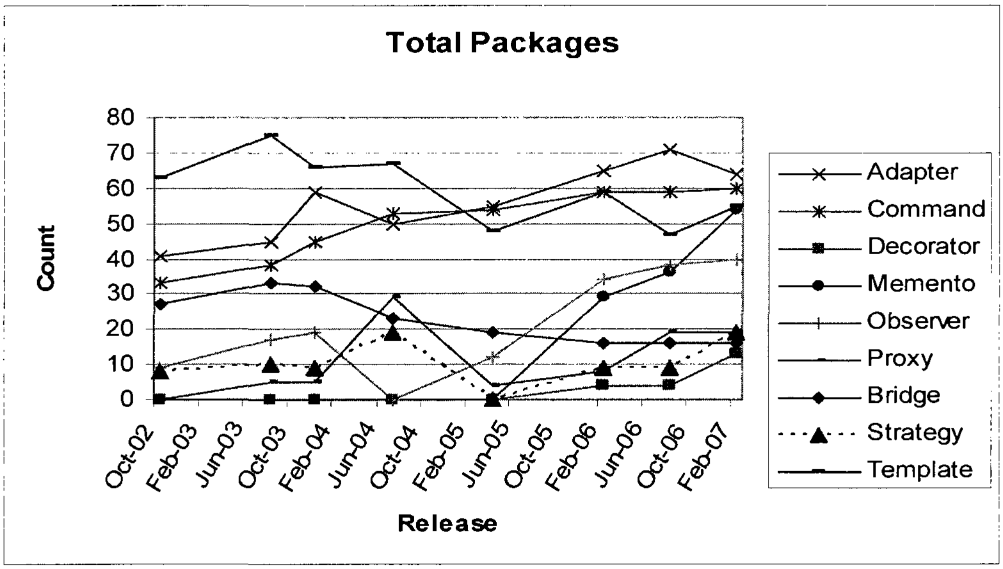


Figure 5.33 ArgoUML number of packages participating in the realization of design patterns

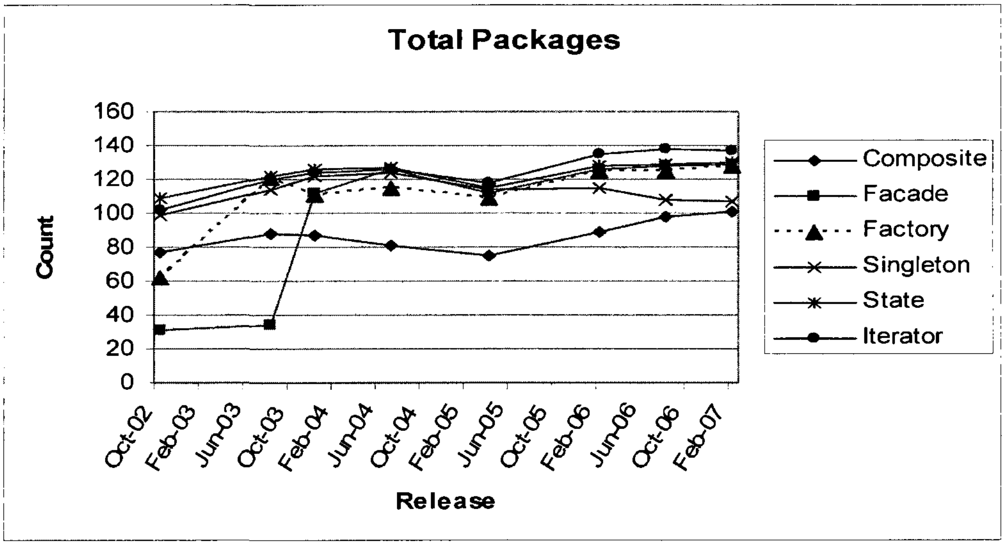


Figure 5.34 ArgoUML number of packages participating in the realization of design patterns

Figure 5.35 displays the total number of packages that participate in the realization of various design patterns of eXist.

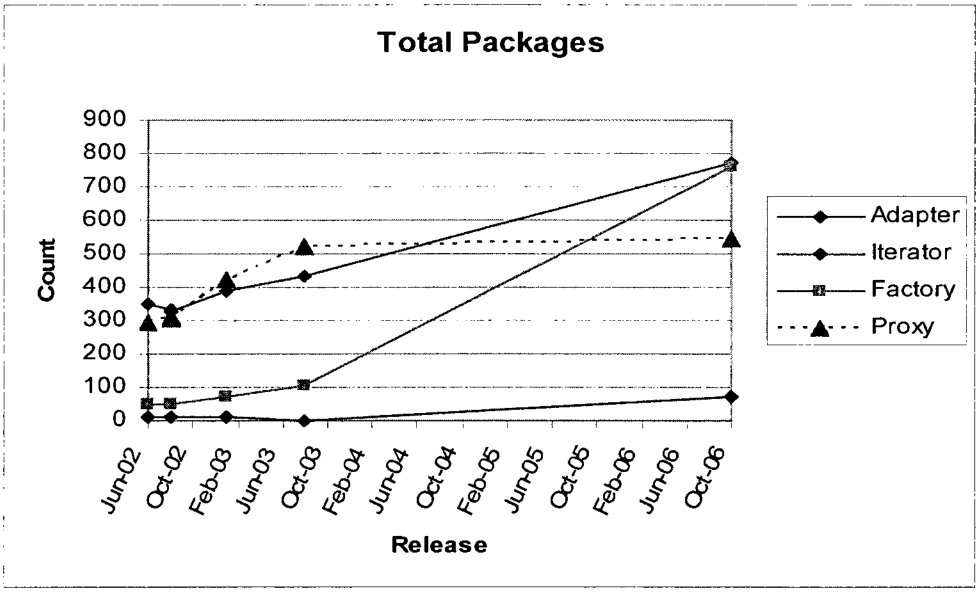


Figure 5.35 eXist number of packages participating in the realization of design patterns

6. ANALYSIS AND DISCUSSION

Finding evidence of grime buildup in widely used systems is important. Each system studied has a significant number of downloads as reported by SourceForge [81]. This case study observed grime buildup in eXist, a widely used database system (almost 250K downloads), and two applications; also widely used in their respective domains, JRefactory and ArgoUML (85K and 16K downloads respectively). We analyze the raw results described in section five and augment the data with additional derived measures. In all cases we summarize our observations and provide explanations for discernible grime buildup exhibited by pattern realizations in the systems studied. There are thousands of object oriented systems potentially available for study, and increasing the sample size is imperative to making stronger generalizations.

We also investigate the consequences that the observed grime buildup has on the testability and the adaptability of design patterns. Consequences are described in sections 6.4 and 6.5. We provide evidence and statistical analysis to support claims. Finally, we evaluate the set of hypotheses posed in section 6.6.

6.1 Modular Grime

All systems studied show that there are increases in modular grime for almost every design pattern studied. Table 6.1 lists all the design patterns that were carefully studied in each of the software systems. Each design pattern is given a nominal evaluation that describes the growth of afferent coupling. A *Negative* evaluation means that afferent coupling tends to decrease as the pattern evolves. A *Flat* evaluation indicates that afferent coupling remains steady, and a *Positive* evaluation indicates growth in afferent coupling as the system evolves.

Table 6.1 Observed growth of afferent coupling

	JRefactory	**ArgoUML**	**eXist**
Singleton	Positive	Flat	
State	Flat	Positive	
Factory	Positive	Flat	Flat
Adapter	Positive	Flat	Positive
Visitor	Positive		
Iterator			Positive
Proxy			Positive

Afferent coupling in all studied systems remains flat or exhibit a tendency to grow. Clearly additional patterns need to be observed. More importantly, the systems studied are deemed successful and this may be due in part, to their ability to keep afferent coupling counts in check. Studies by Basili et al. [5] and Briand et al. [17] provide evidence to support that coupling measures are useful indicators of quality. The results are the same after normalizing afferent coupling counts by the number of realizations of design patterns in a system.

Similarly, table 6.2 lists all the design patterns in each of the software systems with their corresponding efferent coupling nominal classification.

Table 6.2 Observed growth of efferent coupling

	JRefactory	**ArgoUML**	**eXist**
Singleton	Positive	Flat	
State	Flat	Flat	
Factory	Positive	Positive	Positive
Adapter	Positive	Positive	Positive
Visitor	Positive		
Iterator			Positive
Proxy			Positive

The values for the different categories are similar. This is an indication that as design patterns evolve their couplings tend to increase. In our statistical evaluations of the consequences that grime buildup has on adaptability we further explore the degree to which afferent and efferent coupling measures are related to changes in lines of code.

6.2 Class Grime

When design patterns evolve, we expect to find that the total number of classes that make up a pattern grow in cases of dynamic patterns, and remain stable in cases of static patterns such as the Singleton pattern. Dynamic patterns are meant to be extended via realizations of interfaces or generalizations of abstract classes. This is in keeping with the notion that design patterns are extended through new concrete classes. Thus, we should expect the average number of methods and the average LOC will grow proportional to the number of classes and to each other. Grime buildup is indicated by an increase in the number of methods without an increase in the number of classes. We also expect that the number of attributes of patterns should remain constant as design patterns are not meant to be extended via state values.

6.2.1 Class Sizes, Number of Attributes, and Number of Methods

The results presented in chapter five shows that in all systems studied, we see growth in the total number of lines of code in the classes that participate in design patterns. We also see little growth in the number of methods, and almost no growth in the total number of attributes. In order to better understand how classes that participate in the realizations of design patterns are evolving, we also compute the *Average LOC per class* [73], the *average number of attributes* per class, and the *average number of methods* per class. Increases in the average LOC per class are indicative of class bloating and possible buildup of grime.

6.2.1.1 JRefactory

It is clear that the State and Singleton patterns exhibit no average increases in LOC. Only the Adapter and Factory pattern realizations exhibit significant growth after the August 2003 release. Figures 6.1 through 6.3 display the observed values.

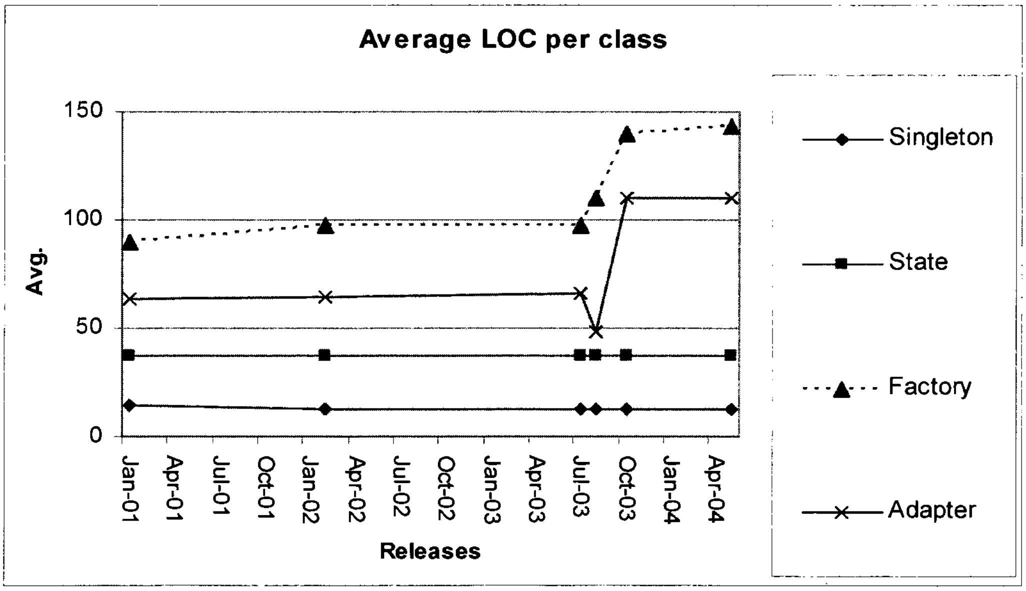


Figure 6.1 JRefactory average LOC per participant class

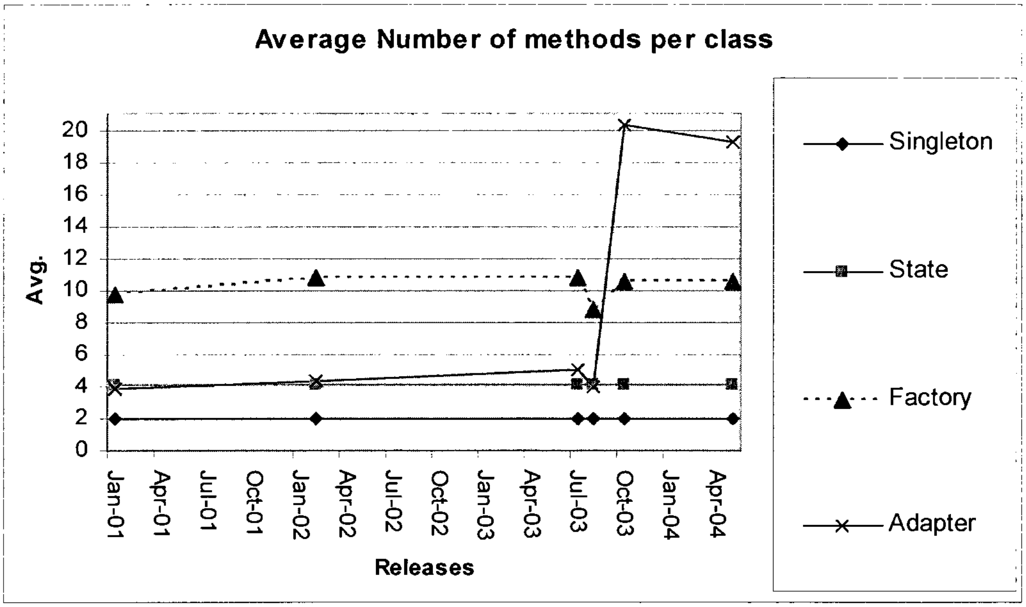


Figure 6.2 JRefactory average number of methods per participant class

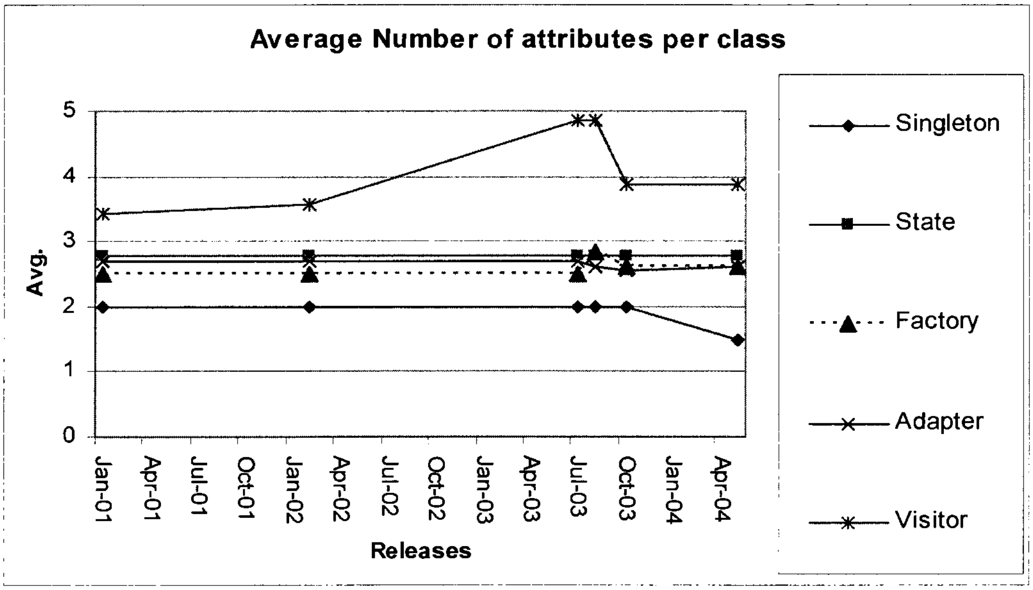


Figure 6.3 JRefactory average number of attributes per participant class

Clearly there is no evidence to indicate that the average number of methods or attributes is increasing in a manner that is not consistent with the expected evolution of design patterns. The Adapter pattern does experience a sudden growth of methods in

October 2003; however this is matched by the average LOC. We observe that only the Factory pattern is experiencing greater growth in the average LOC than that of the average number of methods.

The Visitor pattern has much higher average number of methods and LOC averages. Figures 6.4 and 6.5 display the two graphs and how they relate to other patterns studied in JRefactory. Clearly growth in the average LOC and the average number of methods exhibit similar trends, thus no evidence of class grime can be inferred. Also, close inspection of the code shows that the growth in the number of methods in JRefactory's Visitor pattern realizations is consistent with the RBML of the Visitor pattern. The increase in methods is mainly due to conformant *visit()* and *accept()* pair methods.

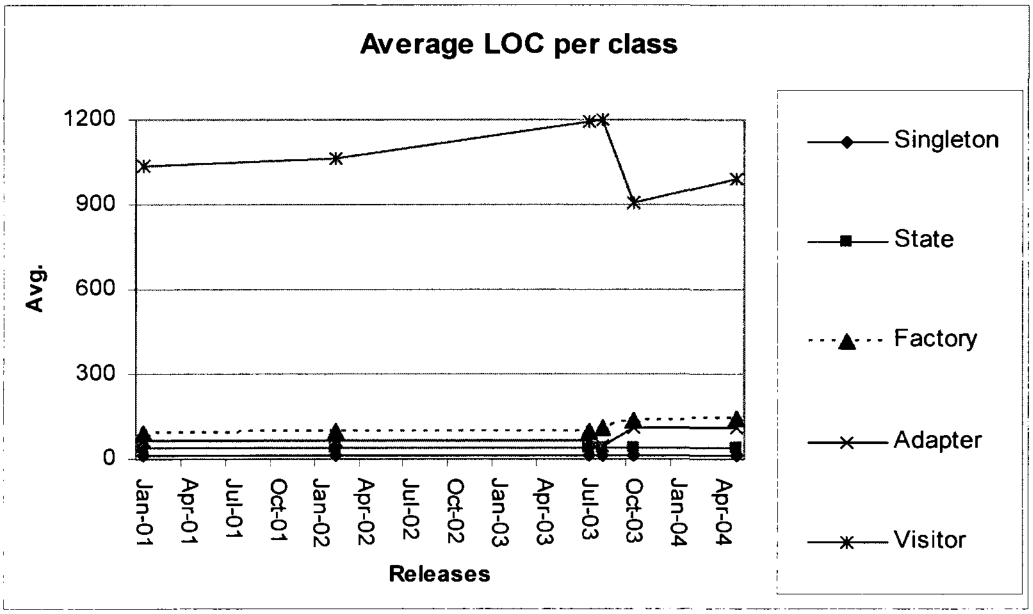


Figure 6.4 JRefactory's Visitor pattern average LOC per participant class

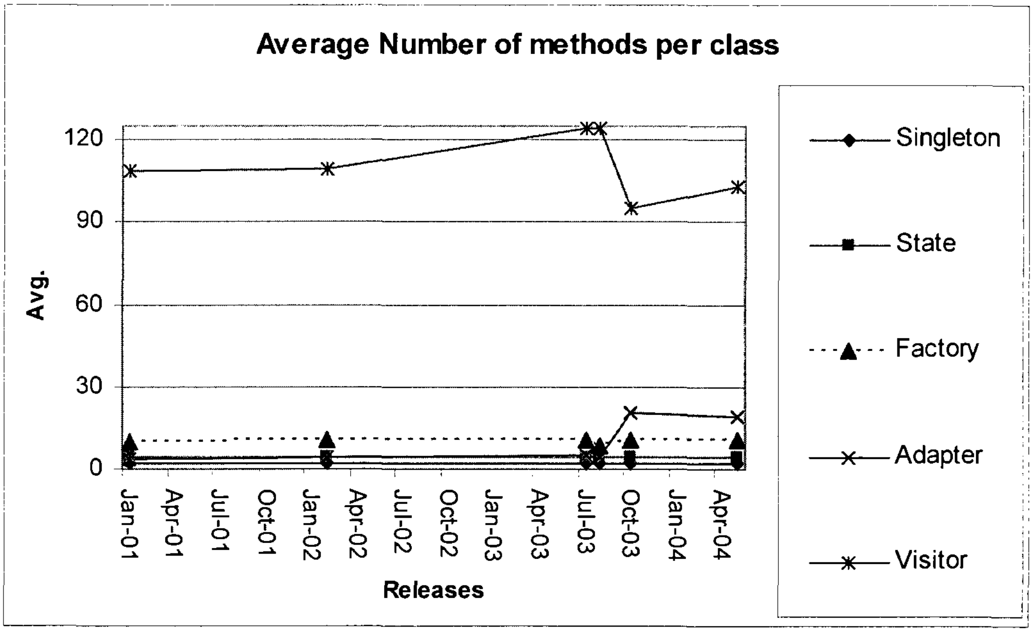


Figure 6.5 JRefactory's Visitor pattern average number of methods per participant class

6.2.1.2 ArgoUML

The analysis of ArgoUML also yields little evidence of class grime. Figures 6.6 through 6.8 show the averages computed for ArgoUML.

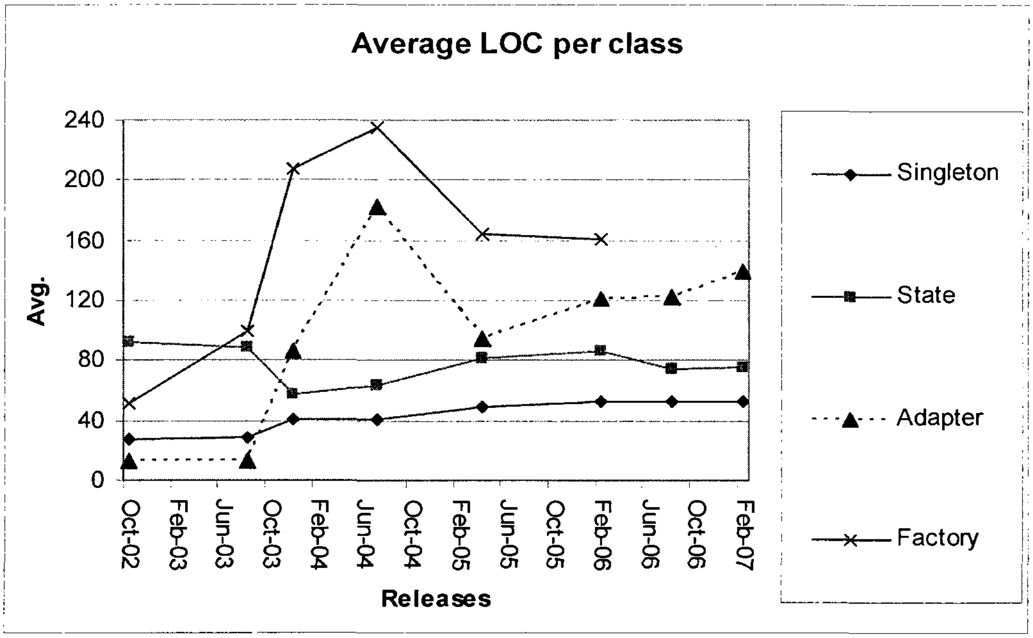


Figure 6.6 ArgoUML average LOC per participant class

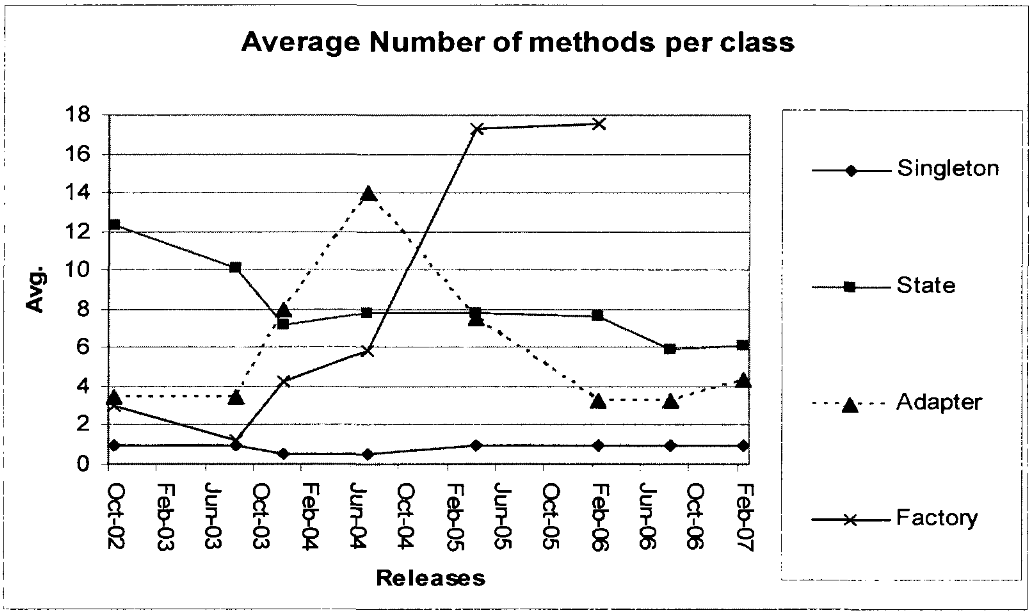


Figure 6.7 ArgoUML average number of methods and per participant class

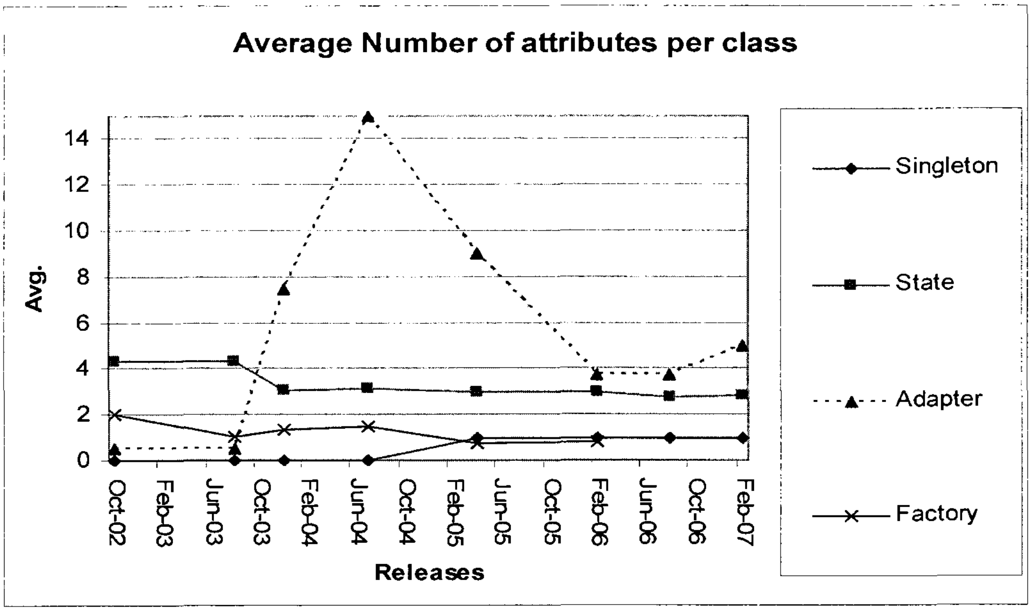


Figure 6.8 ArgoUML average number of attributes per participant class

The Adapter and Factory patterns exhibit high growth counts in average LOCs until the July 2004 release. The averages are quickly brought down after such release. Although grime buildup was beginning to form, it appears that a refactoring process took place. This conjecture about refactoring is supported by observing the average number of methods. The Factory pattern sees a significant increase in the average number of methods through all studied releases. The increase is matched by the average increase in LOC until July 2004. After this release, the average LOC begins to decrease while the average number of methods increases, which would point to possible refactoring.

The State pattern shows a slight decrease in the average number of methods while the average LOC tends to remain constant which also indicates a form of class bloating. The Adapter pattern exhibits a spike in growth in the average number of methods and attributes in the July 2004 release, however this is also matched by a similar trend in the average LOC.

Only the State pattern in ArgoUML is experiencing growth in the average LOC as compared to the average number of methods because the average number of methods shows a steady decrease.

6.2.1.3 eXist

Figures 6.9 through 6.11 show the averages computed for eXist.

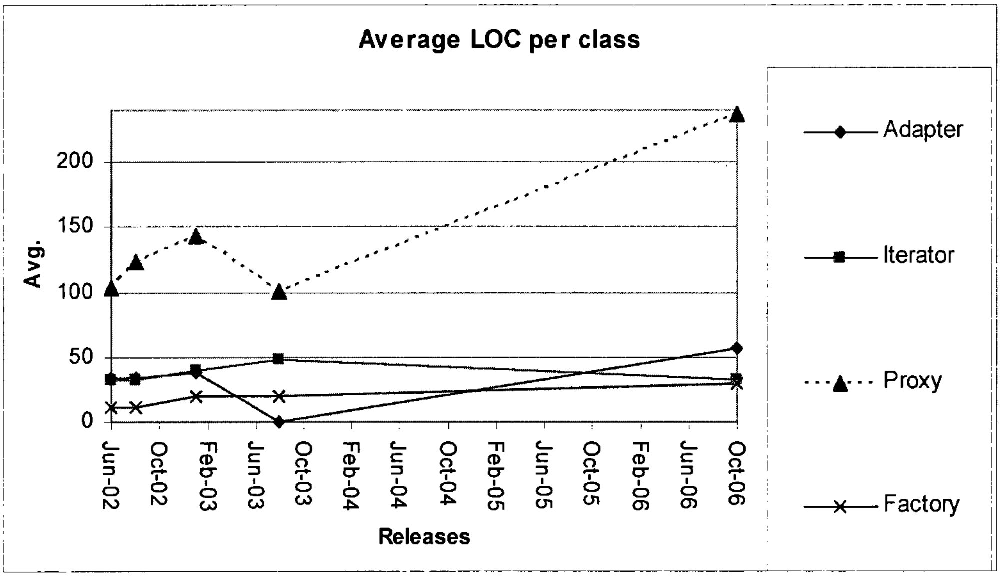


Figure 6.9 eXist average LOC per participant class

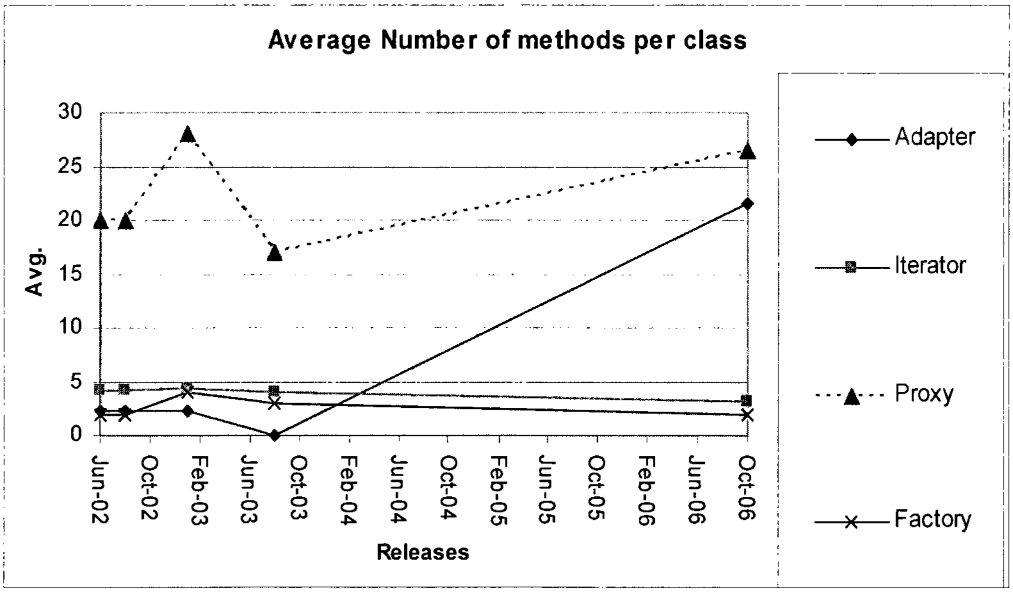


Figure 6.10 eXist average number of methods and per participant class

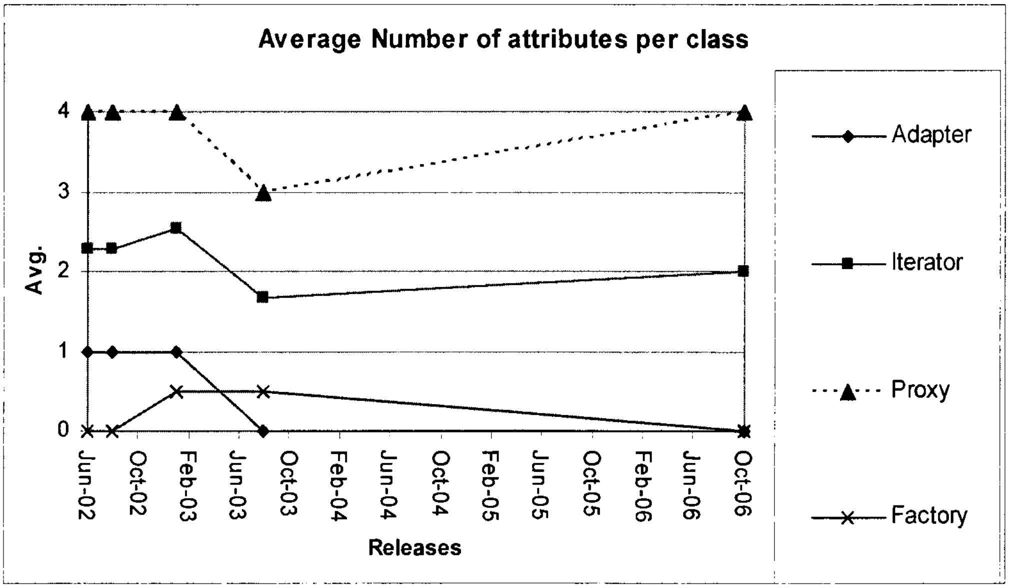


Figure 6.11 eXist average number of attributes per participant class

In eXist we see a general growth trend in average LOC for the Proxy, Adapter, and to a lesser extent the Factory pattern. The Iterator pattern sees a slight decrease after the August 2003 release. We observe an increase in the average number of methods after the August 2003 release. The increases are only significant in the Adapter and Proxy patterns; however these increases are also matched by average LOC increases. These observations indicate healthy and measured growth for these patterns. In the eXist case, we observe no evidence to indicate class bloating for any participants of any class studied.

6.3 Organizational Grime

The afferent levels for all packages that participate in the implementation of design pattern realizations tend to remain constant, with only one exception: JRefactory after the August 2003 release. The relative adaptability of the packages that participate in design patterns does not appear to be affected by afferent coupling measured at this level.

In Java, new classes are implemented in their own files. Thus, when a design pattern is extended in the originally intended way, we should see an increase in the total number of files participating in the realization of the pattern. However this is not the case in many of the realizations studied. We find that classes participating in design patterns are evolving as shown by the modest growth in total number of methods, and to a larger extent in lines of code. This would imply that design patterns become bloated, thus contributing to organizational grime. Similar trends occur in packages participating in design patterns. At an organizational level, there is not enough evidence to support grime buildup.

6.4 Testability Consequences of Grime

The minimum number of test requirements necessary to provide adequate test coverage is computed, and the formation of anti-patterns is observed. Test requirements increased and anti-patterns developed as a result of grime buildup in systems studied. Section 4.4.1.2 provides detailed explanations of the derivations of the equations to compute test requirements, and details of the anti-patterns that are observed.

We examine the effects of grime buildup on the testability of the JRefactory open source system. We studied versions 2.6.12, 2.6.38, 2.7.05, 2.8.00, 2.9.00, and 2.9.19. These releases represent the evolution of the software over a period of almost four years.

6.4.1 Observed Effects on Test Requirements

We evaluate the consequences of modular grime buildup on the adequacy of test requirements by counting the number of tests necessary to provide adequate coverage. First we analyze the impact that associations have on test requirements. Figure 6.12, and 6.13, display the corresponding values for Visitor and Singleton design patterns in JRefactory. The equation for computing the number of tests for associations in JRefactory is given by equation I, $A(n, k) = 8nk + c, c>=0$. The x-axis values represent equally spaced intervals for the various releases of the software. We used CurveExpert [23] statistical software to create our graphs. Although the Singleton realization yielded slightly different results than the Visitor pattern, both results are monotonically increasing.

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

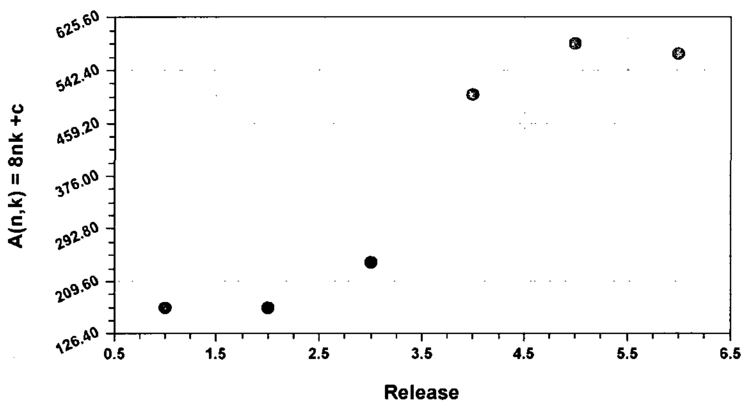


Figure 6.12 Test requirement count for associations in the Visitor pattern of JRefactory

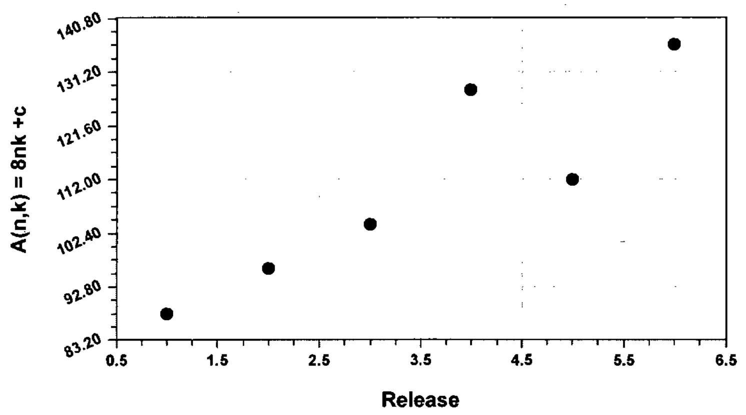


Figure 6.13 Test requirement count for associations in the Singleton pattern of JRefactory

For dependencies (equation V), where D(n, k) = c, $c>=0$, we obtain the results shown in figure 6.14 for the Visitor pattern.

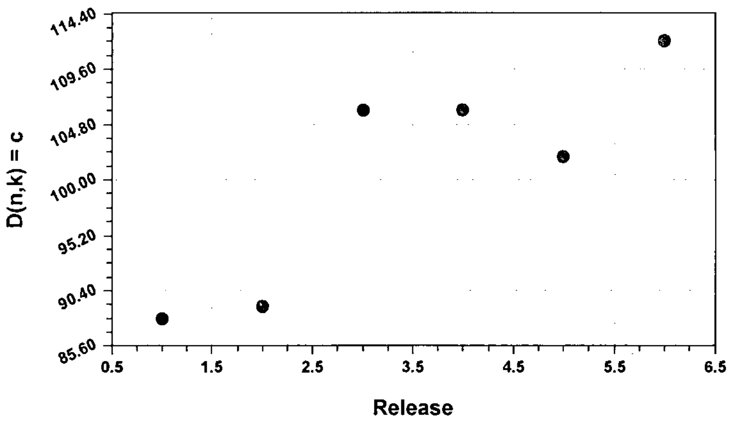


Figure 6.14 Test requirement count for dependencies in the Visitor pattern of JRefactory

The Singleton instance yielded the values displayed in figure 6.15. Generalization consequences are defined by equation IV; G*(n, k)* = 2*nk.* There are no generalizations in the evolution of the Visitor realization studied. However, in the case of the Singleton pattern we found data as shown in figure 6.16.

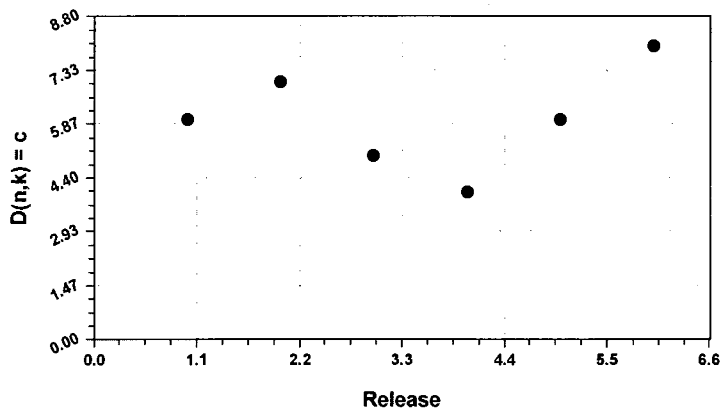


Figure 6.15 Test requirements count for dependencies in the Singleton pattern of JRefactory

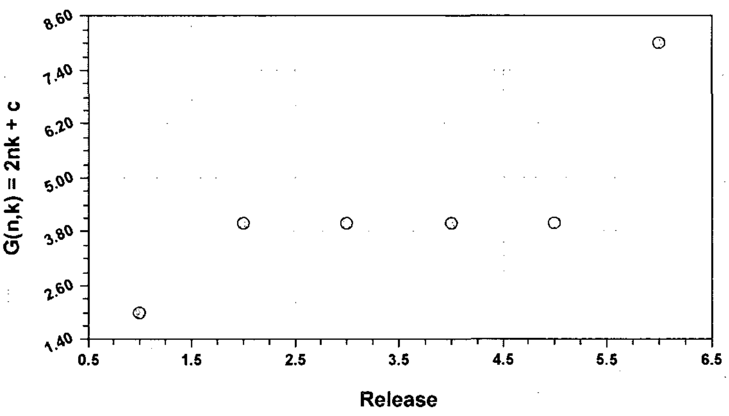


Figure 6.16 Test requirements count for generalization in the Singleton pattern of JRefactory

In general we found that realizations of the Visitor and Singleton patterns show growth in the number of test cases necessary to test new grime buildup.

6.4.2 Observed Appearances of Test Anti-patterns

The following examples demonstrate the formation of testing anti-patterns in JRefactory.

In the first example we observe the evolution of an inheritance hierarchy in a realization of the Visitor pattern. The new hierarchy formed approximately two years after the first release of JRefactory. In this example the gray arrows represent the inheritance hierarchies, the black arrows are associations, and the dashed line represents a use relationship. Figure 6.17 illustrates an example of the *self-use-relationship* anti-pattern.

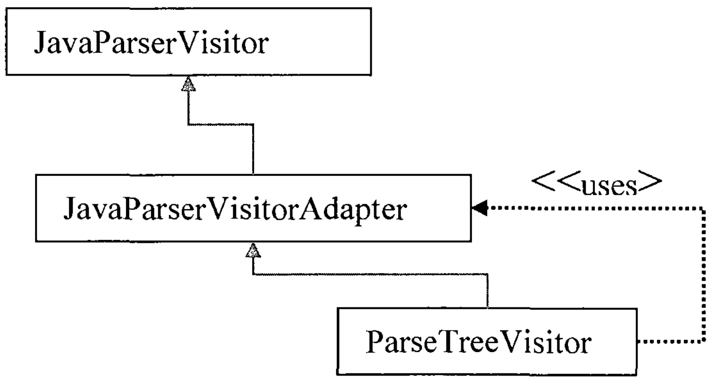


Figure 6.17 Self-Use-Relationship anti-pattern in JRefactory

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

We also find evidence of the formation of anti-patterns described by Brown et al. [19]. Three realizations of the State pattern were studied with no evidence of evolution found. The State pattern never evolves. This is an example of dead code.

In another example, we find evidence of the *swiss-army-knife* anti-pattern. The original design pattern was not intended to implement the methods defined by a new interface. Figure 6.18 illustrates the example found in the JRefactory system.

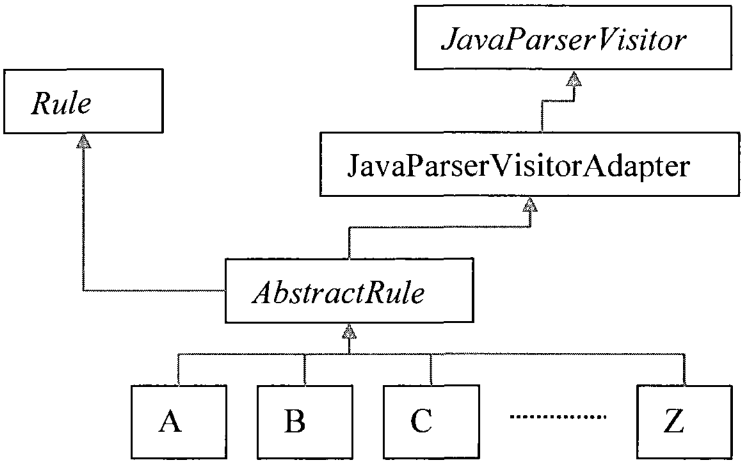


Figure 6.18 Swiss army knife anti-pattern in JRefactory

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

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

Figure 6.19 illustrates the earliest version of a *concurrent-use-relationship* anti-pattern found in all versions of a realization of the Visitor pattern in JRefactory. Clearly, the "summary" hierarchy of classes can be accessed through concurrent paths. A client of class MoveMethodRefactoring can reach various "summary" classes via two paths.

The concurrent access to the "summary" hierarchy is worsened by the inheritance hierarchies involved in both paths because polymorphism must be taken into account when testing. When an instance of the class MoveMethodRefactoring uses an instance of the MoveMethodVisitor class, then it must consider objects of type ChildrenVisitor as well. Baudry et al. [8] find that *"the Visitor pattern is especially known to be difficult to test because of an extensive use of polymorphism."* They provide a testability grid for design patterns that considers the number of paths and self usages to test as a result of anti-patterns.

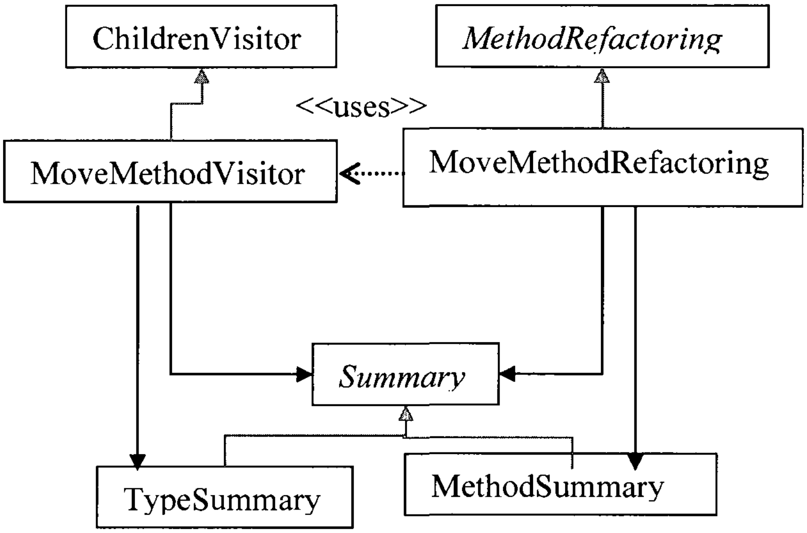


Figure 6.19 Concurrent-Use-Relationship anti-pattern in JRefactory

The formation of anti-patterns as a consequence of grime buildup is likely to be pervasive. To evaluate this, other open source systems are under investigation and early evidence suggests similar results.

6.5 Adaptability Consequences of Grime

A multiple linear regression model is used to understand how changes in LOC of participating pattern classes are related to modular grime. We also use the Spearman correlation model to obtain correlation coefficients. 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. Section 6.5.1 explores adaptability further. In section 6.5.2, *Instability* is measured for various design patterns and the values indicate that the Instability measure, calculated as the ratio of efferent to total coupling, tends to approach one. High values are indicative of young design patterns. Finally, in section 6.5.3 we discuss why *Abstractness* is not an appropriate measure to predict adaptability.

6.5.1 Relationship between changes in LOC and grime

We use Spearman's correlation because our samples are small and we need a non-parametric technique. Table 6.3 suggests mixed results, with significant values highlighted in bold. The coefficients for coupling (afferent and efferent) show that for JRefactory and Exist there is a correlation with coupling, however this is not evident in ArgoUML. According to Ott and Longnecker [70], anything greater than six tenths implies that there is correlation.

Table 6.3 Spearman correlation coefficients for afferent(C_a) and efferent(C_e) coupling vs. changes in LOC with corresponding p-values in parentheses

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

For each system under study, multiple linear regression using independent variables C_a (afferent coupling) and C_e (efferent coupling) is performed. The regression is performed to see if there exists a structural relationship between ΔLOC and afferent/efferent coupling.

Table 6.4 shows the results of the multi-linear regression results for all systems under study. The *coupling model* takes into account afferent and efferent coupling as independent variables and regresses them against the dependent variable ΔLOC. Significant values are displayed in bold.

Table 6.4 The coupling model. *p-values* of a multi-linear regression for afferent(C_a) and efferent(C_e) coupling combined vs. changes in LOC

JRefactory			
Singleton	Factory	Adapter	Visitor
0.0004	0.3724	0.1659	**0.1126**
Exist			
Iterator	Factory	Adapter	Proxy
0.0059	**0.0007**	**0.0038**	**0.0006**
ArgoUML			
Singleton	Factory	Adapter	State
0.6118	0.9540	0.6745	0.4731

A *p-value* represents "the probability of observing a value of the test statistic more supportive, or as supportive, of H_a than the observed value" [21]. The *p-value* tells us whether we would reject a hypothesis for a specified α. The test statistic used in the multiple regression analysis is the F-statistic and is evaluated by the rule probability > F (the *p-value*). The underlying null hypothesis says that the independent variables have no predictive power, and the alternative hypothesis says that the independent variables do have predictive power over the dependent variable (a two sided test). The regression is not used to describe the predictive power that independent variables may have over the dependent variable, but merely to establish that a regression line that minimizes the horizontal distance to the average ΔLOC exists. The smaller the probability calculated (*p-value*), the more significant the relationship between coupling and ΔLOC.

Considering the small number of data sets for all systems under study, both correlation and regression values are surprisingly high. Based on these results, it is clear that a strong structural relationship exists in eXist, and to a lesser extent in JRefactory; however we cannot make such conclusions for ArgoUML. JRefactory's Singleton pattern changes in LOC can be predicted with $\alpha = 0.01$ accuracy and correlates well with coupling. The Adapter and Factory patterns are not statistically significant and the Visitor pattern's changes in LOC have $\alpha = 0.1$ accuracy and a significant correlation value to efferent coupling only. In the case of eXist, we have very strong evidence that changes in LOC are structurally related to coupling for all patterns studied with $\alpha = 0.01$ accuracy. Additionally all correlation values are significant. ArgoUML however was different, and we have no evidence to reject the null hypothesis. Note that we use ninety percent confidence intervals to evaluate all systems. While ninety-five percent would be more significant, ninety percent is acceptable in small statistical samples.

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 are necessarily grimy. Raw data used for calculating these statistics can be found in appendix C.

6.5.2 Instability

As discussed in section 4.4.2.2, instability aims to predict how hard it will be to change a software artifact. It has a value between zero and one, and is based on afferent and efferent coupling measurements. We apply this measure to realizations of design patterns by aggregating coupling counts of all participant classes in the design pattern. The value is a ratio of its total fan out to the total number of relationships.

$$\text{Instability} = Ce / (Ca + Ce)$$

The intuition behind the instability measure is that it helps us determine weather a given pattern realization is hard to modify and adapt. A high afferent coupling compared to efferent coupling describes a pattern whose adaptability is limited. This is referred to as a stable pattern because once established; the pattern tends to remain in place and experiences little change. High efferent coupling compared to afferent coupling point to young designs. In the case of pattern realizations it describes young patterns that are not getting too much use, yet their dependencies on external classes is high. As a result we have brittle pattern instances that have high testability requirements.

The Adapter pattern consistently exhibits high instability values throughout its lifecycle and across all systems. In ArgoUML, we observe the instability values drop between December 2003 and July 2004; coincident with refactoring efforts before once again settling above eight tenths. We also observe a value of zero in the realization of this pattern in the August 2003 release of eXist before climbing to nine tenths. The latter is also due to refactoring efforts. The Factory pattern is also found in all systems studied and its instability values, although not as high as the Adapter pattern, are also consistently high.

In JRefactory we observe low instability values for the realizations of the Singleton pattern. Once in place, its efferent coupling does not change, yet its afferent coupling is much more likely to grow. This observation however does not appear true for the same pattern in ArgoUML. An instability value of one indicates that the pattern is not being used, thus it is easy to remove from the overall design. After the April 2005 release, this pattern begins to see usage; thus lowering its instability value.

The State pattern in JRefactory has an unchanging, yet high instability value. Recall that the State pattern in this system never evolves, thus never changing its afferent or efferent value. ArgoUML's realizations maintain a high instability value.

Finally, JRefactory's Visitor and eXist's Iterator patterns also expose high instability values compared to the realizations of eXist's Proxy pattern which exhibits a moderate and balanced ratio.

Figures 6.20, 6.21, and 6.22 show the respective results for the instability measures of JRefactory, ArgoUML and eXist systems.

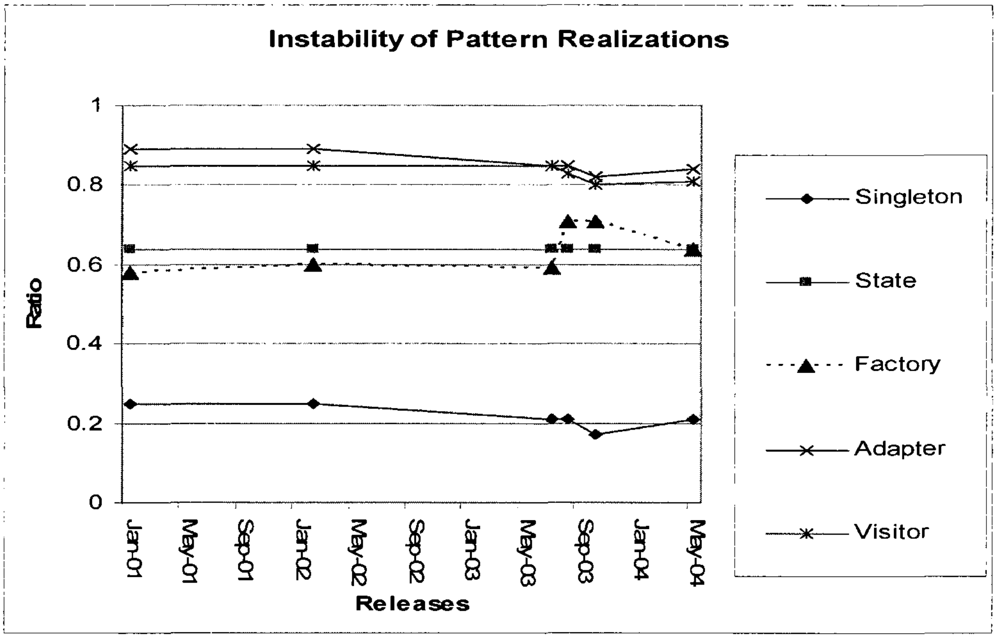


Figure 6.20 Calculated value of Instability of all possible realizations of various design patterns in JRefactory

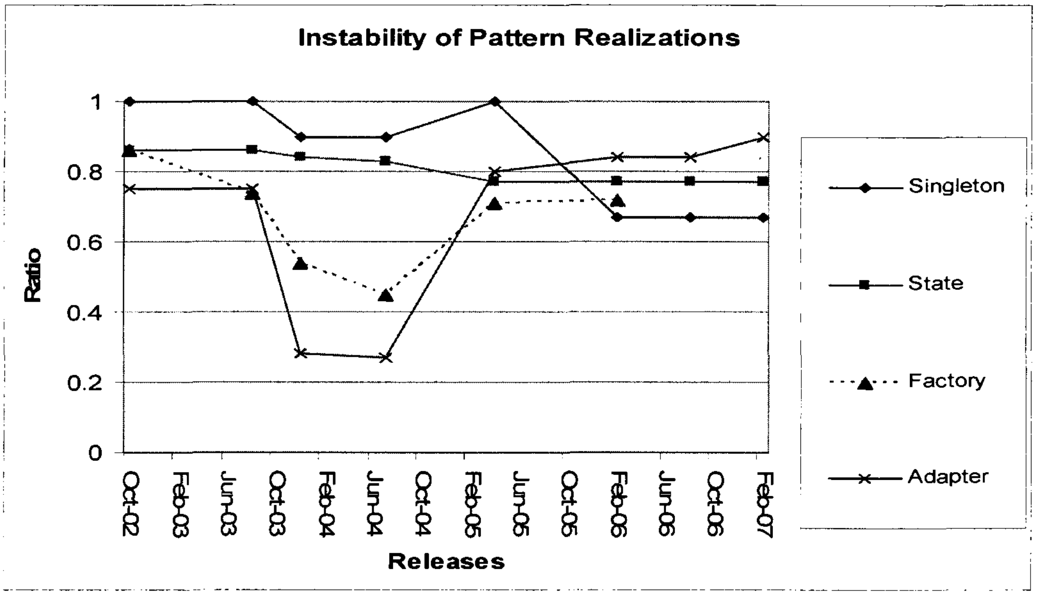


Figure 6.21 Calculated value of Instability of all possible realizations of various design patterns in ArgoUML

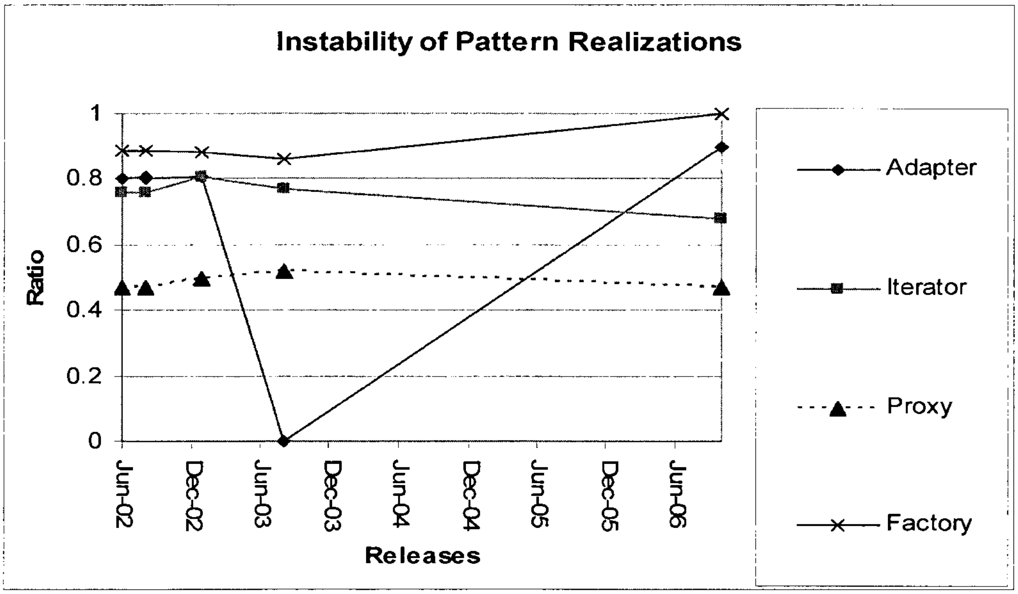


Figure 6.22 Calculated value of Instability of all possible realizations of various design patterns in eXist

Average instability values for design patterns tend to be greater than six tenths (on average) in a range of [0..1]. This leads us to believe that all systems studied are young. In other words, 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 find 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.

6.5.2 Abstractness

After studying the *abstractness* (**A**) metric, we conclude that this is not an effective technique for measuring the consequences of grime on the adaptability of design patterns. The problem with the *abstractness* measure is that grime can buildup in the form of new relationships to other classes that are either abstract or concrete, thus making this measure meaningless. For example, most design patterns promote extensibility by sub-classing from abstract classes. As a realization of a design pattern evolves, we expect **A** to approach zero. This is because the total number of concrete classes is expected to grow more quickly than the number of abstract classes. If modular grime buildup occurs in a realization of a design pattern, then new relationships are formed to external classes that are not part of the RBML specification of the pattern. These relationships can be to other abstract or concrete classes, and more likely to other concrete classes. The effect is that **A** will also approach zero.

A design pattern may evolve via changes made to existing classes rather than by the use of the intended method of extension ---adding concrete classes as shown by Bieman et al. [11]. In this situation patterns evolve, but the patterns maintain a constant abstractness level. This growth suggests grime buildup that is not identified by the abstractness measure.

6.6 Evaluation of Hypotheses

In this section we evaluate all hypotheses posed in section 4.5.

Decay, Rot, and Grime Buildup

$\mathbf{H}_{1,0}$: There is inconsequential *pattern rot* in design pattern realizations. The number of *core deviations* is minor as a system evolves.

We cannot reject $\mathbf{H}_{1,0}$. 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.

$H_{2,0}$: There is inconsequential *grime buildup* in design pattern realizations. The amount of grime buildup is minor as a system evolves.

We have evidence to support rejecting **$H_{2,0}$**. We observe modular grime buildup and the deterioration of the environment surrounding pattern realizations in JRefactory and eXist. To a lesser extent we observe this in ArgoUML. In JRefactory and eXist we observe that realizations continue to gather modular grime buildup at a steady rate, however ArgoUML appears to go through periods of refactoring that bring modular grime down before seeing slight increases again. In general, the level at which grime counts increase are less than what we expected to see. We have not tried to fit a curve to any data because from a statistical perspective, the number of data points is not significant.

$H_{3,0}$: The number of pattern realizations do not increase as the system evolves over time.

We have evidence to support rejecting **$H_{3,0}$**. Tables 6.5 through 6.7 display the total number of pattern realizations we have found as each of the systems studied evolves.

Table 6.5 Number of realizations found for JRefactory

Number of Realizations	Jan-01	Feb-02	Jul-03	Aug-03	Oct-03	May-04
Singleton	1	1	1	1	1	2
State	1	1	1	1	1	1
Factory	2	2	2	3	3	3
Adapter	12	13	13	13	15	16
Visitor	2	2	2	2	2	2

Table 6.6 Number of realizations found for ArgoUML

Number of Realizations	Oct-02	Aug-03	Dec-03	Jul-04	Apr-05	Feb-06	Aug-06	Feb-07
Singleton	1	1	1	1	1	1	1	1
State	10	11	12	12	14	14	14	14
Adapter	2	2	2	1	2	4	4	3
Factory	2	6	6	6	6	7	0	0

Table 6.7 Number of realizations found for eXist

Number of Realizations	Jan-02	Aug-02	Aug-02	Jan-03	Aug-03	Oct-06
Adapter	4	4	4	4	0	9
Iterator	7	7	7	8	9	23
Proxy	1	1	1	1	2	2
Factory	1	1	1	2	2	6

Class Grime Buildup

$H_{4,0}$: The increases in the number of public methods in a class that belongs to a pattern, is inconsequential.

Evidence does not support rejecting **$H_{4,0}$**. In JRefactory only the Adapter pattern shows signs of growth in the number of methods. The eXist system also exhibits 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 occurs in the last revision of software, and thus there is little history to support sustained growth. Prior to the last released version we see no evidence of such growth. ArgoUML exhibits measurable growth in the Factory pattern realizations. The observed growth in the total number of methods is matched by growth in the LOC for the respective patterns.

$H_{5,0}$: The increases in public attributes (fields) in a class that belongs to a pattern, is inconsequential.

Evidence does not support rejecting **$H_{5,0}$**. In JRefactory all patterns appear to maintain a constant number of attributes with the exception of the Adapter and Visitor patterns. The eXist system also exhibits 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 observe a steady growth in the State pattern realizations, however these realizations also see an increase in the number of classes that participate in the pattern realizations, thus bringing the average number of attributes per class down.

$H_{6,0}$: The increases in the sizes of classes that belong to pattern realizations is inconsequential.

The average LOC per class in all three systems studied tend to increase. One expects that such increases in average LOC per class would indicate bloating of the class, however in almost all realizations, the average LOC grew together with the average number of methods in the realizations. The only exceptions are the Adapter pattern in ArgoUML and the Factory pattern in JRefactory. In one instance we also observe a decrease in average LOC as compared to the average number of methods. This is the Factory pattern in ArgoUML, which would indicate refactoring. Therefore there is evidence to suggest accepting $\mathbf{H}_{6,0}$.

Modular Grime Buildup

$\mathbf{H}_{7,0}$: The afferent coupling C_a (fan-in) level increases of pattern realizations over time are inconsequential.

We have evidence to support rejecting $\mathbf{H}_{7,0}$. While the observed growth in afferent coupling levels was less significant than expected, we nevertheless find that every design pattern studied in every system exhibits either a tendency to remain flat or shows positive growth. In ArgoUML we observe 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.

$\mathbf{H}_{8,0}$: The efferent coupling C_e (fan-out) level increases of pattern realizations over time are inconsequential.

We have strong evidence to support rejecting $\mathbf{H}_{8,0}$. While the Singleton pattern shows no changes in efferent coupling levels (expected), all other realizations show positive growth. Growth levels tend to show much sharper increases that their afferent level counterparts.

Organizational Grime Buildup

$H_{9,0}$: The total number of packages that participate in the implementation of a design pattern remains constant over time.

There is no evidence to reject **$H_{9,0}$**. For ArgoUML, the total number of packages that participate in the implementation of various design pattern realizations remains steady and show no evidence of significant growth. In the case of eXist, the number of Java packages shows a tendency to grow over time, but the observed growth is slight, and in the case of JRefactory we observe steady numbers with the exception of the July 2003 release. We certainly do not see any negative trends in terms of the number of packages; however additional data is necessary to understand if the observed growth is significant.

$H_{10,0}$: The total number of physical files that make up the implementation of design patterns remains constant throughout the evolution of the system.

There is no evidence to reject **$H_{10,0}$**. In the cases of JRefactory and ArgoUML we clearly see a constant trend in terms of the number of files. The State pattern in the latter does show steady growth, however this appears to be an exception. In the case of eXist, we observe a steady growth after August 2003.

H$_{11,0}$: The package level afferent coupling C_a (fan-in) of packages containing pattern realizations is inconsequential.

There is no evidence to reject **H**$_{11,0}$. In JRefactory we only observe growth during the July 2003 release. Similarly, eXists exhibits growth after August 2003. ArgoUML exhibits different trends for different patterns observed. In some cases we see steady growth, while in others we see steady declines. In general, we need additional information to be able to refute this hypothesis.

Consequences of Grime Buildup

H$_{12,0}$: There is no correlation between changes in LOC and design pattern grime buildup.

Results are mixed. For both, JRefactory and eXist, we have evidence that changes in LOC lead to an increase modular grime levels. Most patterns studied in these systems show significant p-values and correlation coefficients that help support rejecting the null hypothesis. ArgoUML however did not have significant results to help reject the null hypothesis.

H$_{13,0}$: The adaptability of design patterns, measured by the Instability ratio of Ce / (Ca + Ce) tends to remain the same as patterns evolve.

Evidence does not support rejecting **H**$_{13,0}$. 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 cause the ratio values to trend higher. On average, instability values are greater than six tenths. The Singleton pattern in JRefactory was the only exception with instability values averaging just above two tenths.

$H_{14,0}$: The adaptability of design patterns, measured through its Abstractness value is inconsequential.

There is no data to evaluate this hypothesis. Abstractness, as discussed in section 6.5.2, is not a good measure to predict consequences on adaptability of design patterns.

$H_{15,0}$: Grime buildup has a higher impact on the testability than the adaptability of design patterns.

Evidence does not support rejecting **$H_{15,0}$**. The instability of design patterns tends to be higher than six tenths. This is supported by **$H_{13,0.}$** 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 additional test cases are necessary to adequately verify the functionality of a pattern. The adaptability of patterns is less affected because its afferent levels grow at a slower pace that its efferent level counterparts.

$H_{16,0}$: The minimal number of test requirements necessary to maintain test effectiveness remains constant as grime buildup increases.

We have strong evidence to support rejecting **$H_{16,0}$**. We found clear evidence that test requirements increase as a result of higher efferent levels. High efferent levels were found on almost all pattern realizations of every system under study. Test requirements grow at different pace depending on whether efferent coupling is associated with composition, dependency, generalization, or association relationships.

6.7 Threats to Validity

Clear threats to the validity of this study exist. Construct validity refers to the meaningfulness of measurements, and to validate this you must show that the measurements are consistent with an empirical relation system. An empirical relation system is an intuitive ordering of the data in terms of the attributes of interest. Clearly the dependent variable in this research is the grime buildup as measured in terms of the various metrics used in this work. We can say that some patterns appear to suffer from more grime than other patterns, and thus testability and adaptability consequences are higher. Considerable more realizations of patterns need to be studied to further analyze the validity of results.

Content validity refers to the adequate representation of the content. In order for this study to capture the notion of grime buildup, we need to investigate additional independent variables beyond the metrics we have chosen here. Additional independent variables studied can have significant effects in testability and adaptability of design patterns.

Internal validity focuses on the cause and effect relationships. In this study one can try to determine whether an increased count of some measure is directly related to the grime buildup of a software pattern. The data does provide us with initial evidence that this is the case. Temporal precedence must also be determined when examining the internal validity of a system, and in the case of testability we have clear evidence that as grime buildup occurs, test suite numbers must increase as a result of new relationships and the formation of testing anti-patterns. Additionally, adaptability of patterns suffers because they become obscured by relationships that are not inherent in the definition of the design pattern. Multiple linear regression was used to demonstrate that a structural relationship exists between ΔLOC and modular grime. Statistical models assume that all variables used in such models are independent. One possible threat to validity is that for every attribute there could be a *Set/Get* method. This would imply a dependency between such variables.

Finally, external validity refers to the ability to generalize results, and it is quite evident that we do not have a large sample to make general conclusions. Further studies of additional systems, additional design patterns, and additional grime buildup measures are required.

7. CONCLUSIONS

It is not possible to stop the aging and deterioration of designs. Evidence suggests that as design patterns age, the realizations of patterns remain and modular grime builds up. Decay is due mainly to grime rather than rot. This research has accomplished our three original goals. We have clearly defined new terms for understanding and quantifying decay, rot and grime buildup in software design patterns, we have shown, through an observational case study, empirical evidence 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 to a lesser extent, the adaptability of design patterns.

We have carefully selected measures that give us 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 methodology to gather statistics on all pattern realizations under study in order to help evaluate the consequences of decay and grime on testability and adaptability of patterns. We found no significant evidence of class or organizational grime; however we did find evidence that grime buildup is mostly due to the increase coupling observed in the classes that participate in the realizations of design patterns, i.e. modular grime.

We evaluated sixteen hypotheses 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 appears to go through periods of refactoring that bring modular grime down before seeing slight increases again. Class grime levels were not significant. All systems studied with the exception of the Adapter pattern in JRefactory display growth that is matched by an increase in the LOC for the respective patterns. In the case of eXist, we do observe a marked increase in class grime (i.e. number of methods) for the Adapter, Iterator, and Proxy patterns in the last release of the software, however this may be a single instance of this scenario, and there is no history to support a noticeable trend. At the organizational level, we found no evidence to support grime buildup. We observed how packages, number of files, and afferent coupling at a package level evolved, and with the exception of single data points, found no support to substantiate grime buildup.

Testability and adaptability of design patterns are 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, indicates 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.

We developed and applied a method to compute the minimum test requirements necessary to test various relationships between classes. We also found evidence of *concurrent-use-relationships*, *self-use-relationships*, *swiss army knife*, and *lava flow* testing anti-patterns. Further, a refinement of the existing equations is necessary to account for additional independent variables and the development of anti-patterns. Adaptability was evaluated by studying the relationship that exists between changes to lines of code and the buildup of coupling. Spearman correlation and multivariate regression showed 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. We would expect that grime buildup would be more significant in systems that have not been as successful. Additionally, we must increase the number of pattern realizations studied and the total number of releases.

The long term goals of this research are to obtain a deeper understanding of how software systems decay, and the consequences of decay on quality attributes of systems. Helping the engineering community avoid or retard decay and grime buildup can benefit software development. Our characterization of decay and grime shows how grime builds up around design patterns. We identify 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.

8. LITERATURE CITED

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

[2] Arisholm E., Sjoberg D.I.K. *Towards a framework for empirical assessment of changeability decay.* The Journal of Systems and Software 53 (1), 2000 3-14.

[3] Arisholm E., *Empirical assessment of the impact of structural properties on the changeability of object-oriented software.* Information and Software Technology 48, 2006. pp. 1046-1055.

[4] Arisholm E., Briand L.C., Foyen A. *Dynamic Coupling Measurement for Object-Oriented Software.* IEEE Transactions on Software Engineering, Vol. 30, No. 8, August 2004.

[5] Basili, V.R., Briand, L.C., Melo, W.L. *A Validation of Object-Oriented Design Metrics as Quality Indicators.* IEEE Transactions on Software Engineering, Vol 22, Number 10, October 1996.

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

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

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

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

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

[11] Bieman J.M., Straw G., Wang H. Munger P.W., Alexander R. *Design patterns and change proness: An examination of five evolving systems.* Proc. Ninth Int. Software Metrics Symposium (Metrics 2003), pp. 40-49.

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

[13] Blewitt A., Bundy A., Stark I. *Automatic Verification of Design Patterns in Java.* School of Informatics, University of Edinburgh, ASE '05, Long Beach, CA.

[14] Blewitt A. *HEDGEHOG: Automatic Verification of Design Patterns in Java.* PhD Thesis, School of Informatics, University of Edinburgh, 2005.

[15] Booch G., Jacobson I., Rumbaugh J. *The Unified Modeling Language User Guide.* Rational Software Corporation, Addison Wesley Longman, Inc. 1999.

[16] Bosch, J., *Design Patterns as language constructs.* Journal of Object-Oriented Programming, 11(2):18-32, 1998

[17] Briand, L.C., Devanbu, P., Melo, W., *An Investigation into Coupling Measures for C++.* Proceedings of the International Conference of Software Engineering, ICSE '97, Boston, MA 1997.

[18] Briand, L.C., Wust, J., Lounis, H., *Using Coupling Measurement for Impact Analysis in Object-Oriented Systems.* Proceedings of the IEEE International Conference on Software Maintenance, page 475, 1999.

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

[20] Cain, J.W., McCrindle, R.J., *An Investigation into the Effects of Code Coupling on Team Dynamics and Productivity.* Proceedings of the 26th Annual International Computer Software and Applications Conference, COMPSAC '02.

[21] Chapman, P.L. *Design and Data Analysis for researchers I.* Lecture notes for ST 511, 2007.

[22] Chidamer, S. R., Kemerer, C.F. *Towards a metrics suite for object-oriented design.* Proceedings: OOPSLA 1991, pp. 197-211.

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

[24] Design Pattern Finder. http://www.codeplex.com/DesignPatternFinder

[25] Eclipse. http://www.eclipse.org

[26] Eden, A. *Object-Oriented Modeling in LePUS3 and Class-Z.* Spring 2006, http://www.lepus.org.uk/ref/lepus3-tutorial.pdf

[27] Eden A. and Nicholson J., *The Gang of Four Companion.* December 2007, http://www.lepus.org.uk/ref/companion/index.xml

[28] Eden A. and Nicholson J., *Abstract Factory Pattern.* December 2007, http://www.lepus.org.uk/ref/companion/AbstractFactory.xml

[29] Eick S.G., Graves T.L., Karr A.F., Marron J.S., Mockus A. *Does Code Decay? Assessing the Evidence from Change Management Data.* IEEE Transactions on Software Engineering, 2001, 27(1):1-12.

[30] Fenton N.E., Pfleeger S.L. *Software Metrics: A Rigorous and Practical Approach.* PWS, Computer Press, 1996.

[31] Fowler M., Beck K., Brant J., Opdyke W., Roberts D., *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2001.

[32] France R.B., Kim D.K, Song E., Ghosh S. *A UML-Based Pattern Specification Technique.* IEEE Transactions On Software Engineering, Vol 30., No. 3, March 2004.

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

[34] Freeman Eric, Freeman Elisabeth, *Head First Design Patterns.* O'Reilly Media, Sebastopol CA, 2004.

[35] Fujaba Software Engineering Group, University of Paderborn, "FUJABA Tool Suite RE," March 2005, http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/index.html.

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

[37] Gilb T., *Principles of Software Engineering Management.* Addison Wesley, England, 1988.

[38] Godfrey M.W., Tu Q. *Evolution in Open Source Software: A Case Study.* Proceedings of the 2000 International Conference on Software Maintenance (ICSM 2000). San Jose, California, October 2000.

[39] Graves T.L., Karr A.F., Marron, J.S., Siy, H. *Predicting Fault Incidence Using Software Change History.* IEEE Transactions On Software Engineering, Vol.26, No. 7, July 2007.

[40] Gueheneuc Y.G., Albin-Amiot H. *Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-Class Design Defects.* Proc 39th International Conference and Exhibition Technology of Object Oriented Languages and Systems, pp. 296-305, 2001.

[41] Gueheneuc Y.G., Antoniol G. *DeMIMA: A Multilayered Approach for Design Pattern Identification.* IEEE Transactions on Software Engineering, Vol. 34, No. 5, pp. 667-684, September/October 2008.

[42] Gueheneuc Y.G., Albin-Amiot H. *Recovering Binary Class Relationships: Putting Icing on the UML cake.* Proc 19th Conference Object Oriented Programming, Systems, Languages, and Applications. D.C. Schmidt, ed., pp. 301-304, October 2004.

[43] Gustafsson J., Paakki J., Nenonen L., Verkamo A.I. *Architecture-Centric Software Evolution by Software Metrics and Design Patterns.* Proceedings of the 2002 Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002).

[44] Hamlet D., Voas J. M. *Faults on its sleeve: amplifying software reliability testing.* Proceedings of the ISSTA '93, Boston, MA, pp.89-98, 1993.

[45] IEEE Standard 1219-1988: *IEEE Standard for Software Maintenance.* IEEE Computer Society Press, p.4. 1998.

[46] International Standards Organization (ISO): *Standard 14764 on Software Engineering Software Maintenance.* ISO/IEC. 1999.

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

[48] Izurieta, C., Bieman, J.M. *Testing Consequences of Grime Buildup in Object Oriented Design Patterns.* 1st ACM-IEEE International Conference on Software Testing, Verification and Validation, ICST '08, Lillehamer, Norway, April 2008.

[49] Izurieta C.I., Bieman J.M. *The Evolution of FreeBSD and Linux.* Proceedings of the 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2006). Rio de Janeiro, Brazil. September 2006.

[50] Izurieta C.I., Bieary S. *A comparison of the modeling, specification, and detection capabilities of Pattern Languages.* CS 517 Class Project, Colorado State University, Spring 2008.

[51] Jacky J., *The Way of Z: Practical Programming with Formal Methods.* Cambridge: Cambridge Unversity Press, 1996.

[52] JavaNCSS. http://www.kclee.de/clemens/java/javancss

[53] JDepends Open Source Software. http://www.clarkware.com/software/JDepend.html

[54] JRefactory Open Source Software. http://jrefactory.sourceforge.net

[55] Kelly D. *A Study of Design Characteristics in Evolving Software Using Stability as a Criterion.* IEEE Transactions on Software Engineering, Vol. 32, No. 5, May 2006.

[56] Kim D. K. *A Meta-Modeling Approach to Specifying Patterns.* Colorado State University PhD Dissertation, June 21, 2004.

[57] Kim D. K., Shen W., *Evaluating Pattern Conformance of UHL Models: A Divide-and Conquer Approach and Case Studies.* Software Quality Journal, Vol. 16, No. 3, pp.329-259, 2008.

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

[59] Lehman M.M. *Program Life Cycles and Laws of Software Evolution.* Proc. IEEE Spec. Iss. On Software Engineering., vol. 68, no. 9, Sept 1980, pp. 1060-1076.

[60] Lehman M.M., Perry D.E., Ramil J.F. *Implications of Evolution Metrics on Software Maintenance.* Proc. Proc. Of the 1998 International Conference on Software Maintenance (ICSM '98), Nov 1998, Bathesda, Maryland.

[61] Li W., Henry S. *Object-oriented metrics that predict maintainability.* Journal of Systems and Software 23 (2), 1993, pp. 111-122.

[62] Lientz B.P., Swanson E.B. *Software Maintenance Management: A study of the maintenance of computer application software in 487 data processing organizations.* Addison-Wesley, 1980 [4, 7, 173].

[63] Madhavji N.H., Fernandez-Ramil J., Perry D.E. *Software Evolution and Feedback Theory and Practice.* Wiley and Sons Ltd. 2006.

[64] Martin R.C., *Agile Software Development: Principles, Patterns, and Practices.* Pearson Education. 2002.

[65] Mattsson M., Bosch J. *Observations on the Evolution of an Industrial OO Framework.* 15th IEEE International Conference on Software Maintenance (ICSM 1999), pp. 139, 1999.

[66] De Moor O., Verbaere M., Hajiyev E., Avgustinov P., Ekman T., Ongkingco N., Sereni D., Tibble J. *.QL for Source Code Analysis.* International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France 2007.

[67] De Moor O., Verbaere M., Hajiyev E., Avgustinov P., Ekman T., Ongkingco N., Sereni D., Tibble J. *.QL: Object-Oriented Queries Made Easy.* International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Costa Verde, Portugal 2007.

[68] Niere J., Schäfer W., Wadsack J.P., Wendehals L., and Welsh J.. "Towards Pattern-Based Design Recovery," *in International Conference on Software Engineering*, 2002, pp. 338-348.
URL: http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2002/ICSE2002_final.pdf

[69] Niere J., Wendehals L., and. Zündorf *A. An Interactive and Scalable Approach to Design Pattern Recovery,* University of Paderborn, Paderborn, Germany, Tech Report tr-ri-03-236, 2003.
URL: http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2003/tr-ri-03-236.pdf

[70] Ott R.L, Longnecker M. *An Introduction to Statistical Methods and Data Analysis, Fifth Edition.* Duxbury, Pacific Grove CA, 2001.

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

[72] PatternSeeker Tool. Colorado State University.

[73] Resource Standard Metrics. *What's Inside Your Source Code?* Metrics definitions. http://www.m2tech.net/m2rsm/docs/rsm_metrics_narration.htm

[74] Rombach H.D., *Design Measurement: Some Lessons Learned*, IEEE Software 17-25, 1990.

[75] Rosdal A. *Empirical study of Software Evolution and Architecture in OSS projects.* Department of Computer and Information Science. Norwegian University of Science and Technology. 2005.

[76] SAS Statistical software. http://www.sas.com

[77] Scacchi W. *Understanding Open Source Software Evolution: Applying, Breaking, and Rethinking the Laws of Software Evolution.* Technical Report. Institute for Software Research. University of California, Irvine, April 2003.

[78] Semmle Query Technology. http://semmle.com

[79] Shi N., *Reverse Engineering of Design Patterns from Java Source Code*, Ph.D. Dissertation, University of California-Davis, Davis, CA, USA, 2007. http://www.cs.ucdavis.edu/~olsson/students/ShiNija.pdf

[80] Software Engineering Group, University of Paderborn, *FUJABA Tool Suite RE*, March 2005, http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/index.html.

[81] SourceForge. Open Source Software, http://sourceforge.net

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

[83] Turski W.M. *The Reference Model for Smooth Growth of Software Systems Revisited.* IEEE Transactions on Software Engineering, 22(8), August 1996.

[84] University of Essex Department of Computer Science, *Legend: Key to symbols, LePUS3 and Class-Z.* http://www.lepus.org.uk/ref/legend/legend.xml

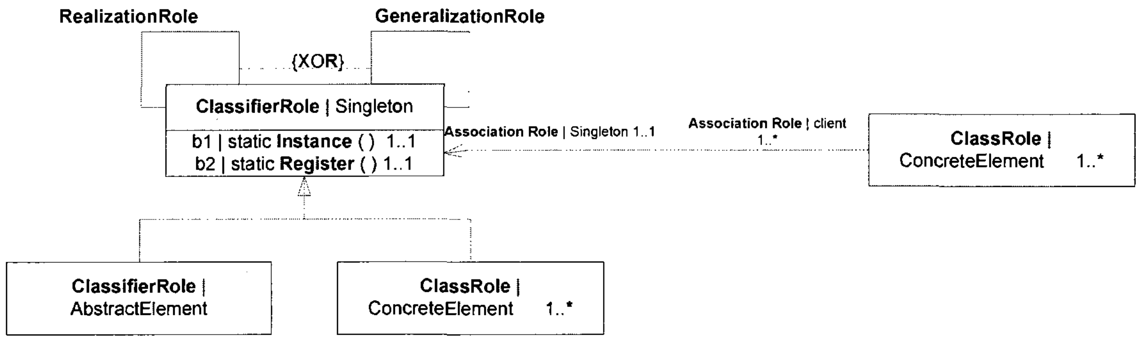
[85] Voas J.M., Miller K.W. *Software testability: the new verification.* IEEE Software, 12(3), pp.17-28, May 1995.

[86] Windows Operating System. http://www.microsoft.com

Appendix A RBML Descriptions

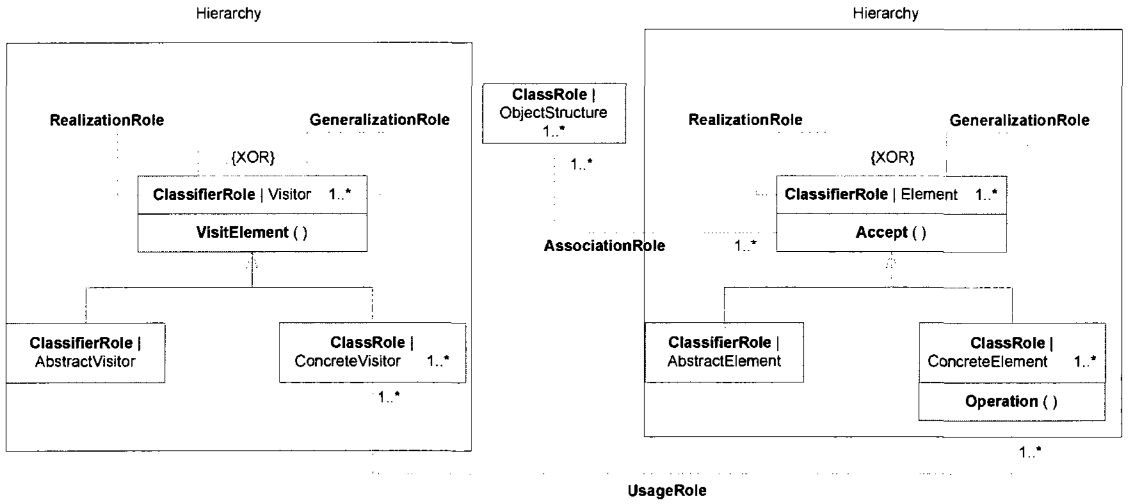
The following are the abbreviated RBML static meta role models of the design patterns studied.

1. Singleton Pattern

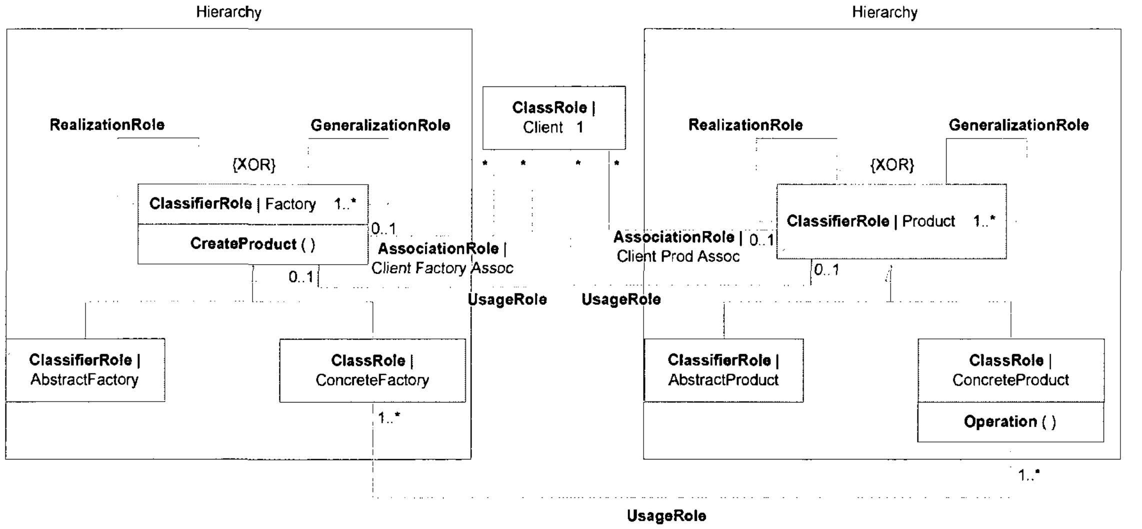


RealizationRole
GeneralizationRole
{XOR}
ClassifierRole | Singleton
b1 | static Instance ( ) 1..1
b2 | static Register ( ) 1..1
Association Role | Singleton 1..1
Association Role | client
1..*
ClassRole |
ConcreteElement 1..*
ClassifierRole |
AbstractElement
ClassRole |
ConcreteElement 1..*

2. Visitor Pattern

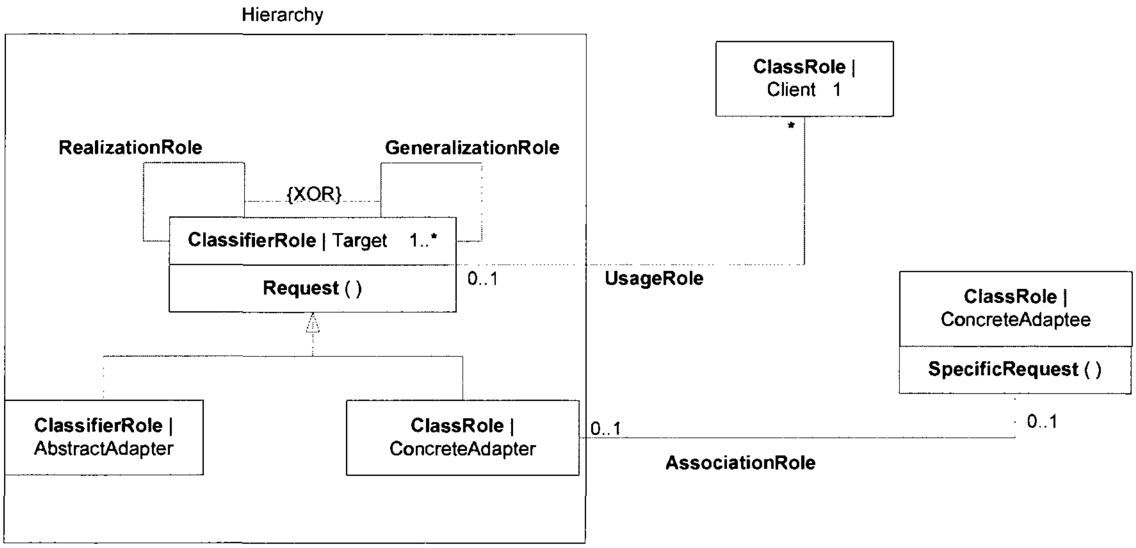


3. Factory Pattern

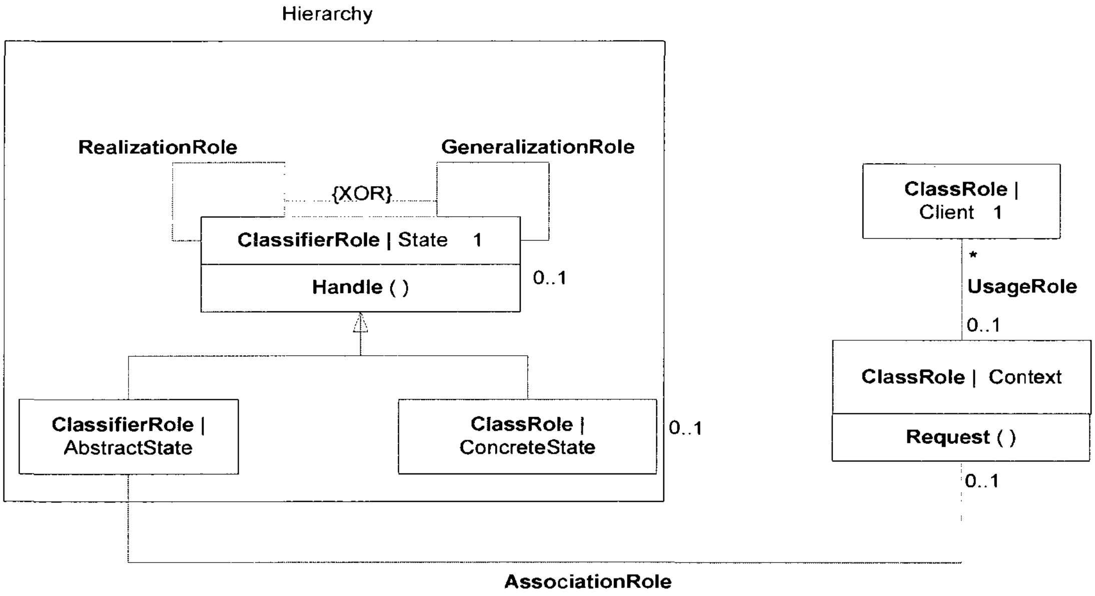


4. Adapter Pattern

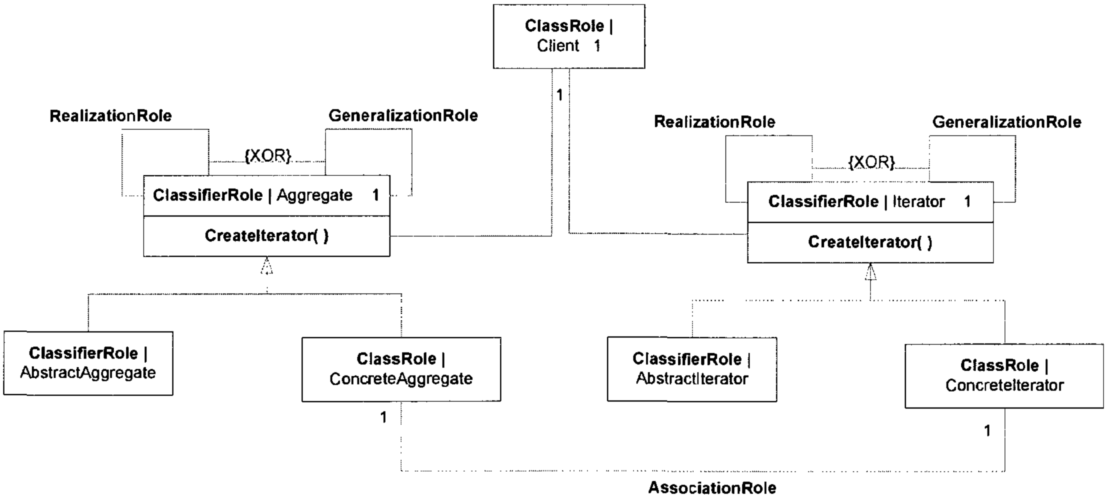
The search for adapter patterns was restricted to the object composition adapter type. Class adapter realizations are not feasible in Java code as multiple inheritance is not available.



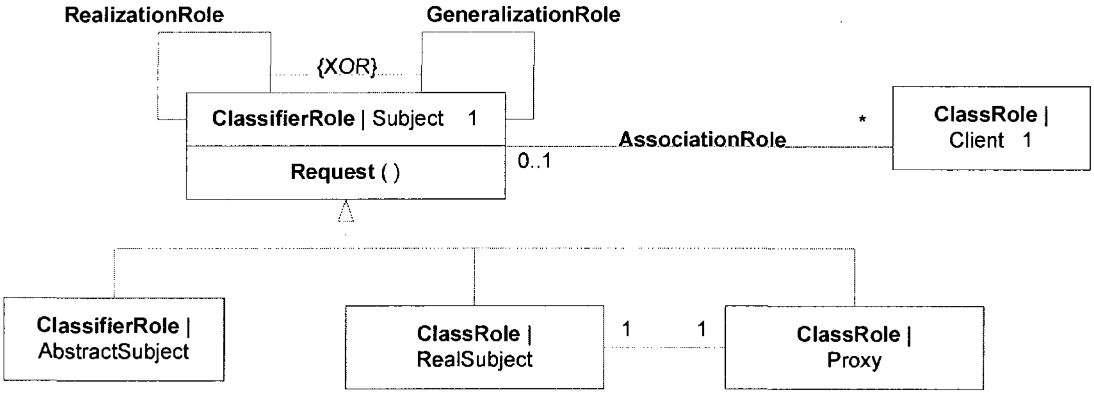
5. State Pattern



6. Iterator Pattern


ClassRole | Client 1
1
RealizationRole
GeneralizationRole
{XOR}
ClassifierRole | Aggregate 1
CreateIterator( )
RealizationRole
GeneralizationRole
{XOR}
ClassifierRole | Iterator 1
CreateIterator( )
ClassifierRole | AbstractAggregate
ClassRole | ConcreteAggregate
1
ClassifierRole | AbstractIterator
ClassRole | ConcreteIterator
1
AssociationRole

7. Proxy Pattern



Appendix B Custom Scripts

The following script was used to gather class counts, abstract class counts, efferent and afferent coupling at the Java Package level. The script uses the pattern finder tool as well as JDepend.

```
#
# Metrics gatherer.
#
#  Clem Izurieta
#  CS 799 Dissertation Research
#

function processFile()
{
      echo "Processing $1..."
      # Process the txt design pattern file.  We only care about the
      # java files.
      sed -n -e '/java/p' $1 > tmp.txt
      mv tmp.txt $1

      # Clean up the bag and other possible residuals
      rm -f bag/*
      rm -f raw.out

      # Copy the actual files to the bag.  This is a list of files that
      # was output by Design Pattern Finder.
      cat $1 | while read -r line
      do
            echo \"$line\"
            cp "$line" bag
      done

      #
      # Run tools to gather metrics
      #
      # JDepends gathers many coupling and class metrics for any java
      # files that are in the bag.
      java jdepend.textui.JDepend -file ${1}.jdepend bag

      echo "Adding up metrics..."
      # Add up the relevant metrics for all packages involved in this
      # pattern.
      sed -e '1,/- Summary/d' ${1}.jdepend > tmp1.txt
      sed -e '1,/Name/d' tmp1.txt > tmp2.txt
      sed -e '/^$/d' tmp2.txt > raw.out
      rm -f tmp*.txt

      name=`basename ${1} txt`
      rm -f ${name}total
      echo > ${name}total
      echo >> ${name}total
      echo "******** $name" >> ${name}total
      echo "-------------------------" >> ${name}total
      echo "Total number of packages involved in this pattern = `wc -l
raw.out`" >> ${name}total

      total=0
      for value in `cut -d, -f2 raw.out`
      do
            total=`expr $total + $value`
      done
      echo "Class Count = $total" >> ${name}total

      total=0
      for value in `cut -d, -f3 raw.out`
      do
            total=`expr $total + $value`
      done
      echo "Abstract Class Count = $total" >> ${name}total

      total=0
      for value in `cut -d, -f4 raw.out`
      do
            total=`expr $total + $value`
      done
      echo "Ca Total = $total" >> ${name}total

      total=0
      for value in `cut -d, -f5 raw.out`
      do
            total=`expr $total + $value`
      done
      echo "Ce Total = $total" >> ${name}total
}

#
# Main
#
rm -f *.total
for i in `ls *.txt`
do
      echo "Pattern: =========> $i"
      processFile $i
done
rm -f bag/*
rm -f raw.out
```

Appendix C Maintenance Effort Statistics

JRefactory numbers:

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Singleton	0	0	9	3	2	2
	0	-1	9	3	2	2
	0	0	11	3	2	2
	0	0	11	3	2	2
	0	0	15	3	2	2
	1	12	22	6	3	4

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
State	0	0	30	54	25	37
	0	0	30	54	25	37
	0	0	30	54	25	37
	0	0	30	54	25	37
	0	0	30	54	25	37
	0	0	30	54	25	37

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Factory	0	0	41	57	10	39
	0	31	43	65	10	43
	0	0	46	65	10	43
	2	272	59	142	17	53
	-1	39	61	148	13	53
	0	17	85	151	13	53

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Adapter	0	0	13	104	19	27
	0	11	13	109	19	30
	0	12	19	109	19	35
	3	21	19	109	26	40
	-1	506	48	225	23	183
	1	106	47	246	26	192

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Visitor	0	0	169	927	24	759
	0	171	170	934	25	766
	0	933	190	1038	34	872
	0	22	215	1037	34	872
	2	-223	255	1026	35	858
	0	785	268	1124	35	924

eXist numbers:

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Adapter	0	0	10	44	4	9
	0	0	10	44	4	9
	0	0	10	40	4	9
	0	15	11	47	4	9
	-4	-151	0	0	0	0
	9	507	5	45	0	194

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Iterator	0	0	13	42	16	30
	0	0	13	42	16	30
	0	0	13	42	16	30
	2	30	16	67	23	39
	0	73	15	49	15	36
	14	322	56	117	46	74

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Factory	0	0	1	8	0	2
	0	0	1	8	0	2
	0	0	1	8	0	2
	1	29	2	14	1	8
	0	0	1	6	1	6
	4	143	0	27	0	12

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Proxy	0	0	11	10	4	20
	0	19	11	10	4	20
	0	0	11	10	4	20
	0	21	12	12	4	28
	1	59	16	17	6	34
	0	270	25	22	8	53

ArgoUML numbers:

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Singleton	0	0	0	1	0	1
	0	1	0	1	0	1
	0	12	1	9	0	2
	0	0	1	9	0	2
	0	8	0	1	0	1
	0	4	1	2	1	1
	0	0	1	2	1	1
	0	0	1	2	1	1

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
State	0	0	27	165	86	247
	0	-72	26	158	86	203
	10	-46	49	262	92	215
	0	180	56	281	94	233
	7	1134	67	221	110	289
	0	134	67	226	110	282
	11	418	80	262	134	283
	-1	-53	79	263	134	287

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Adapter	0	0	4	12	1	7
	0	0	4	12	1	7
	0	145	38	15	15	16
	-1	10	35	13	15	14
	1	8	3	12	18	15
	2	295	5	26	15	13
	2	7	5	26	15	13
	-1	-73	3	26	15	13

Name	Change in Classes	Change in LOC	Ca	Ce	Number of Attributes	Number of Methods
Factory	0	0	2	12	4	6
	12	1297	14	39	15	17
	0	1512	69	82	19	60
	0	375	136	110	21	82
	10	661	53	128	17	414
	1	84	55	142	20	440