A TAXONOMY OF MODULAR GRIME IN DESIGN PATTERNS

by

Travis Steven Schanz

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2011

APPROVAL

of a thesis submitted by

Travis Steven Schanz

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency and is ready for submission to the Division of Graduate Education.

Dr. Clemente Izurieta

Approved for the Department of Computer Science

Dr. John Paxton

Approved for the Division of Graduate Education

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Clemente Izurieta, for the immense amount of support and direction he provided me throughout my research. His guidance during our research kept me focused on the road ahead, even when I tended to get sidetracked. Whenever I felt as though I was out of ideas, a meeting with Clem gave my brain the restart that it needed.

His general guidance as an advisor was invaluable. I feel as though I have maximized my learning and research potential in the amount of time that I have had. His advice allowed me to finish this research as well as publish results along the way. I couldn‘t have done it without him.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

TABLE OF CONTENTS - CONTINUED

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES - CONTINUED

LIST OF FIGURES – CONTINUED

ABSTRACT

Software designs decay over time. While most studies focus on decay at the system level, this research studies design decay on well understood micro architectures, design patterns. Formal definitions of design patterns provide a homogeneous foundation that can be used to measure deviations as pattern realizations evolve. Empirical studies have shown modular grime to be a significant contributor to design pattern decay. Modular grime is observed when increases in the coupling of design pattern classes develop in ways unintended by the original designer. Further research is necessary to formally categorize distinct forms of modular grime. We identify three properties of coupling relationships that are used to classify subsets of modular grime. A taxonomy is presented which uses these properties to group modular grime into six disjoint categories. We gather data from three open source software systems to test hypotheses about the significance of grime buildup for each of the six taxonomy categories. The results reveal that one form of modular grime is more apt to develop than others. This was observed in all the systems in the study. We also found that some types of modular grime show insignificant growth while others vary between systems. We conclude that certain types of modular grime are more likely to contribute to design pattern decay.

1. INTRODUCTION

Software evolution has been the subject of empirical research for the last 35 years [52]. Evolution occurs as software changes over time. Changes occur as a result of functionality being added, operating environment changes, defect fixes, etc. Developers cannot reasonably predict all future changes during the initial development, and changes can create new problems. Software decay is a type of evolution and the consequences of decay result in software that is less maintainable. The possible implications of decay reveal a predicament. As the complexity and fault-proneness of the software increases as a result of decay and code size [8], developers must spend more time maintaining and adapting the source code. Software decay research has been sparse [28], [62]. There is little or no research on techniques for decay detection.

The introduction of design patterns in object oriented software has had a significant effect on development techniques. Design patterns are defined by Gamma, et. al. [30] as ―simple and elegant solutions to specific problems in object-oriented software design". The extensibility of design patterns has led to their common use in object oriented software. This suggests another question. How do design patterns decay?

Izurieta [39] initially proposed a hierarchical definition of design pattern decay. Empirical studies [39], [41] revealed that one form of decay called modular grime was most prominent. Modular grime occurs when design pattern classes develop new relationships unintended by the original designer over time. Relationships become increasingly coupled to other classes as systems evolve.

The significant buildup of modular grime in Object Oriented systems studied by Izurieta [39] provides the motivation for this research. In order to discover the underlying causes of modular grime buildup, we study how modular grime develops in design patterns.

We propose a taxonomy for modular grime. The taxonomy is created using characteristics of coupling relationships which form the basis of modular grime. We use three characteristics apparent in all coupling relationships: strength, scope, and direction to classify each relationship contributing to modular grime. We propose six disjoint subsets of modular grime. To validate the taxonomy, we conduct an empirical study to determine which parts of the taxonomy contribute to the buildup of modular grime. We pose hypotheses about the growth trends of each type of modular grime and gather data from 3 open source software systems. We use statistical analysis to test our hypotheses.

Section 2 presents a literature review. Section 3 defines the criteria for classification along with the resulting taxonomy. The study methodology and resulting data are discussed in sections 4 and 5. Sections 6, 7, and 8 contain analysis, discussion, issues with validity and concluding remarks.

2. BACKGROUND

This section provides the necessary background on state of the art and relevant research on software evolution, software decay, and design pattern decay.

2.1 Software Evolution

Software changes over time. External attributes that influence the source code (e.g. changes to requirements, addition of features, bug fixes) cause incremental changes to the software structure over time. Software evolution refers to the sequential series of changes that occur during a development lifecycle.

2.1.1 History

Seminal work in software evolution began in the 1970‘s when Lehman released research results from his study on the OS/360 system [52]. The study documented the growth (defined by module count) of the system over several revisions. The results showed growth trends that continued through all revisions of the software. This seminal study led to the creation of three software laws for E-type systems. An E-type system is defined by Lehman as ―one implementing some computer application in the real world" [56]. The three laws are:

1. *Continuing Change*: E-type systems must be continually adapted to stay satisfactory.
2. *Increasing Complexity:* Complexity increases as the system evolves.
3. *Self Regulation*: Evolution is self regulatory.

Later research [53], [54], [56] added 5 more laws of evolution:

4. *Invariant work rate*: Average activity rate is invariant.

5. *Conservation of familiarity*: As a system evolves, everyone involved must stay familiar. Excess growth does not allow this so incremental growth occurs.

6. *Continuing growth*: Functional content increases to keep users satisfied.

7. *Declining quality*: Quality seems to decline unless great steps are taken to maintain the system and adapt it to the operational system changes.

8. *Feedback System*: Evolution processes are feedback systems.

These laws started the first historical phase of research into how software evolves. Subsequent studies focused on gathering data to test the laws.

A study in 1998 by Lehman [55] as part of the FEAST/1 project, sought to examine the effects of feedback on software evolution. Lehman, et. al. used an expanded set of metrics in addition to basic size metrics to gather data on a much larger set of software systems. The expansion of the metrics set was necessary to increase the range of tests on the laws of evolution. New metrics are continually created to measure characteristics of software; they can be further applied to evolution by measuring changes in these characteristics over time. The results from the study were supportive of laws 1, 2, 4, 6, and 8 while the evidence was inconclusive for laws 3, 5, and 7. Research by Yuen [18] on different software systems supported the first 5 laws. The results, however, did not support the laws 6, 7, and 8. Yuen concluded that these laws were system specific. Cook and Roesch [20] used standard complexity metrics such as McCabe‘s cyclomatic [58] complexity and Halstead‘s complexity [36] measures to examine phone switch software in 1994. Their results were also supportive of laws 1, 2, and 3. More recently, a study in 2009 by Ali and Maqbool [1] gathered data on modules added, deleted, and changed. Their results supported the 5th law of evolution.

In the last 30 years, research has continued to expand the set of metrics used to test the hypothesized laws. The number of systems studied also continues to increase. Further research [8], [32] shows that software evolution is still a popular research topic. The first era of research from the 1970s to the 1990s studied E-type systems and most experiments [10] used size metrics (LOC, module count, etc.) to analyze the laws of software evolution created by Lehman. Although testing of the original laws continues, research on software evolution has itself evolved with the introduction of new technologies and new paradigms of software distribution. As a result, today‘s software evolution research presents new challenges that must be considered by researchers.

Although Lehman‘s laws were a seminal part of evolution research, they are not easily quantifiable. Thus limiting their testability and making them controversial in the research community.

2.1.2 Current Work on Evolution

Today, software evolution research is defined by changes in the areas of technology, software research, and software distribution. Specifically, there are three major changes that have affected how researchers study software evolution. First, the development and popularity of object oriented programming languages has changed how people view software systems. Object oriented software is fundamentally different from procedural software, which was the focus of most studies in early evolution research. Encapsulation, inheritance, and polymorphism are features of object oriented languages but not procedural languages.

Initial research [49] showed that these features should be taken into account when gathering data from object oriented systems. Secondly, advances in metrics created for empirical software research have given researches more tools to analyze how software systems change. Examples are complexity metrics such as [58], and other advanced metrics for coupling and cohesion such as [17], especially those created as a result of research on object oriented systems. Thirdly, distribution paradigms have changed. Research conducted during the infancy of software evolution focused on commercial systems [52], [10]. During the last 15 years, new distribution paradigms like GNU [31] have given researchers access to large, high quality software code bases. The introduction of open source software creates more research questions. Does open source software evolve differently from commercial software? Are the laws of evolution apt at describing open source software? These questions can only be answered with additional research. Lastly, meta-evolution research, defined as the study of the metrics and methods used in evolution research, is the result of researchers taking a critical look at their own techniques. Meta-evolution analyzes the techniques used in evolution research with the goal of improving the usefulness and validity of results.

2.1.2.1 Object Oriented Languages. The use of Object Oriented programming languages has increased significantly in the last 15 years. Empirical software research has followed suit by developing metrics specific to object oriented software [17]. The use of inheritance and polymorphism in object oriented languages lessens the value of traditional software metrics like McCabe‘s Cyclomatic Complexity [58], and traditional size metrics (module count, source lines of code). The seminal work in software evolution focused wholly on size metrics like module count and source lines of code (SLOC). The original work by Lehman studied the size of the OS/360 system using simple size metrics. A study designed to validate the laws of software evolution [18] used the same metric; size was the independent variable. As the field of research matured, researchers [37], [1] started to view the changes in software over time in different ways (modules changed, added, and deleted). The popularity of object oriented languages gave researchers greater motivation to focus on changes to software structure in addition to size.

A common set of metrics for object oriented software were introduced by Chidamber and Kemerer [17]. They measure structural characteristics specific to object oriented software; the metrics are depth of inheritance tree (DIT), weighted methods per class (WMC), number of children (NOC), coupling between object classes (CBO), response for a class (RFC), and lack of cohesion in methods (LCOM). These metrics are applied to many areas in empirical software research, including software evolution. Li, et al. studied object oriented systems using system and class metrics [50]. Systems metrics are object oriented metrics that are similar to the size metrics originally used in software evolution studies. These metrics give a general idea of changes in the system by measuring classes added, classes deleted, classes names changed, etc. Class metrics measure structural characteristics of the object oriented design. They used metrics created by Chidamber and Kemerer as well as similar metrics from earlier research [49]. In the study, Li's results show three distinct phases of evolution as characterized by changes in the results for system and class metrics. The phases are initial, smooth growth, and final. System metrics increase quickly during the initial phase; which occurs during design when classes are created and extreme design changes are most likely. The lack of implemented classes during design causes little change in the class metrics. The smooth growth phase is marked by a steady increase in the systems metrics as classes are consistently added to the implementation. In the final phase the system metrics become stable, but the class metrics are more likely to change as implementation details are perfected. This result could be the effect of modifications to design as problems are solved through refactoring in mid development. The use of a variety of metrics in this study allowed researchers to discern different phases of evolution from design to implementation. The discrepancy between system and class metrics over time allowed the authors to predict the processes that occur during the phases of evolution.

Another characteristic of object oriented systems are design patterns [30]; which are common solutions to well known problems in design. The increased use of object oriented languages has indirectly given researchers another angle to view software evolution in the form of design patterns. Gustaffson, et al. [35] combined traditional metrics like cyclomatic complexity with object oriented metrics. They also mined design patterns and gathered metrics specific to design patterns like relative number of classes (a ratio of the number of classes that play a role in a design pattern to the total number of classes). They use these metrics as quality indicators and store results over a series of releases to analyze trends of design patterns over the evolution of software. The authors use the history of the data to predict design quality.

The object oriented paradigm has forced empirical researchers to investigate evolution in a new way. The unique characteristics of object oriented systems create new research questions and allow researchers to develop new metrics to study those questions. Simple metrics like module size and SLOC may not capture increases in complexity and other signs of decay when used alone. Higher abstractions of design used in object oriented languages, such as design patterns, have also had consequences on software evolution research.

2.1.2.2 New Metrics. Since Lehman‘s seminal work, researchers have developed new metrics to measure complexity and size in order to predict maintainability and developer effort [49] [65] [23].

Ramil and Lehman created evolution metrics to predict future developer effort. Their metrics were useful as a predictor of future effort. Li, et. al. analyzed changes in common object oriented metrics, like depth of inheritance tree (DIT), over the evolution of a system; no significant changes were found. Dagpinar and Jahnke used the same object oriented metrics to predict maintainability. Their results varied.

Ali and Maqbool [1] studied the number of modules added, deleted, and changed to examine the evolution of software at a finer granularity. Cook and Roesch [20] examined a phone switch system using complexity metrics. Their results supported Lehman‘s first three laws of software evolution. Cook et al. [19] created static metrics that were used to predict the future evolvability of a system. Specifically they created metrics to measure analyzability, changeability, testability, and compliance. All of their metrics were created or modified with the intention of measuring static characteristics of systems that are used to predict future evolution.

2.1.2.3 Open Source Software. The popularity of open source software distribution has changed the way that software is developed. From a research perspective, open source software is easier to study. Source code and revision histories are usually available in the public domain and can be accessed by interested researchers. Unlike open source software, commercial software source code is closely guarded. Early researchers of evolution had to make arrangements with commercial companies to study their software and published results required discretion about software details.

Researchers must also consider how empirical results from experiments on open source software will translate to commercial software and vice versa. This question has been the subject of much research. Izurieta and Bieman studied the evolution of FreeBSD and Linux [40]. They gathered the following metrics: SLOC, number of directories, total size, and SLOC of header files. Their results showed that the rate of change of open source software did not seem to differ from studies of growth in commercial software.

2.1.2.4 Meta-evolution. In recent years researchers have began studying the methodology used to study evolution. This is referred to as meta-evolution. The field of meta-evolution studies software evolution research and determines what processes and metrics work best to answer the questions about how software evolves and decays. Results in meta-evolution can be used to focus future work using proven, valid, and standardized methodologies. This section summarizes some results from research on meta-evolution.

Kemerer and Slaughter [48] published results from a case study they conducted on software used by a large retailer. Instead of reporting results alone, they used the opportunity to give details about the methods they used in the creation of their case study. The motivation of their research was to use their study of software evolution as an example and look back on the methods used by them during the study. Kemerer and Slaughter give in-depth analysis of the three main processes they undertook during the case study: subject (i.e. software) selection, data gathering, and data analysis.

Finding relevant software to study is important. Choosing software that does not provide necessary information, depending on the motivation of the study, will not provide results of any value, regardless of the data gathering and analysis techniques used. Kemerer and Slaughter [48] wanted to analyze the maintenance tasks undertaken during the evolution of the software. They were able to find a commercial company that had employed mostly the same developers for many years. The company also put a high value on keeping detailed change records for a long period of development history. Most importantly, the company was cooperative in providing all information necessary for data collection. Not all of these qualities (tenured developers, detailed development records, and cooperative management) can be found in every module of software, but research candidates should be analyzed with regard to these characteristics to determine if useful data can be found.

With regards to data gathering, Kemerer and Slaughter recognize that many longitudinal studies of software evolution focus on software releases [10] [20]. There are generally very few distinct releases available for study, as compared to development days or code changes; this results in a small sample size. They recommend using a finer unit as the independent variable; in their case they used each change in the change history. This provided them with a sample size of 25,000 (i.e. one copy of source code after each change was logged).

There are many options for data analysis. While time series analysis is commonly used, some data may not meet the statistical assumptions necessary for its use. For example, the data must be continuous. Kemerer and Slaughter also note that a time series analysis ―did not exploit the richness of our coding categories for the data". They decided to use a sequence analysis most often used in the social sciences. Their conclusion is that time series analysis, although very common, may not always be the best choice for data analysis. It should be used when time as an independent variable is strictly related to the variable of study (e.g. when the rate of change over time of some value is important to the research question). There may be alternative methods from other areas of empirical research, such as sequence analysis, that work best to analyze data gathered in evolution studies.

2.2 Software Decay

Software decay is a specialized type of software evolution. While evolution follows all changes to software over its lifespan, decay occurs specifically when software becomes less maintainable over its evolution. Eick [28] defines code as being decayed if it is ―harder to change than it should be". Difficulty of change can be determined by increases in the number of developers as a result of a decrease in maintainability and adaptability of the code. Decay is measured by studying how code becomes less maintainable and adaptable as it evolves.

Whereas research on evolution of software has continued over the span of many decades, studies on software decay have been infrequent. However, although decay itself has not been studied often, some research on software evolution has focused on analogous concepts such as maintainability and adaptability in order to predict future developer effort [49], [65]. In some sense these are studies of decay even though they do not explicitly use the term. The rest of this section describes some details of the research conducted on software decay.

Eick [28] presents detailed research on software decay using a medical analogy. Decay is like a medical disease; with a cause, symptoms, and risk factors. Causes are characteristics of the software and development environment that create changes that evoke code decay. Symptoms are measurable traits of the code that indicate decay has occurred. Risk factors increase the probability that code decay will occur. He uses decay indices that quantify these definitions and data is gathered from change management data. The decay indices create a model that is used to predict future faults. The most relevant result suggests that a large amount of recent changes to a module is a significant predictor of future faults in said module. The data also shows that the number of developers that touched a module was not a predictor of future faults. The ability of the code indices to model developer effort was lacking. This research is one of the most in-depth studies of code decay but, as Izurieta [39] notes, the decay indices are a quantitative measure of *all* change and not necessarily negative change.

Ohlsson et. al. [62] studied the decay of legacy software. They attempt to model the fault proneness of C modules using a variety of size, structure, and relationship (coupling) metrics. Their research reveals that coupling measurements of modules were indicative of fault proneness. The use of the C programming language in this study makes it less relevant to current day object oriented research on software decay.

Several studies of software evolution have implicitly studied decay, even though decay was not a stated objective of the study. For example, Cook et. al. [19] studied the evolution of software systems from two different points of view: static and dynamic. The dynamic viewpoint focuses on trends over the evolution of software. This is similar to most studies in software evolution. The static viewpoint focuses on static characteristics of software that make it more *evolvable;* where evolvability is defined as ―the capability of a software product to be evolved to continue to serve its customer in a cost effective way." By defining evolvability using cost effectiveness, the authors imply that static characteristics of software predict future developer effort. In this sense, evolvability is the inverse of decay. When evolvability increases, decay is less likely. The evolvability of software is primarily determined by the use of good design practices, such as design patterns and modularization, that make changing the software less likely to introduce new bugs or complexities.

Gustaffson et. al. [35] study software evolution from the design phase to the coding phase using software architecture as the unit of study. Their goal is to gather quality metrics, store them in a repository, and use them to predict the future quality of the system. They do not define quality, but the assumption is that decay has an inverse relationship with design quality. They research the prevention of decay in order to increase future design quality.

Software decay is a form of software evolution. Specifically, decay occurs when changes to software over its lifetime cause increases in developer effort. This happens because the code becomes more complex and less maintainable. Research on software decay aims to understand how software decays, and to predict how software will decay throughout the product lifecycle. All past research has been done at the system or module level. This thesis is motivated by studies that view decay from the design pattern level. The next section gives a background of design pattern evolution and decay.

2.3 Design Pattern Evolution

This section discusses the introduction of design patterns to object oriented software as well as the limited research on design pattern evolution.

2.3.1 Design Patterns

Design patterns are defined by Gamma et. al. as ―simple and elegant solutions to specific problems in object-oriented software design”. Design patterns are also reusable, meaning they are used for a variety of situations in many different architectures. They realize a generic solution for a set of functional requirements. More importantly, they are simple and elegant, which allows developers to easily understand them and extend them without modifying existing classes and increasing code complexity.

The common use of design patterns gives empirical researchers a new subject of study during the evolution of software. While much research on software evolution has been done on architectures, systems, or modules, design patterns provide researchers a micro-architecture as the subject of study [42]. The universal definitions used to define design patterns create a common understanding of what constitutes a design pattern. This allows researchers to study the evolution of design patterns across software in general. The original definitions of design patterns from Gamma et. al. are intuitive but not formal. Different formalizations have been introduced [29], [34]. This research uses Role Based Metamodeling Language (RBML) [29] to formally define design patterns; section 3.1 discusses how RBML is used to formally define design patterns.

The rest of this section gives a background on past studies of the evolution of design patterns. Section 2.4 introduces research on design pattern decay that forms a basis for this thesis.

2.3.2 Pattern Evolution

A common research question related to design patterns is: are classes involved in design patterns more change prone than other classes? Several studies [11], [28] have investigated this question. The general intuition is that design patterns are efficient, adaptable solutions that can be extended using abstraction and delegation. This implies that changes to design patterns come in the form of specializations, not in the form of modifications of existing concrete classes.

Early research on design patterns by Bieman et. al. [11] sought to test the intuitions of design pattern advocates by analyzing change histories of a software system. They designed a case study around two questions. Are larger classes more change prone? If so, are design pattern classes less change prone than other classes after accounting for class size? The author‘s hypothesize that the answer to both questions is yes. Larger classes should be more change prone simply because there is more code that may change. Design pattern classes should be less change prone because design patterns emphasize extension by specialization. Therefore existing pattern classes should not change.

Results from the Bieman study show that larger classes were more change prone, but design pattern classes were actually *more* change prone than average classes. This contradicts the intuition that design patterns allow for extensibility. A secondary study [12] on four additional software systems confirmed these results. One interesting result was that the one system that did not demonstrate these results was the only open source system in the case study.

More recently Di Penta, et. al. [25] conducted another study on the change proneness of design pattern classes. In addition to studying the relationship between design pattern membership and change proneness, they also analyzed the relationship between design pattern role and change proneness. By breaking design pattern classes into groups based on roles, the authors extended earlier studies by increasing granularity; the roles in each design pattern were studied as opposed to design pattern as a whole. As a secondary motivation, they also studied what types of changes occurred to the design pattern classes. Their results were unsurprising. Pattern classes that were concrete (i.e. playing concrete as opposed to abstract roles) were more change prone. This result provides evidence that the system under study evolved in a manner in which it was intended, through extension of abstract and interface classes. The authors note that interfaces should be carefully designed to prevent extensive change because changes to interfaces make other parts of the program less fit for change.

Vokac [72] also tests intuitions about the positive effects of design pattern usage. The author notes, ―By designing an application with the proper use of design patterns, we should reduce the number of defects". To test this hypothesis, the defect frequencies of design pattern classes of a commercial software product are analyzed. A linear regression is used to model defect frequency using pattern membership along with class size and time (to test for sequential trends). The regression coefficients act as a probability ratio for defects. The results show that class size does significantly correlate with defect rate. Adding 1000 lines of code to a class doubles the amount of defects, as measured by the number of bug fixes. After class size is considered, the results show that Template and Factory pattern classes have lower defect rates (72 and 63 percent of the average number of bug fixes respectively).

Singleton classes have a significant interaction with size, meaning that Singleton classes that increase in size see an increase in defect frequency. An interaction between Singleton and Observer shows a very low defect frequency, 32 percent of the average. The authors note that there are a set of classes in the system that are both Singletons and Observers. These classes are important and well designed, leading to the low defect frequency of the interaction.

Some studies compare the evolution of design patterns with non-pattern based solutions. Prechelt, et. al. [64] studied the evolution of design patterns using maintenance history. The authors compared maintenance histories of design patterns to alternative solutions, which are defined as those that accomplish the required functionality without the added functionality and extensibility provided by design patterns. This study tested the hypothesis that design patterns are more maintainable than other solutions. The simplicity of the design pattern solutions should intuitively lead to less complexity and less maintenance time. The results varied for different design patterns. Notably the Observer and Decorator patterns had less time spent on maintenance than other non-pattern solutions. This supports the hypothesis. Patterns such as Visitor were inconclusive.

Vocak, et. al. [73] extended the work by Prechelt by using an actual development environment with an increased data set. The results also showed differences between patterns. An interesting result indicated that the Visitor and Composite patterns actually had a negative effect on maintenance time. This supports the rejection of the hypothesis that design patterns take less time to maintain.

Research on evolution of design patterns has been sparse. A common goal of empirical research is to test hypotheses about the positive effects of design pattern usage. A common intuition is that design pattern classes should be less change prone because they provide developers with an interface which is extended through specialization. Therefore classes are added, and existing classes are unchanged. Empirical research does not support this intuition. Studies have shown that design pattern classes are actually more change prone than their counterparts [11], [12]. The hypothesis that design pattern classes will have a lower defect rate because they are used ―properly” within the design has also been tested by Vokac [72]. Some design patterns had lower defect rates than non-pattern classes, but some patterns did not; these results tend to be inconclusive.

2.4 Design Pattern Decay

The study of design pattern decay is still in its infancy. Seminal research by Izurieta and Bieman [41] analyzed how design patterns can evolve in a way that has negative effects on testability and adaptability. In further studies they gather empirical data to characterize and formally define design pattern decay [42], [39]. This section provides a summary of design pattern decay and its components as well as a review of empirical results from the literature.

2.4.1 Definitions

Izurieta [39] defines two types of design pattern decay, *rot* and *grime,* but leaves open the possibility that other types exist. Rot describes violations to the structural characteristics of the design pattern while grime describes artifacts that are irrelevant to the design pattern. Rot occurs when the structure of the design pattern ceases to exist as a result of changes. The design pattern will cease to be conformant after rot has occurred. Near-instances of patterns occur when rot affects the pattern in such a way that it nearly conforms to its role-based metamodeling language (RBML) characterization, which is described further in section 3.1. Design pattern rot causes the design pattern to lose key structural characteristics and the pattern becomes either a near-instance or does not resemble a pattern at all.

Grime occurs when extraneous features are added to the design pattern while still leaving the structure of the pattern intact. These features are not specified in the formal definition of the design pattern. Additions are classified as grime because they are not necessary for the function of the design pattern. It is important to note that some additions may be necessary to fulfill requirements of the overall design, but they are defined as grime from the point of view of the design pattern realization because they are not part of the formal definition that describes the pattern. Grime is defined by three sub-categories: *class* grime*, modular* grime, and *organizational* grime.

Class grime refers to changes within the classes that are components of a design pattern realization. Classes are defined by their internal attributes and methods. Some attributes and methods are necessary for a class to play a role in a design pattern. Any method or attribute that is added over the lifespan of the pattern that is not required by the design pattern contributes to class grime.

Modular grime occurs when coupling increases for classes involved in design pattern realizations. Specifically, new relationships that occur over the lifetime of the pattern realization that do not play a role in the design pattern contribute to modular grime. Izurieta notes that modular grime reduces modularity by increasing coupling between the design pattern realization and its environment. The design pattern encompasses a module of functionality that should be loosely coupled to other modules. In general, increases in coupling also increase complexity which makes the code more difficult to understand and maintain.

Organizational grime refers to the logical and physical organization of design pattern members. Design pattern classes can be members of different logical groupings such as namespaces and packages. As new classes play roles in the pattern the number of packages involved in the pattern realization may increase. This makes maintenance more complex because the classes are spread over a large range of packages. Pattern classes are also organized physically in files in the file system. The addition of physical files is a consequence of pattern extension by the addition of concrete classes. An increase in the size of pattern classes without increases in physical file counts is a sign of organizational grime.

2.4.2 Empirical Research

Empirical research on design pattern decay is recent. A pilot study by Izurieta and Bieman [41] in 2008 examined the Visitor, State, and Singleton patterns for evidence of structural changes (rot) as well as the addition of artifacts not related to patterns (grime). Preliminary results indicated that patterns kept their structure throughout the evolution of the software. There were no signs of rot. Increases in coupling of pattern classes indicated the presence of modular grime.

In another study [42] Izurieta and Bieman test the effects of grime buidup on system testability. They hypothesize that increases in grime cause increases in complexity and therefore increases in test requirements. The results confirm the hypothesis that as modular grime builds, the number of test requirements increase. The authors also found examples of anti-patterns that decrease testability. The authors note that the extensibility of the Visitor pattern (through inheritance) opens the door to the formation of anti-patterns.

Izurieta published empirical results from an all-encompassing study on design pattern decay [39]. The study investigates three open source systems for the appearance of rot as well as all types of grime. It also examines correlations between design pattern decay and testability and adaptability. Results support earlier research. Modular grime does buildup through the evolution of the design patterns. This supports the notion that modular grime is the most significant form of design pattern decay.

Research on design pattern decay is new, however current research suggests that modular grime requires further investigation. The research in this thesis accomplishes two goals. It investigates modular grime in new systems at a finer granularity and it provides validation of previous results.

3. MODULAR GRIME TAXONOMY

We use role-based metamodeling language (RBML) to formally define and characterize design patterns. Candidate patterns are compared to RBML diagrams in order to determine conformance.

Modular grime is defined as the non-conforming coupling of objects that play a role in a design pattern realization. Relationships that involve pattern classes but do not conform to the RBML pattern definition are classified as grime. Section 3.1 gives a brief description of how RBML is used to determine conformity of design pattern realizations with RBML relationships and classifiers. Section 3.2 describes the taxonomy criteria.

3.1 RBML and Pattern Conformity

Not all relationships that develop over the lifetime of a pattern realization contribute to grime. Patterns can be used or extended in ways intended by the original designer. Use dependencies are one compatible explanation for usage of design pattern classes. Use dependencies occur when the system uses a design pattern realization in a way that is consistent with the pattern‘s intent. An example is increased usage of the Singleton pattern. A class that uses a singleton pattern will generally create some level of coupling between itself and the Singleton. This increases the **Ca** count of the Singleton realization, but the usage is not classified as grime because the Singleton class is meant to be used in this way. A counterexample can occur in the Factory pattern where abstract classes are meant to be used as the interface. A use dependency involving concrete pattern classes is not correct usage as defined by the pattern. In this case the use dependency is considered grime and not usage.

Patterns may also be extended through specialization. Patterns are abstractions that allow the functionality of the pattern to be extended with the addition of new concrete classes. This allows abstract classes and interfaces to remain unchanged. The addition of new classes through pattern extension increases coupling (in the form of inheritance) and does not contribute to modular grime.

The role-based metamodeling language is a specialization of the UML metamodel that provides a practical mechanism to differentiate between usage, extension, and grime. Roles are the primary object in RBML diagrams; they are defined using constraints. The objects in a UML class diagram can be mapped to the roles in an RBML model when they fall within the constraints of the role. RBML Structural Pattern Specifications (SPSs) are used to define design patterns by defining all of the roles necessary to the pattern. A complete mapping of UML elements to SPS roles in a design pattern definition reveals a conforming pattern realization. RBML also uses multiplicities of classifiers to allow for optional classes.

When a design pattern realization is already identified by its conformance to an SPS, the SPS is used to determine if new relationships (relationships added in subsequent releases) are conformant with the RBML pattern definition. In essence, the SPS is used to determine if a new coupling is due to usage or extension. Otherwise, the coupling increases the modular grime of the realization.

3.2 Taxonomy Criteria

Evidence suggests that design pattern decay in the form of modular grime does occur [39]. In a pilot study [66] of a single system, Schanz and Izurieta explore modular grime by classifying it into 6 distinct categories. The study provides an extension to previous research on design pattern decay. It expands the definition of modular grime by proposing an initial taxonomy.

The taxonomy classifies the couplings that define modular grime using three characteristics. The characteristics are *strength, scope*, and *direction*. The strength of a relationship describes the level difficulty in removing a relationship and it is measured using an ordinal scale [59]. Relationships with greater strength are more difficult to remove. Scope refers to the boundary that exists between coupling relationships as defined by the set of design pattern classes compared to the set of all other classes. Direction is determined by the originating class that establishes the relationship.

3.2.1 Coupling Strength

The strength of a coupling relationship has been defined differently by several authors [15], [38], [57]. Under our current taxonomy, the strength of a coupling relationship can fall into two distinct categories. *Persistent* relationships occur when two objects are coupled throughout their lifetime. This happens when class A has an attribute of type B. The coupling between A and B is deemed strong because class A has a reference to class B throughout its lifetime in memory. *Temporary* relationships exist when a method of class A requires access to class B. This occurs when said method has a parameter, return value, or local variable of type B. Temporary relationships are not considered as strong because they are transient.

Recent research by Briand, et. al. [14] uses a similar classification system. They define a Class-Attribute (CA) interaction as the equivalent of a persistent coupling relationship. Class A and B are CA coupled if class A has an attribute of type B. A Class-Method (CM) interaction is equivalent to temporary coupling as defined above. Class A has a CM interaction with class B if a part of the method signature of class A (parameters and return type) is of type B. Temporary couplings also include local variables.

Formally, the sets **Persistent** and **Temporary** are ordinal because although relationships with differing strengths can be ordered, the difference in strength cannot currently be quantified. Each set is defined as follows.

Persistent = {class_attribute}

Temporary = {method_local_variable, method_return_value, method_parameter_value}.

The sets are disjoint and each displays its members in increasing order of strength. Though not part of this research, each set is open to the addition of other less frequent coupling types such as globally shared data. All relationships studied in this thesis are considered persistent or temporary. Briand, el. al. [14] provide a summary of other coupling types.

3.2.2 Coupling Scope

Each grime relationship affects at least one design pattern. We define the scope of a coupling as *internal* or *external*. A grime relationship is internal if it involves two design pattern classes. A grime relationship is external if it involves a design pattern class and an external class that is not involved in the design pattern.

Formally, let ***P*** be a specialization of RBML that describes a design pattern. The set of classes that describes ***P*** is denoted by ***C(P)*** and the set of relationships is denoted by ***R(P)***. The **order** of a pattern is defined as the total number of classes in ***P***, and is denoted by **|*C(P)*|**, and the **size** of a pattern is defined as the total number of relationships in ***P***, and is denoted by **|*R(P)*|**. A valid classifier is defined as a class ***c*** or relationship ***r*** allowed by the RBML of the pattern. Valid classifiers in a design are seminal or evolve as permitted by the extensibility rules of the pattern's RBML. Thus, a relationship $\mathbf{r}_{\mathbf{ci,\,cj}}$

is **internal** iff $\forall$ i, j : *c*i, *c*j $\in$ ***C(P)***

is **external** iff $\exists$ i, j : *c*i $\in$ ***C(P)*** $\wedge$ *c*j $\notin$ ***C(P)*** $\wedge$ i $\neq$ j

3.2.3 Coupling Direction

The direction of coupling is determined by the originating class. Import and export coupling are commonly used to differentiate direction [14], [15]. Import coupling occurs when a class uses another class, i.e., it establishes the initial relationship. Export coupling defines the alternative case. Coupling direction is defined using the terms afferent (Ca) and efferent (Ce) [14]. The afferent coupling of a class **c** increases when a relationship appears which originates in another class **c'**, where **c'**$\neq$ **c**. Efferent coupling for class **c** increases when a new relationship appears which originates in **c**. These terms are defined for a single class while a pattern realization normally contains a set of classes.

Afferent and Efferent coupling for design pattern realizations differ because all pattern classes function as the classes of interest. Efferent relationships originate in any class that is part of the pattern of interest and afferent relationships originate in any non-pattern class. The afferent coupling count of a design pattern realization increases when an external class develops a dependency with one of the classes in the set of classes that belongs to a design pattern realization. The class becomes coupled to an external class, and the coupling originates in the external class. The efferent coupling count of a design pattern realization increases when a class in a design pattern realization develops a dependency on an external class. The relationship originates from within the pattern. Coupling direction applies to all relationships regardless of coupling strength.

3.3 Taxonomy Classes

The three coupling criteria defined in section 3.2 are used to create a taxonomy of modular grime. We use a simple method to create the taxonomical classes. Each class is determined by a unique combination of coupling strength, scope, and direction.

Relationships that have internal scope, by definition, only couple design pattern classes. Therefore, external classes are not involved. Coupling direction is defined as afferent if the relationship originates in a class external to the pattern or efferent if the relationship originates within the pattern. Coupling direction is trivial when scope is internal because there is no possibility that the coupling originates anywhere but within the pattern. This eliminates two possible classifications from the grime taxonomy, Persistent Internal Afferent Grime and Temporary Internal Afferent Grime. Figure 1 shows how we use strength, direction, and scope to classify modular grime. The leaf node depict all six modular grime classifications.

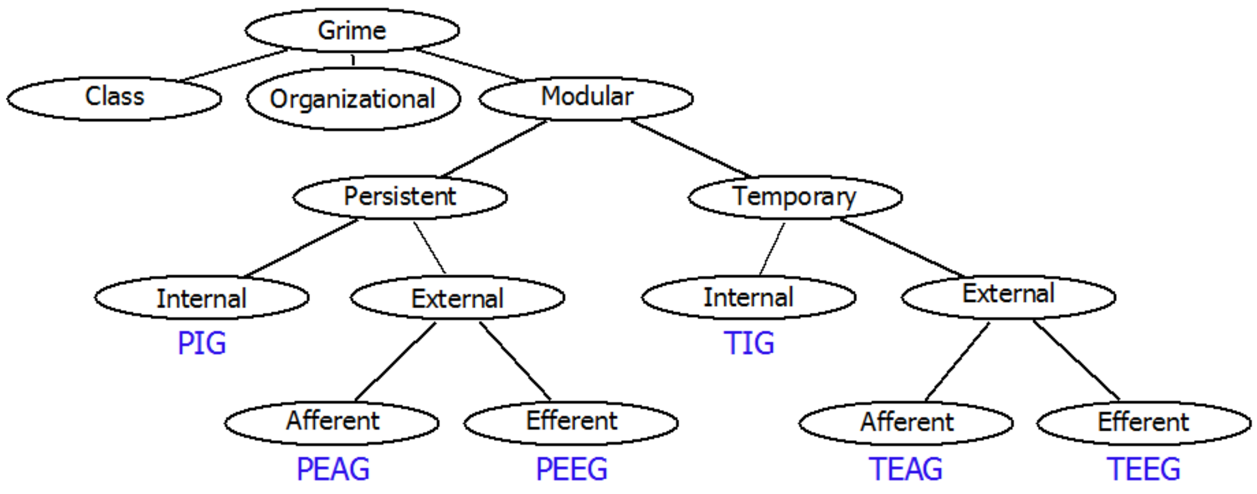


Figure 1. Grime Taxonomy [66].

3.3.1 Persistent External Afferent Grime (PEAG)

This is the set of all invalid relationships between a pattern and a non-pattern class where grime originates in a class ∉ **C(P)**. These relationships are similar to those in the PEEG category except that the external coupling is afferent, thus increasing the responsibility of the pattern realization and making the refactoring significantly more difficult. Given an instance of a relationship **r**, PEAG is observed when **r** ∈ **Persistent** and **Ca** increases when **r** is invalid.

3.3.2 Temporary External Afferent Grime (TEAG)

This is the set of invalid temporary relationships between pattern classes and external pattern classes where grime originates in a class ∉ **C(P)**. TEAG is observed when **r** ∈ **Temporary** and **Ca** increases when **r** is invalid.

3.3.3 Persistent External Efferent Grime (PEEG)

This is the set of invalid persistent relationships between pattern classes and external pattern classes. The persistence of the relationships makes refactoring difficult, yet direction of coupling simplifies refactoring because dependencies on external classes can be easily removed from the originating internal classes. PEEG is observed when **r** ∈ **Persistent** and **Ce** increases when **r** is invalid.

3.3.4 Temporary External Efferent Grime (TEEG)

This is the set of invalid temporary relationships between pattern classes and external pattern classes. Refactoring is simplified by the weaker coupling strength and, similar to PEEG, the external relationships are comparatively easier to refactor. TEEG is observed when **r** ∈ **Temporary** and **Ce** increases when **r** is invalid.

3.3.5 Persistent Internal Grime (PIG)

This is the set of all invalid relationships that strongly couple two pattern classes ∈ **C(P)**. The persistence of these relationships makes grime removal (refactoring) more difficult when compared to temporary relationships. PIG is observed when **r** ∈ **Persistent** and the size of the pattern **|R(P)|** increases when **r** is invalid.

3.3.6 Temporary Internal Grime (TIG)

This set contains invalid temporary relationships involving two pattern classes ∈ **C(P)**. Relationships are similar to those described by the PIG set except they are easier to refactor due to weaker coupling strength. TIG is observed when **r** ∈ **Temporary** and **|R(P)|** increases when **r** is invalid.

4. STUDY METHODOLOGY

This section describes the study methodology in detail. Section 4.1 describes the software and its selection criteria. Section 4.2 describes tools used to mine for realizations of design patterns and gather grime data. Section 4.3 describes in detail how queries were built in order to identify specific patterns and count grime relationships from said patterns. Section 4.4 describes six hypotheses we test in the study.

4.1 Software Studied

Three criteria were used to select software. First, we only explored open source software. Commercial software can be difficult to find because organizational cooperation can be difficult to obtain due to proprietary intellectual property. Second, all software chosen was written in Java [43]. This was a necessary condition because the tools we use to gather patterns and mine data are specific to Java. Third, we required that all studied software span at least 8 versions. Whilst this number was arbitrarily chosen, it provided us with an acceptable sample size for each software product under study.

4.1.1 Apache Tomcat

Apache Tomcat [5] is an open source web server implemented in Java. It was created and is currently maintained by the Apache Software Foundation [4]. Apache Tomcat is different than the traditional Apache web server [6] in that it provides a unique web server implementation in Java that can be incorporated with other Java projects.

Table 1. Apache Tomcat release information by month/year.

	12/06	03/07	05/07	08/07	02/08	06/08	06/09	01/10	03/10
Version	6.0.0	6.0.10	6.0.13	6.0.14	6.0.16	6.0.18	6.0.20	6.0.24	6.0.26
# of Classes	1,520	1,549	1,600	1,608	1,622	1,620	1,628	1,648	1,649
KLOC	315	318	322	323	325	326	329	334	335

Table 2. Apache Tomcat number of design patterns studied in every release.

Pattern	Singleton	Adapter	Façade	Factory	Observer
# of Realizations	3	2	2	7	4

The initial version was released in 1999 and it has been in active development for more than 10 years. The longevity of the Tomcat project along with the popularity of Apache (especially the traditional Apache web server) makes Tomcat a viable subject for study. We studied stable releases from 2006 to 2010. Table 1 contains release information about the versions studied and table 2 displays information about the design patterns mined.

4.1.2 Dr. Java

Dr. Java [26] is an integrated development environment (IDE) for Java developers. It is simple and lightweight and primarily used for basic Java development. Dr. Java is written in Java and is available on several platforms. It is an open source system and it has been developed and maintained at Rice University for more than 8 years. We gathered data from 9 releases between 2004 and 2009. Detailed release information is given in tables 3 and 4.

Table 3. Dr. Java release information by month/year.

	03/04	**07/04**	**02/06**	**04/06**	**01/08**	**04/08**	**07/08**	**01/09**	**10/09**
Version	0037	2332	1750	1814	1942	4436	4592	4756	5122
# of Classes	921	1,039	1,266	1,385	1,613	1,730	1,847	1,862	1,962
KLOC	109	119	122	123	127	137	145	143	152

Table 4. Dr. Java number of design patterns studied in every release.

Pattern	Singleton	Strategy	Iterator	Visitor	Factory	State	Observer
# of Realizations	13	1	1	2	3	2	3

4.1.3 Apache Derby

Apache Derby [3] is a Java implementation of a relational database. Its implementation uses industry standards such as SQL [69] and JDBC [44]. Apache Derby is an open source system and it can be easily used in conjunction with other Java projects. It has a 2 MB disk-space footprint [3], has been developed by the Apache Software Foundation since 2004, and is still in active development. We studied 9 releases from 2005 to 2009. Tables 5 and 6 provides detailed information.

Table 5 Apache Derby release information by month/year. Version is 10.x.

	08/05	11/05	06/06	10/06	12/06	04/08	05/08	09/08	04/09
Version	1.1.0	1.2.1	1.3.1	2.1.6	2.2.0	3.3.0	4.1.3	4.2.0	5.1.1
# of Classes	1,803	1,810	1,831	2,141	2,139	2,491	2,756	2,787	3,114
KLOC	676	679	689	785	785	887	944	949	1,034

Table 6. Apache Derby number of design patterns studied in every release.

Pattern	Adapter	Builder	Factory	Strategy	Visitor
# of Realizations	1	1	12	1	1

4.2 Tools Used

We used a wide variety of tools to gather and analyze data. Two separate and distinct processes were used: 1) finding design patterns and 2) gathering data from them. Tools for finding patterns include [21], [22], [34], [67]. Pattern finding software generally produces high counts of false positives unless each pattern realization is very specific [7] thus eliminating other pattern variations from contention. Other pattern mining algorithms find pattern realizations based on common characteristics of observed realizations [2] (e.g. by convention most Singletons have a static local variable named singleton), which could lead to false positives. For these reasons we use tools to provide an initial hint to the location of a pattern. The pattern is then manually verified against its RBML. Once a pattern is identified, our grime definitions require that we gather specific grime coupling data from each realization. This requires a finer granularity than most common metric gathering tools provide. We used tools that allowed us to specify exactly how to mine grime data. The following tools were used:

4.2.1 Eclipse

Eclipse [27] is a well known IDE that provides support for several different languages. It is most commonly used for Java development. It is open source and uses a plug-in driven architecture to provide a wide variety of extra functionality. We use Eclipse to initially analyze source code from the software studied. In addition, many eclipse plug-ins [46], [47], [61] are used to find design pattern realizations and gather data.

4.2.2 Design Pattern Finder

Design Pattern Finder [24] is a simple and fast Windows based tool used to find design pattern realizations in Java projects. It searches Java files for common signs of design pattern realizations. This is mostly done by semantically matching class names with pattern names. This is acceptable for our use because we focus on intentional pattern realizations. We use Design Pattern Finder as an initial scouting tool for finding pattern realizations. It provides a starting point in the search for design patterns. Manual validation is required to remove false positives.

4.2.3 ObjectAid UML

ObjectAid [61] is an Eclipse plug-in that provides UML class diagrams within the Eclipse IDE. It allows users to generate class diagrams from existing source code files and packages. We primarily use this tool to generate quick diagrams for groups of files that are suspected to be involved in a design pattern realization. Visualizations of pattern realizations allow us to manually verify pattern conformance to an SPS quickly.

4.2.4 Browse By Query

Browse By Query (BBQ) [16] is an Eclipse plug-in that provides a query language for browsing source code. Specifically, BBQ stores source code information in an Object Oriented Database and allows the user to query the database using English-like queries. We used BBQ to gather pilot data during the initial stages of our study. The flexibility of the query language was essential because it allowed us to explicitly define coupling as it contributes to grime buildup. While other metric gathering utilities [45], [46] reveal coupling or dependency information, BBQ allowed us to be more precise in our measurements.

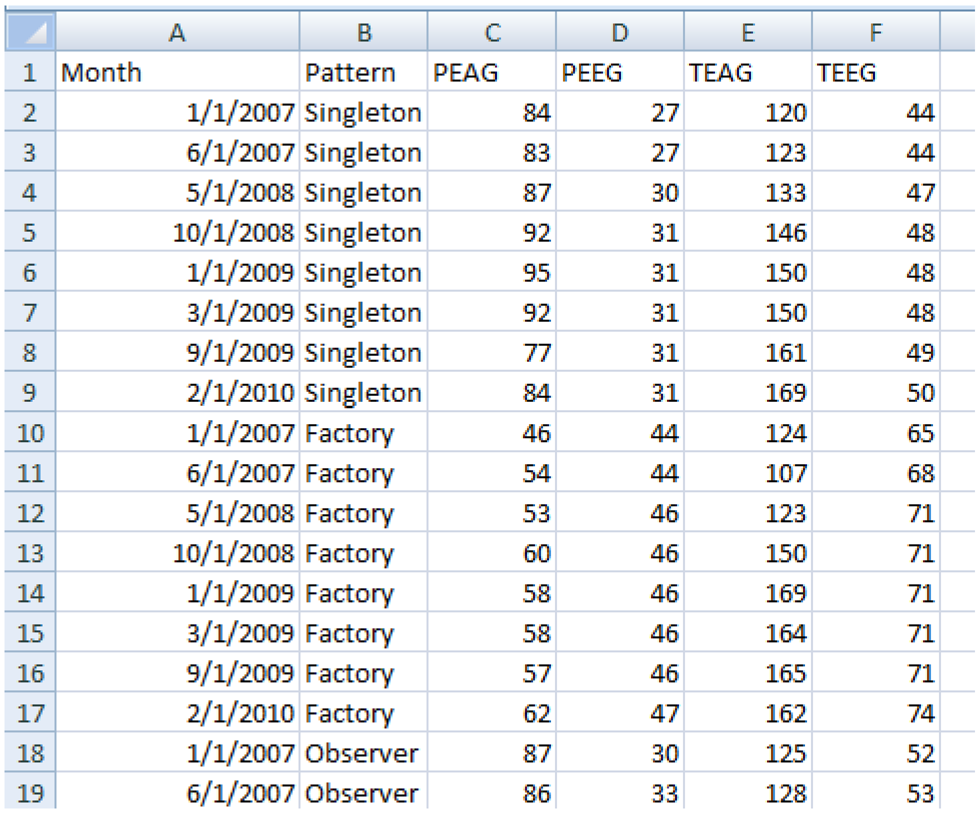
4.2.5 JQuery

JQuery [47] is a source code query tool. It is an Eclipse plugin that allows users to query properties of source code using a Prolog-like language called TyRuba [71]. JQuery creates a database of facts from the Abstract Syntax Tree of a Java project. Users can query the database using predicates and retrieve a wide variety of information. JQuery was necessary for this research because it is more robust and stable than BBQ. It also allowed us to eliminate some construct validity problems that we encountered in our pilot research. JQuery increases construct validity because it allows us to differentiate between internal and external grime. BBQ did not provide such operators, causing inaccuracies in the aggregation of grime counts for internal and external measures. The identification of measures that correctly represent the subject of study improves construct validity.

4.2.6 Dynamic Chart

Dynamic Chart (see appendix C), written in Java, is a tool we developed to assist with the graphing of modular grime data. The taxonomy of modular grime motivated us to create a graphing tool that would allow us to visualize individual categories as well as combinations of modular grime categories. Dynamic chart allows a user to view data dynamically without having to reload and modify the input data. The following sections describe and demonstrate the functions of Dynamic Chart.

4.2.6.1 File Input. Data is entered (currently only in .csv format) as a time series containing one or many variables. The time series consists of a series of release dates. There may also be different groups of time series if the user wishes to import several groups of data at a time. Figure 2 shows an example input file with column ‗A' containing dates in a time series.



	A	B	C	D	E	F
1	Month	Pattern	PEAG	PEEG	TEAG	TEEG
2	1/1/2007	Singleton	84	27	120	44
3	6/1/2007	Singleton	83	27	123	44
4	5/1/2008	Singleton	87	30	133	47
5	10/1/2008	Singleton	92	31	146	48
6	1/1/2009	Singleton	95	31	150	48
7	3/1/2009	Singleton	92	31	150	48
8	9/1/2009	Singleton	77	31	161	49
9	2/1/2010	Singleton	84	31	169	50
10	1/1/2007	Factory	46	44	124	65
11	6/1/2007	Factory	54	44	107	68
12	5/1/2008	Factory	53	46	123	71
13	10/1/2008	Factory	60	46	150	71
14	1/1/2009	Factory	58	46	169	71
15	3/1/2009	Factory	58	46	164	71
16	9/1/2009	Factory	57	46	165	71
17	2/1/2010	Factory	62	47	162	74
18	1/1/2007	Observer	87	30	125	52
19	6/1/2007	Observer	86	33	128	53

Figure 2. Example input file for Dynamic Chart

Each time series is repeated with different groupings for each design pattern. In this example, there are four variables for each point in time. The variables are PEAG, PEEG, TEAG, and TEEG. In this example each pattern category has an equivalent set of dates associated with it. This is not a requirement. Note that PIG and TIG are not included in this example data set. They could easily be added in additional columns.

4.2.6.1 Features. A detailed description of features is included in the Dynamic Chart documentation in Appendix A. Current features supported by Dynamic Chart include:

Dynamic data charting – Dynamic Chart creates a linear combination of all input variables at each point in time. Each point is described mathematically by the following linear equation.

$$G_t = \alpha_1 PEAG_t + \alpha_2 PEEG_t + \alpha_3 TEAG_t + \alpha_4 TEEG_t$$

Where G_t represents a singular grime count at time t. $PEAG_t$, $PEEG_t$, $TEAG_t$, and $TEEG_t$ are the actual grime counts for each type of modular grime at point t and α_1 through α_4 are the linear coefficients that are independent of time. Each point in time has one corresponding value per group and there exists a G_t for each group at each time t. The user has control over the coefficients for each variable in the form of sliders. As the sliders are changed, the chart is dynamically updated to show the resulting change in G at all points in time.

Coefficients α_1 through α_4 allow users to adjust how much each type of grime contributes to the overall measure of grime. For example, setting all coefficients to 1 would result in the total grime count. Alternatively, setting α_1 to 1 and setting α_2, α_3, α_4 to 0 would display grime counts for PEAG only.

Data grouping – Data can be input in groups. Groups are displayed initially in separate tabs, one for each group. The user has the ability modify groupings as needed. This allows users to view and compare different series of data. For example, the user may group the Singleton and Factory data into a new group called the ~~–c~~reational pattern" group. The data for both Singleton and Factory will now be displayed on the same chart and the user can examine differences between the two patterns. Users can add other patterns to the creational data group or split Singleton and Factory back into separate groups.

Data modification – While data groupings allow users to view different data on the same chart, there is also an option to aggregate different data sets into a singular set. Extending the example above, the user could aggregate the Singleton (S) and Factory (F) results into one set of results. This can be shown mathematically as follows.

$G(S)_t = \alpha_1 PEAG(S)_t + \alpha_2 PEEG(S)_t + \alpha_3 TEAG(S)_t + \alpha_4 TEEG(S)_t$; $G(S)_t$ represents the Singleton data

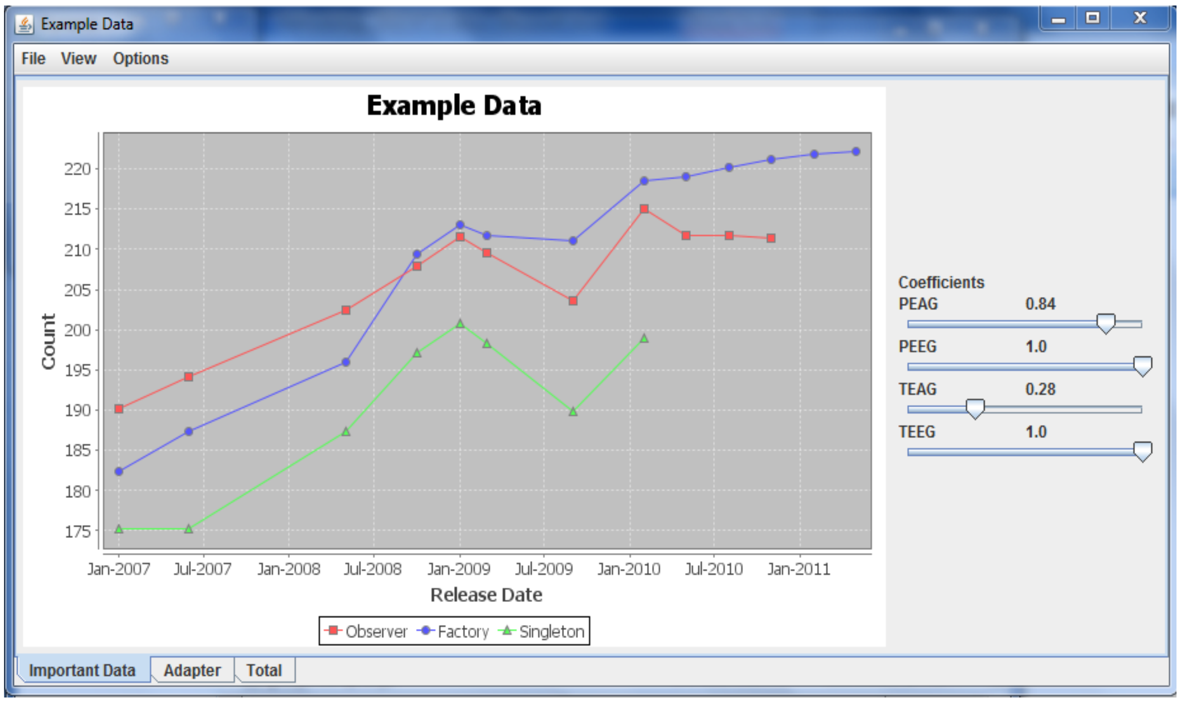
$G(F)_t = \alpha_1 PEAG(F)_t + \alpha_2 PEEG(F)_t + \alpha_3 TEAG(F)_t + \alpha_4 TEEG(F)_t$; $G(F)_t$ represents the Factory data

$G(F, S)_t = \alpha_1[PEAG(F)_t+PEAG(F)_t] + \alpha_2[PEEG(F)_t+PEEG(F)_t] + \alpha_3[TEAG(F)_t+TEAG(F)_t] + \alpha_4[TEEG(F)_t+TEEG(F)_t]$

$G(F, S)_t$ represents the resulting data set after the Singleton and Factory data are aggregated. To aggregate the data for multiple groups, the individual data points for each variable at each point in time are summed for all of the groups. Dynamic Chart allows the user to aggregate the data for any number of initial data groups. Data can also be removed from the aggregated sets as necessary.

Data extension – Dynamic chart provides a simple prediction mechanism to facilitate extrapolation of data. Users can create *data extensions* which determine future points in time to be predicted. When a data extension is set a curve fitting algorithm is used to predict the points in the extension. The curve fitting algorithm generates several regression models (linear, quadratic, polynomial) from the initial data and chooses the one with the least squared error. This feature can be performed individually for each time series. Extensions can be set and changed at any point during the operation of Dynamic Chart.

4.2.6.2 Demonstration. An example data set is displayed in Dynamic Chart in figure 3. The tabs in the lower left allow users to switch between each group of data. The sliders in the right hand side of the window allow users to modify the linear combinations of the modular grime variables. In the current display tab the user has combined the Singleton, Factory, and Observer groups into one display group called ―Important Data”. Predictions have been added for both the Observer and Factory Timer series.



Coefficients

PEAG	0.84
PEEG	1.0
TEAG	0.28
TEEG	1.0

Figure 3. A Dynamic Chart display

4.3 Data Gathering

This section describes the process and tools used to mine for modular grime in selected design patterns. 4.3.1 describes our initial pilot study. Sections 4.3.2 and 4.3.3 describe how we used JQuery to mine for grime and remove threats to construct validity identified during the pilot study.

4.3.1 Pilot study

After defining the modular grime taxonomy, we manually searched open software repositories for software to mine for grime data. The main source of open software used was SourceForge [68]. We selected Vuze [74] because the number of downloads was high (more than 100,000 downloads for a recent version from one of many download mirrors) and it was written in Java. Additional systems, as described in section 4.1, were added after the pilot study helped validate the proposed taxonomy.

The criteria for selecting Vuze in the pilot study included evidence of modular grime buildup as defined by the taxonomy definitions described in section 3.3. Several metrics tools were initially used to gather coupling information; however they proved unacceptable for a variety of reasons. JHawk [46] only gathered information at higher organizational levels (i.e. packages or namespaces) lacking the necessary granularity needed to study grime buildup. We specifically needed to record coupling counts of design pattern classes. JDepend [45] gathered dependency information, but did not specify how a dependency was defined. Our taxonomy definitions use a specific definition of coupling that requires finer detection granularity.

The limitation of BBQ was that it did not allow us to differentiate between **internal** and **external** grime. Although BBQ allowed us to gather coupling data for the classes that compose the design pattern realizations, it did not allow us to specify *where* the coupling originated from. For example, we gather data about persistent grime for all pattern class realizations when we gather PEAG data. We could not determine the source of the grime and therefore differentiate between PEAG and PIG. BBQ aggregated the counts for PEAG and PIG. Similarly, we could not differentiate between internal and external grime for temporary couplings. Therefore, TEAG and TIG were also aggregated.

Another issue emerged from our definition of temporary grime. Temporary grime is composed of three types of coupling: method return types, parameter types, and local variables. BBQ gives us no access to the local variables of methods. This left us with the possibility that measurements of temporary grime were understated.

4.3.2 Gathering data using JQuery

JQuery solved all the threats to validity we encountered with BBQ. Like BBQ, it is also a query tool, but it is more flexible. It stores all of the properties of, and relationships between, Java objects in a factbase. We simply queried the factbase for each specific relationship that the taxonomy defines as grime. JQuery uses a Prolog like language called Tyruba [71] to query the factbase generated from source code. The queries we used allowed us to determine the source of grime, solving the problem of differentiating between internal and external grime. JQuery also generates facts about local variables within methods which allowed us to consider all types of temporary grime. We used JQuery to gather data from all versions in every system we studied. The next section details the queries we used to gather data for each category of grime in the taxonomy.

4.3.3 JQuery Queries

In the Tyruba [71] language used by JQuery, variables are preceded by a question mark and denote a set of objects. Predicates can initially determine a set of objects for a variable or narrow the scope of the variable. For example, the basic query

```
    package(?P),name(?P,Base)
```

Creates a variable ‗?P' that contains all packages in the factbase. The second predicate eliminates packages that are not named ‗Base'. Commas are used to represent conjunction while semicolons are used to represent disjunction.

We use a top down approach to gather grime data. Starting with all of the classifiers in the source, we gather all coupling relationships as defined by coupling strength. We then eliminate relationships that don't belong depending on where the coupling originates and ends. For example, when calculating afferent coupling, eliminate all relationships that originate in pattern classes. After the query is executed, any relationship that occurs because of correct usage must be manually removed from the results. Data is gathered for each taxonomical category as follows:

PEAG: The full query is

```
allClassifiers(?P,?C),

attributeTypesOfClassifier(?C,?T),

(name(?T,PatternClass1);...;name(?T,PatternClassn)),
(NOT(name(?C,PatternClass1);...;name(?C,PatternClassn)))
```

There are four main parts in the query connected by conjunctions. Each can be described in more detail.

allClassifiers(?P,?C): This predicate gathers all package objects in ?P and all classifiers in ?C. It is a custom predicate defined for our objective. The full code can be found in the appendix.

attributeTypesOfClassifier(?C,?T): This predicate gathers the attributes for each classifier in the set ?C obtained by allClassifiers(?P,?C). It is another custom predicate. We now have a set of classifiers (?C) and a set of types (?T) for all of the types of attributes in the classifiers. These are persistent relationships that originate in a classifier in the set ?C and end at a type ?T.

(name(?T,PatternClass1);...;name(?T,PatternClassn)): This set of predicates eliminates all relationships that do not end at a pattern class. There is a name() predicate for each class in the design pattern realization from Class1 to Classn. The ellipsis denotes the name predicates included for all classes between Class1 and Classn.

(NOT(name(?C,PatternClass1);...;name(?C,PatternClassn))): This set of predicates eliminates all relationships that originate in a pattern class. Any relationships that originate in a pattern class would be internal in this case. There is a name() predicate for each pattern class.

All of the queries take this general form. ?C contains the originating classifiers and ?T contains the types that denote where the relationship ends. The remaining queries follow.

TEAG:

```
allClassifiers(?P,?C),

temporaryTypesofClassifier(?C,?T),

(name(?T,Class1);...;name(?T,Classn)),

(NOT(name(?C,ClassA1);...;name(?C,ClassA2)))
```

This query is the same as the query for PEAG with one difference. The predicate **temporaryTypesofClassifier(?C,?T)** gathers ?T using all of the temporary relationships defined in section 3.2.1 instead of persistent attributes. This custom query can be found in the appendix.

PEEG:

```
allClassifiers(?P,?C),

attributeTypesOfClassifier(?C,?T),

(name(?C,Class1);...;name(?C,Classn)),

(NOT(name(?T,ClassA1);...;name(?T,ClassA2)))
```

This query looks similar to the preceding queries with one important difference. The NOT logic is reversed because the direction of coupling has changed. This query eliminates all relationships that end in a pattern class, and it accepts all relationships that originate in a pattern class.

TEEG:

```
allClassifiers(?P,?C),
temporaryTypesofClassifier(?C,?T),

(name(?C,Class1);...;name(?C,Classn)),

(NOT(name(?T,ClassA1);...;name(?T,ClassA2)))
```

TEEG is similar to PEEG with an alternative predicate used to find temporary relationships.

PIG:

```
allClassifiers(?P,?C),

attributeTypesOfClassifier(?C,?T),

((name(?C,Class1);...;name(?C,Classn)),

(name(?T,Class1);...;name(?T,Classn)))
```

The pig query uses two sets of name() predicates to confirm that both ends of the relationship are contained within the set of design pattern classes. This is the same as the PEEG query with the removal of a NOT clause.

TIG:

```
allClassifiers(?P,?C),

temporaryTypesofClassifier(?C,?T),

((name(?C,Class1);...;name(?CL,Classn)),

(name(?T,Class1);...;name(?T,Classn)))
```

TIG is equivalent to PIG with a change of predicates to differentiate for coupling strength.

4.4 Hypotheses

We gather data to investigate hypotheses for each category of the proposed grime taxonomy. The hypotheses are tested with data gathered for each modular grime category using the methods described in section 4. We test the following 6 hypotheses for each system studied:

$\mathbf{H}_{1,0}$: There is no evidence to suggest linear growth of PEAG counts over time.

$\mathbf{H}_{2,0}$: There is no evidence to suggest linear growth of TEAG counts over time.

$\mathbf{H}_{3,0}$: There is no evidence to suggest linear growth of PEEG counts over time.

$\mathbf{H}_{4,0}$: There is no evidence to suggest linear growth of TEEG counts over time.

$\mathbf{H}_{5,0}$: There is no evidence to suggest linear growth of PIG counts over time.

$\mathbf{H}_{6,0}$: There is no evidence to suggest linear growth of TIG counts over time.

The hypotheses are tested using statistical analysis. Each hypothesis is tested once per system using aggregated data for all design pattern realizations.

5. RESULTS

Data gathered to test all hypotheses are presented in this section. Data was collected for all six categories in the modular grime taxonomy. Grime types that show no discernable evidence of change as a pattern evolves are excluded in all figures for clarity. Table 7 shows the number of pattern realizations mined in each system and tables 8 through 10 display realization information about each pattern. In sections 5.1 through 5.3 we display raw data for Apache Tomcat, Dr. Java, and Apache Derby respectively. Section 6 contains discussion and analysis of the data.

Table 7. Aggregate number of design pattern realizations studied in each system

System	Pattern realizations
Apache Tomcat	18
Dr. Java	25
Apache Derby	16

Table 8. Apache Tomcat number of design patterns studied in every release.

Pattern	Singleton	Adapter	Façade	Factory	Observer
Realizations	3	2	2	7	4

Table 9. Dr. Java number of design patterns studied in every release.

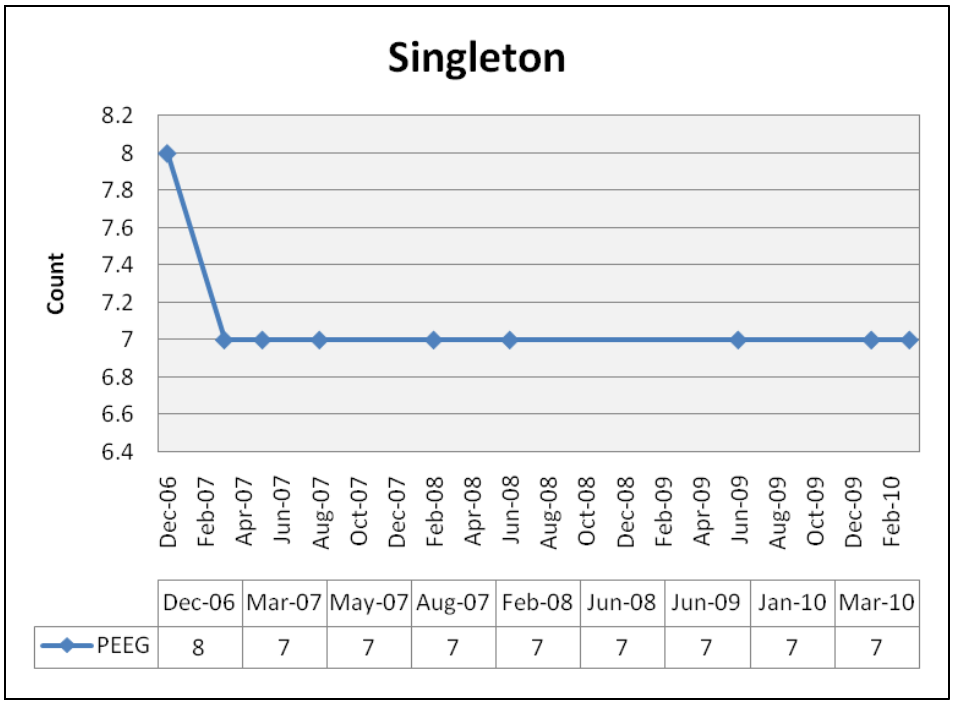
Pattern	Singleton	Strategy	Iterator	Visitor	Factory	State	Observer
Realizations	13	1	1	2	3	2	3

Table 10. Apache Derby number of design patterns studied in every release.

Pattern	Adapter	Builder	Factory	Strategy	Visitor
Realizations	1	1	12	1	1

5.1 Apache Tomcat

We mined 18 design pattern realizations of five distinct design patterns. Figure 4 displays the grime counts for all 3 Singleton pattern realizations. Overall, grime counts change very little and there is no internal grime. PEEG is the only type of grime that changes between releases.

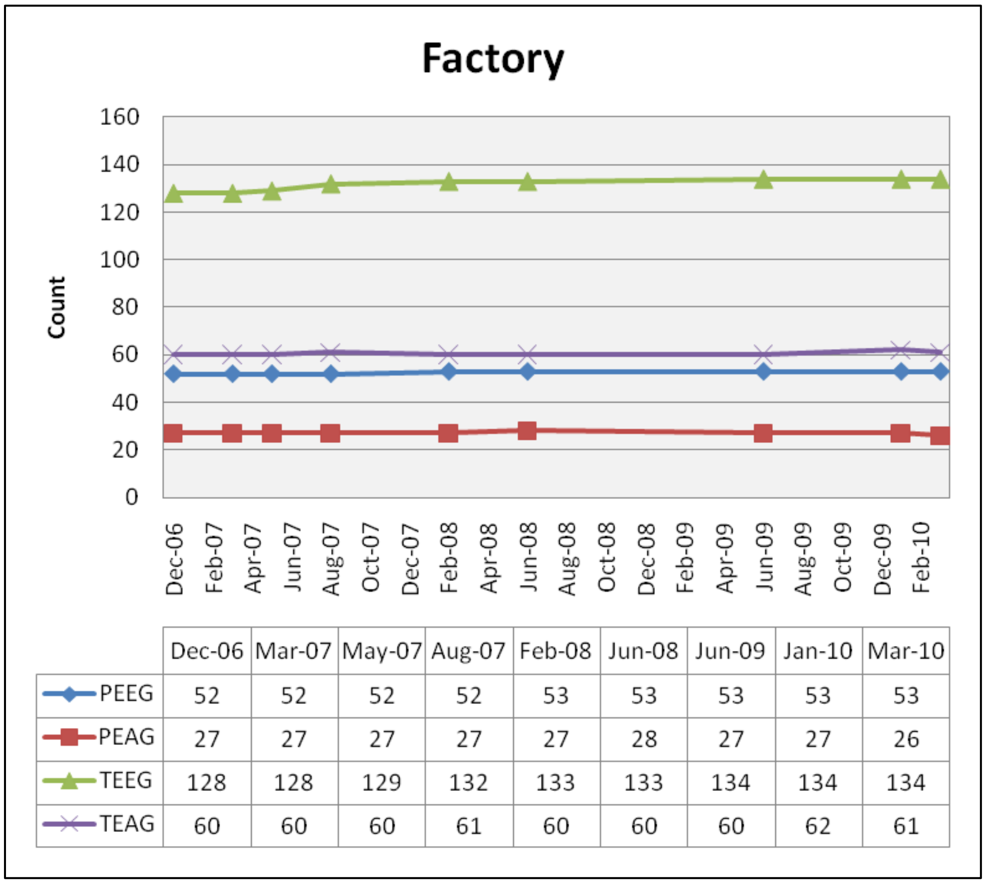


	Dec-06	Mar-07	May-07	Aug-07	Feb-08	Jun-08	Jun-09	Jan-10	Mar-10
PEEG	8	7	7	7	7	7	7	7	7

Figure 4. Aggregate PEEG grime count for 3 Singleton pattern

Figures for the Adapter and Façade pattern realizations are excluded as the data does not change at any point over the six releases that comprise the period of study. The grime counts for the Adapter pattern realizations are 0, 13, 5, 14, 0, and 0 for PEEG, PEAG, TEEG, TEAG, PIG, and TIG respectively. Grime counts for the Façade pattern realizations are 2, 1, 13, 1, 0, and 0 for PEEG, PEAG, TEEG, TEAG, PIG, and TIG respectively.

Data for the Factory and Observer patterns show more variation over time than the Singleton, Adapter and Façade patterns. Figure 5 shows results for the Factory pattern.

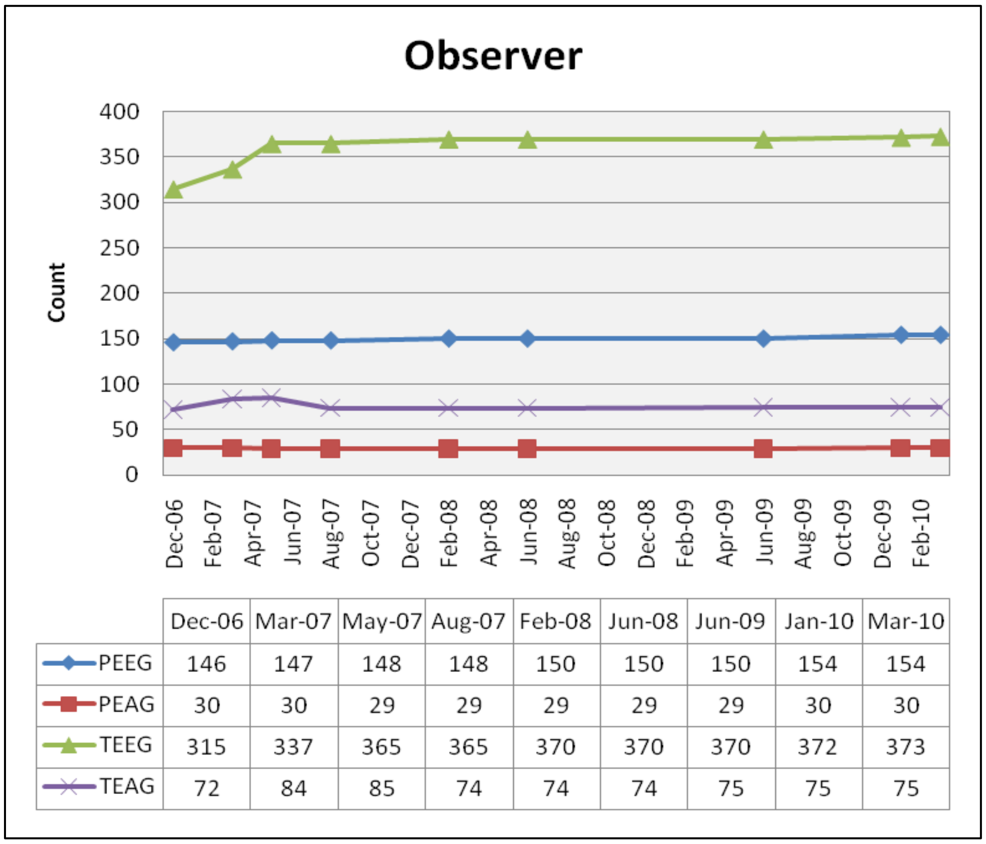


	Dec-06	Mar-07	May-07	Aug-07	Feb-08	Jun-08	Jun-09	Jan-10	Mar-10
◆ PEEG	52	52	52	52	53	53	53	53	53
■ PEAG	27	27	27	27	27	28	27	27	26
▲ TEEG	128	128	129	132	133	133	134	134	134
✕ TEAG	60	60	60	61	60	60	60	62	61

Figure 5. Aggregate grime counts for 7 Factory pattern realizations (Tomcat)

PEEG, PEAG, and TEAG counts show minimal changes; each varying by at most 2 grime counts during period of study. PIG and TIG counts are consistent throughout with values of 2 and 11 respectively. TEEG counts increase steadily and never decrease, rising from a count of 128 in December 2006 to 134 in March 2010.

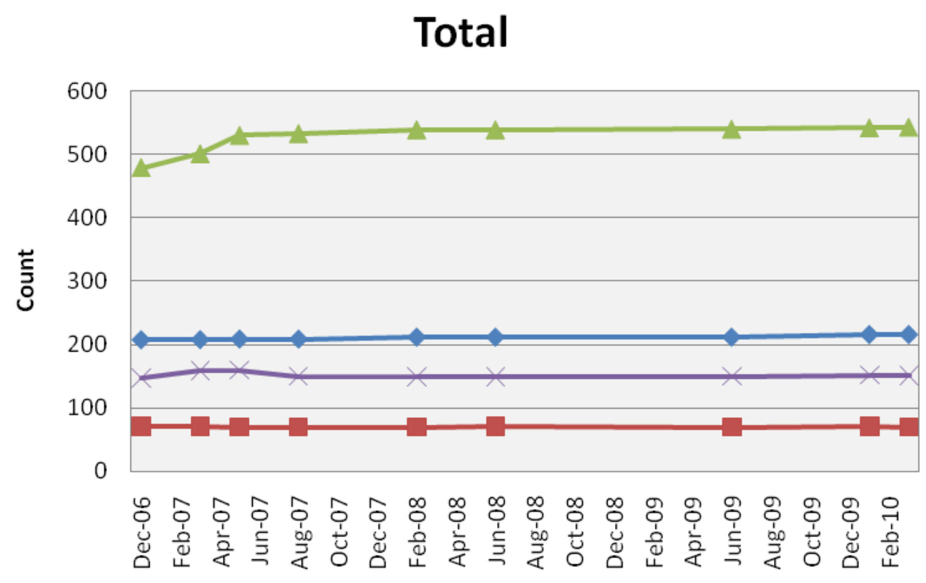
Data for the Observer pattern mirrors that of the Factory pattern data. TEEG counts rise from 315 in December, 2006 to 373 in March 2010. PEEG counts increase steadily from 146 to 154 over that same time period. There is an abrupt increase in TEAG counts in March 2007 followed by an abrupt decrease in May 2007. Excluding these two release dates, the data for TEAG remains constant. The Data for PEAG shows little variation while PIG and TIG counts do not change at all remaining at 11 and 19 respectively.



	Dec-06	Mar-07	May-07	Aug-07	Feb-08	Jun-08	Jun-09	Jan-10	Mar-10
◆ PEEG	146	147	148	148	150	150	150	154	154
■ PEAG	30	30	29	29	29	29	29	30	30
▲ TEEG	315	337	365	365	370	370	370	372	373
× TEAG	72	84	85	74	74	74	75	75	75

Figure 6. Aggregate grime counts for 4 Observer pattern realizations (Tomcat)

The aggregate data for all design pattern realizations is shown in figure 7. TEEG counts increase consistently while PEAG, TEAG, and PEEG counts vary slightly throughout the study. PIG and TIG counts show no change with values of 11 and 19 respectively.



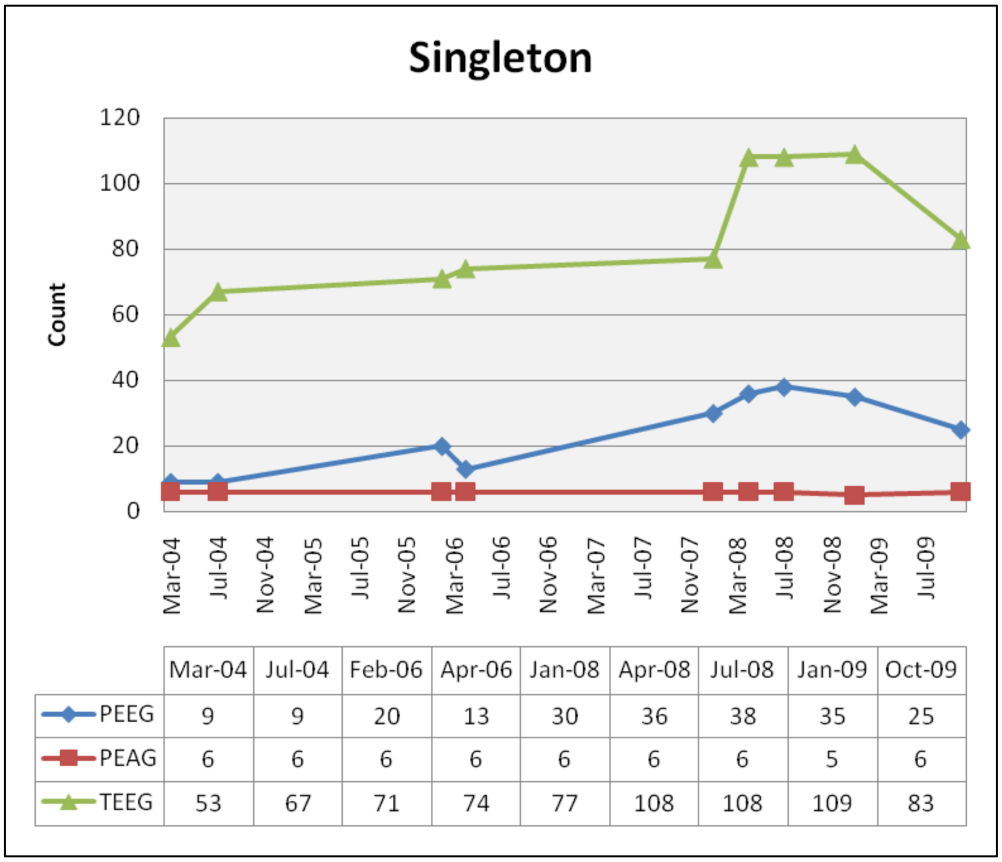
	Dec-06	Mar-07	May-07	Aug-07	Feb-08	Jun-08	Jun-09	Jan-10	Mar-10
◆ PEEG	208	208	209	209	212	212	212	216	216
■ PEAG	71	71	70	70	70	71	70	71	70
▲ TEEG	479	501	530	533	539	539	540	542	543
✕ TEAG	147	159	160	150	149	149	150	152	151

Figure 7. Aggregate grime counts for 18 design pattern realizations (Tomcat) (3 Singleton, 2 Adapter, 2 Façade, 7 Factory, and 4 Observer)

5.2 Dr. Java

We gathered data from 25 realizations representing 7 design patterns. This section displays the data for each pattern as well as the aggregate data for all design pattern realizations.

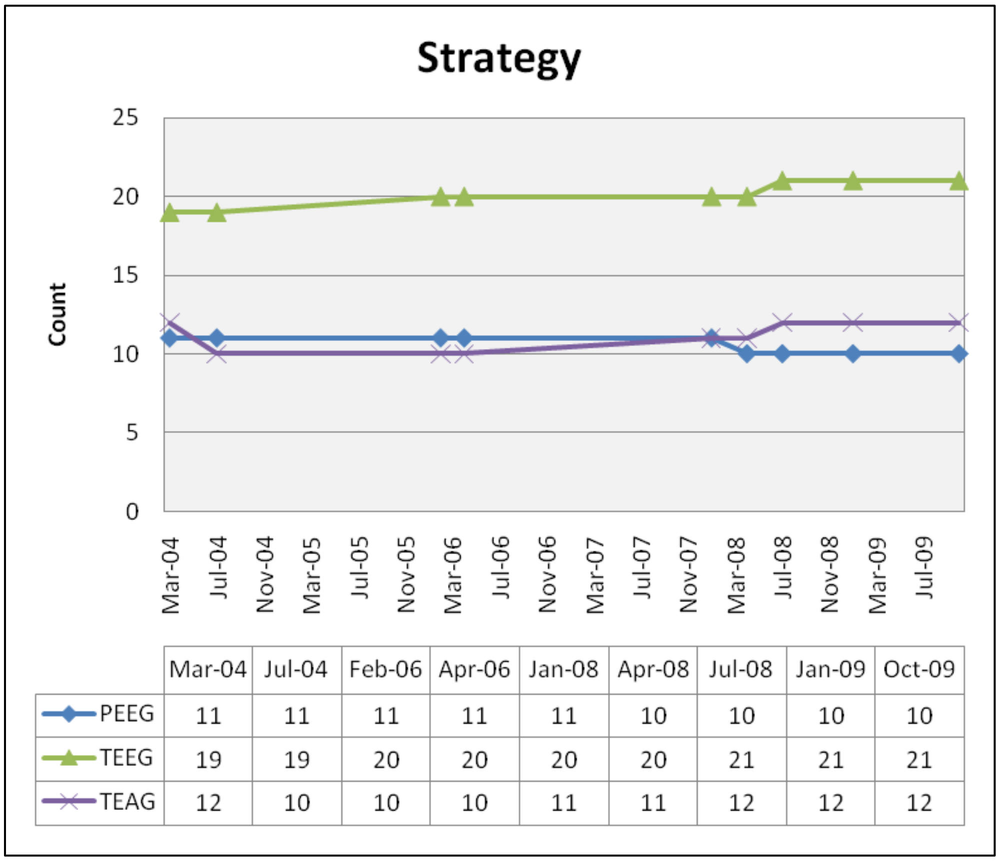
Figure 8 shows the results for 13 Singleton pattern realizations. TIG, PIG, and TEAG have a count of 0 throughout. PEAG shows very little change throughout the time span of the study. Grime counts for PEEG and TEEG consistently increase from March 2004 to January 2009, increasing from 9 to 35 and 53 to 109 respectively. There is a noticeable decrease in grime count for both PEEG and TEEG between the last two releases studied.



	Mar-04	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
PEEG	9	9	20	13	30	36	38	35	25
PEAG	6	6	6	6	6	6	6	5	6
TEEG	53	67	71	74	77	108	108	109	83

Figure 8. Aggregate grime counts for 13 Singleton pattern realizations (Dr. Java)

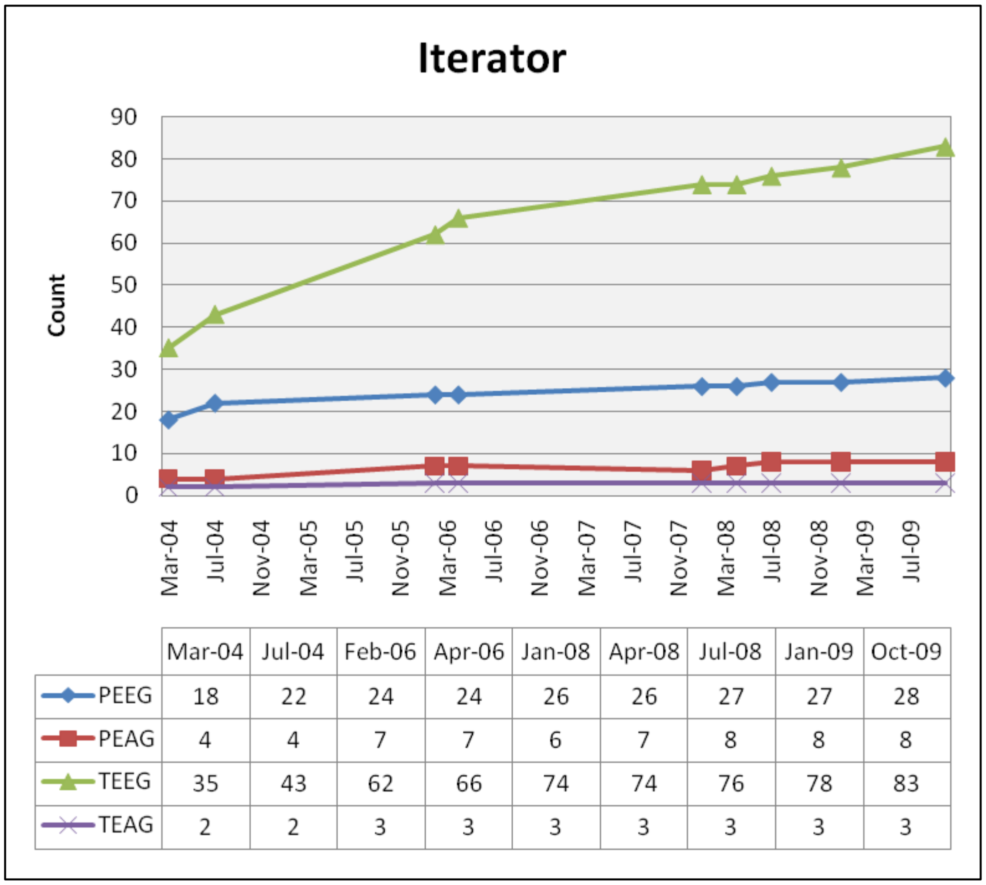
The data for the strategy pattern is relatively invariable. PEAG, PIG, and TIG counts show no change with values of 1, 0, and 0 respectively. PEEG, TEEG, and TEAG counts vary noticeably.



	Mar-04	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
◆ PEEG	11	11	11	11	11	10	10	10	10
▲ TEEG	19	19	20	20	20	20	21	21	21
× TEAG	12	10	10	10	11	11	12	12	12

Figure 9. Aggregate grime counts for 1 Strategy pattern realization (Dr. Java)

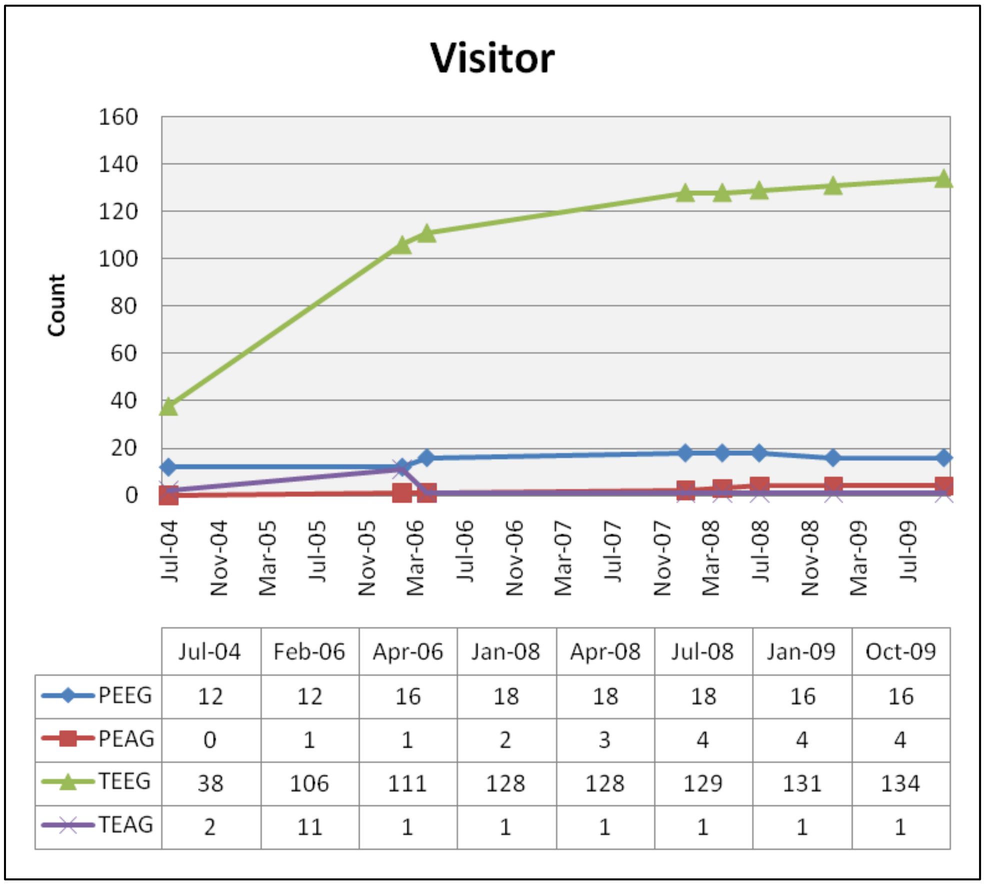
Data for the Iterator pattern show similar trends as those from the Singleton pattern. PEEG and TEEG counts show consistent growth throughout the lifecycle of the study. PEEG counts increase from 18 in March 2004 to 28 in October 2009. During the same time span TEEG counts increases from 35 to 83. PEAG counts show some variability, increasing from 4 to 8 throughout all releases. TEAG counts increase by 1 over the study period. PIG stays constant with a count of 1 while TIG has a count of 0 throughout.



	Mar-04	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
PEEG	18	22	24	24	26	26	27	27	28
PEAG	4	4	7	7	6	7	8	8	8
TEEG	35	43	62	66	74	74	76	78	83
TEAG	2	2	3	3	3	3	3	3	3

Figure 10. Aggregate grime counts for 1 pattern realization (Dr. Java)

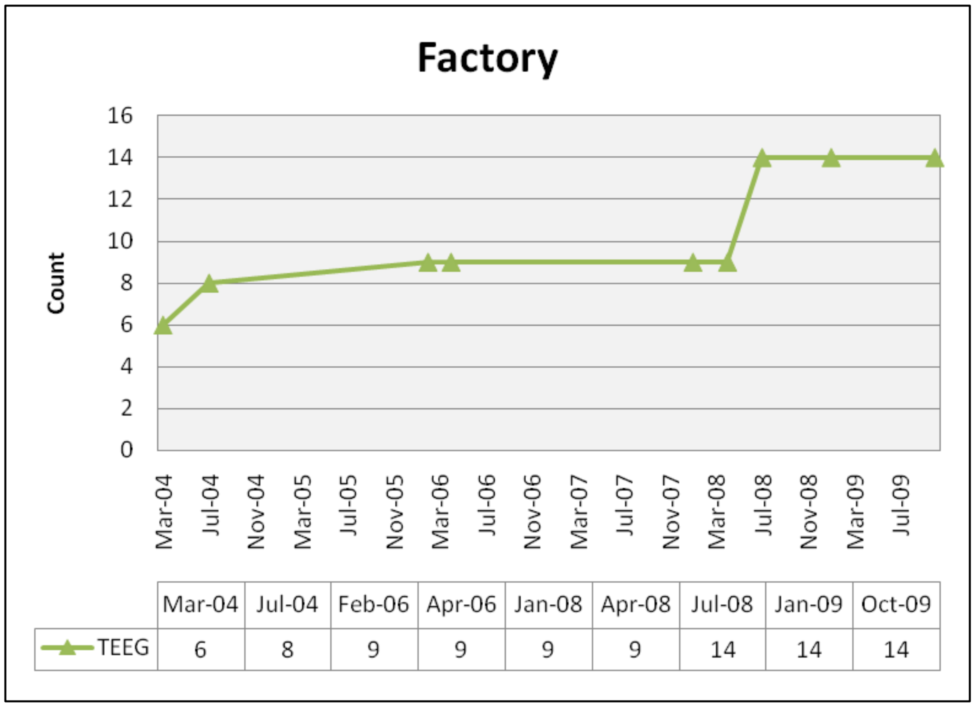
In the Visitor data TEEG count increases from 38 to 134 over all releases. PEAG, TEAG, and PEEG show some small variations. PIG and TIG have constant counts of 4 and 6 respectively.



	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
PEEG	12	12	16	18	18	18	16	16
PEAG	0	1	1	2	3	4	4	4
TEEG	38	106	111	128	128	129	131	134
TEAG	2	11	1	1	1	1	1	1

Figure 11. Aggregate grime counts for 2 visitor pattern realizations (Dr. Java)

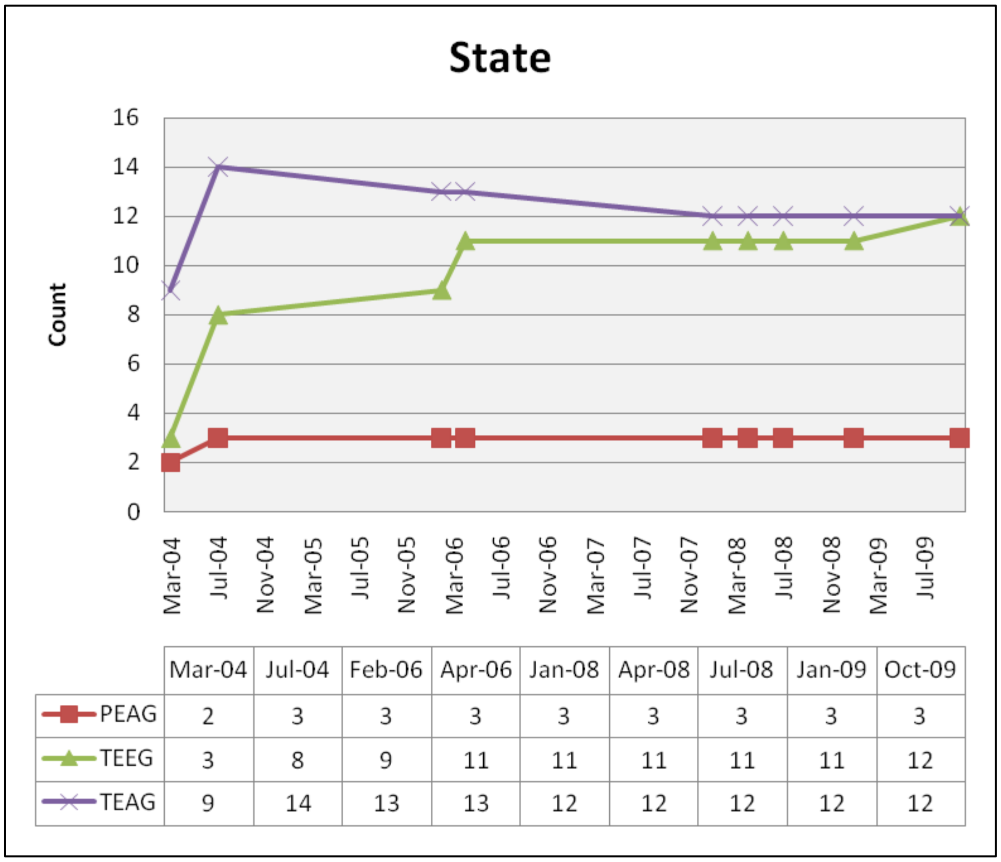
Factory pattern data is displayed in figure 12. TEEG counts show marked increases over the period of the study. The counts of all other grime categories remain unchanged.



	Mar-04	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
TEEG	6	8	9	9	9	9	14	14	14

Figure 12. Aggregate grime counts for 3 Factory pattern realizations (Dr. Java)

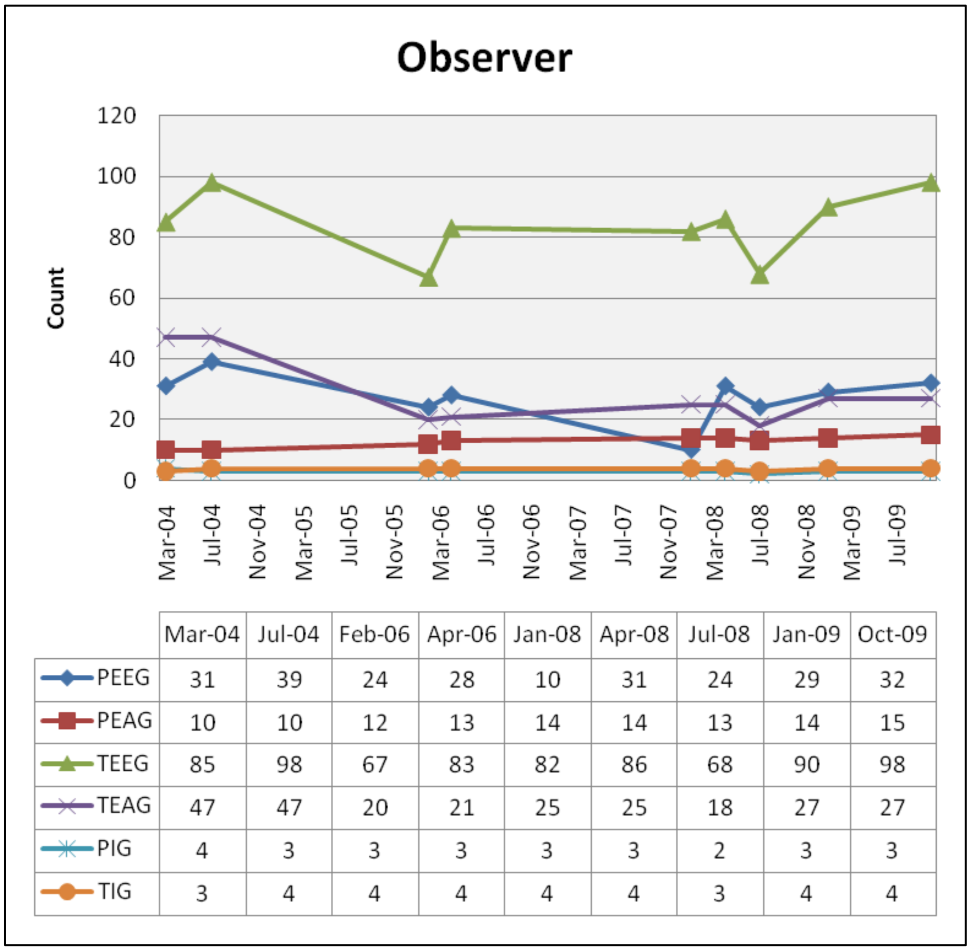
Grime counts from the State pattern behave in a similar manner to those of the Factory pattern. TEEG counts increase several times between releases and increase from 3 to 12 over the span of the study. PEAG counts increase by 1 after the first releases while PIG and TIG counts show no variance. In contrast to the data from the other pattern realizations, TEAG counts show significant variation. There is a sharp increase between the first two releases, followed by a consistent decline.



	Mar-04	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
PEAG	2	3	3	3	3	3	3	3	3
TEEG	3	8	9	11	11	11	11	11	12
TEAG	9	14	13	13	12	12	12	12	12

Figure 13. Aggregate grime counts for 2 State pattern realizations (Dr. Java)

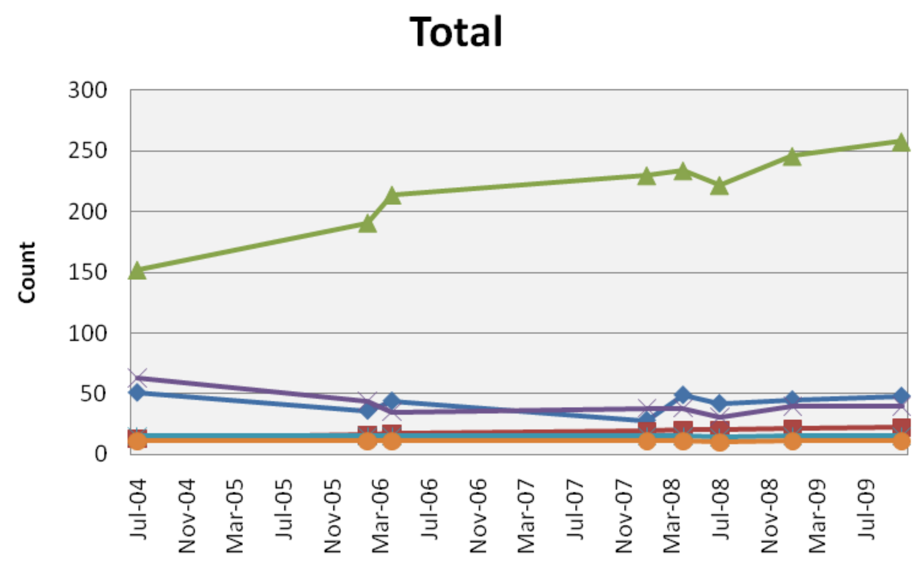
The Observer data shows the most variation. Similar to the Visitor and Iterator data, PEAG counts slowly grows while PEEG, TEEG, and TEAG counts all display abrupt changes between several releases. PIG counts increases by 1 after the first release and TIG counts decrease by 1 over the same period.



	Mar-04	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
PEEG	31	39	24	28	10	31	24	29	32
PEAG	10	10	12	13	14	14	13	14	15
TEEG	85	98	67	83	82	86	68	90	98
TEAG	47	47	20	21	25	25	18	27	27
PIG	4	3	3	3	3	3	2	3	3
TIG	3	4	4	4	4	4	3	4	4

Figure 14. Aggregate grime counts for 3 Observer pattern realizations (Dr. Java)

The grime data for all design pattern realizations shown in figure 15 reflects the trends from data for individual patterns. TEEG counts show consistent increases throughout the study period. PEAG counts show some slow growth due to the Visitor, Iterator, and State patterns. TEAG and PEEG counts show the most variance. PIG and TIG counts show an insignificant amount of change varying by at most 1 during the period of study.



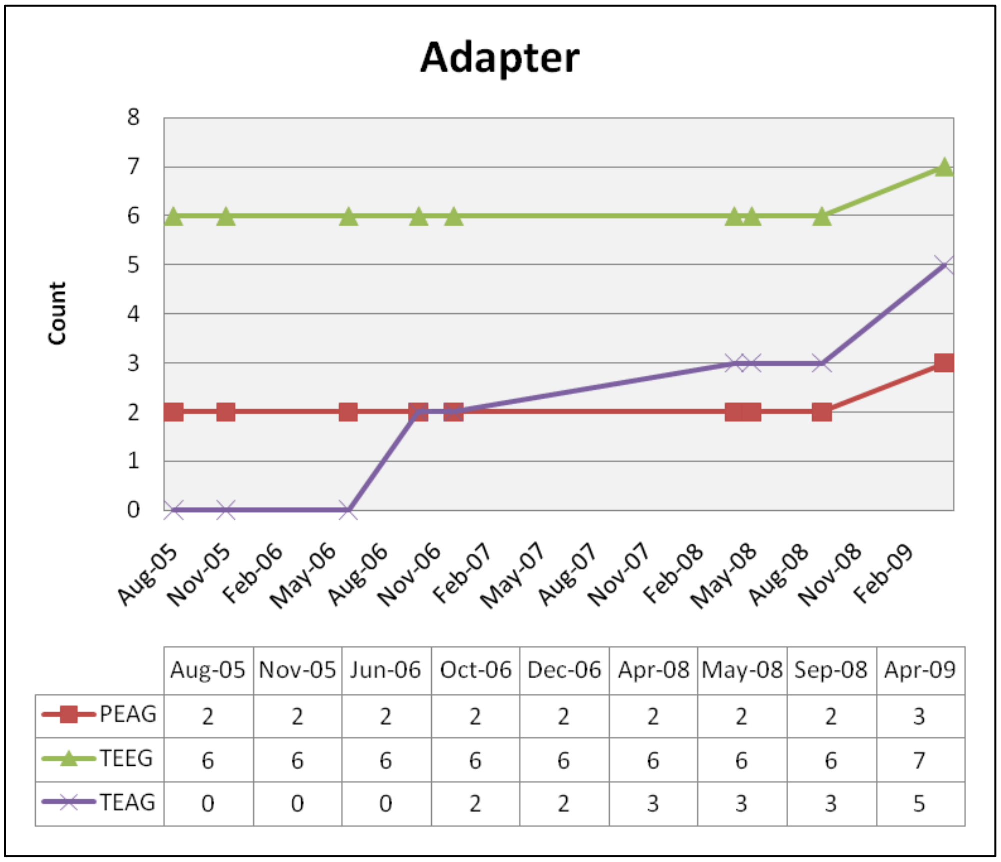
	Jul-04	Feb-06	Apr-06	Jan-08	Apr-08	Jul-08	Jan-09	Oct-09
◆ PEEG	51	36	44	28	49	42	45	48
■ PEAG	13	16	17	19	20	20	21	22
▲ TEEG	152	191	214	230	234	222	246	258
× TEAG	63	44	35	38	38	31	40	40
✳ PIG	15	15	15	15	15	14	15	15
● TIG	11	11	11	11	11	10	11	11

Figure 15. Aggregate grime counts for 25 pattern realizations (Dr. Java) (13 Singleton, 1 Strategy, 1 Iterator, 2 Visitor, 3 Factory, 2 State, and 3 Observer)

5.3 Apache Derby

Data mined from Apache Derby consists of a total of 16 design pattern realizations from 6 design patterns. Grime counts for the Observer pattern are not shown because there were no changes in the initial data over the 6 releases of the study period; all categories remained stable at 11, 6, 17, 10, 0, and 2 for PEEG, PEAG, TEEG, TEAG, PIG, and TIG respectively.

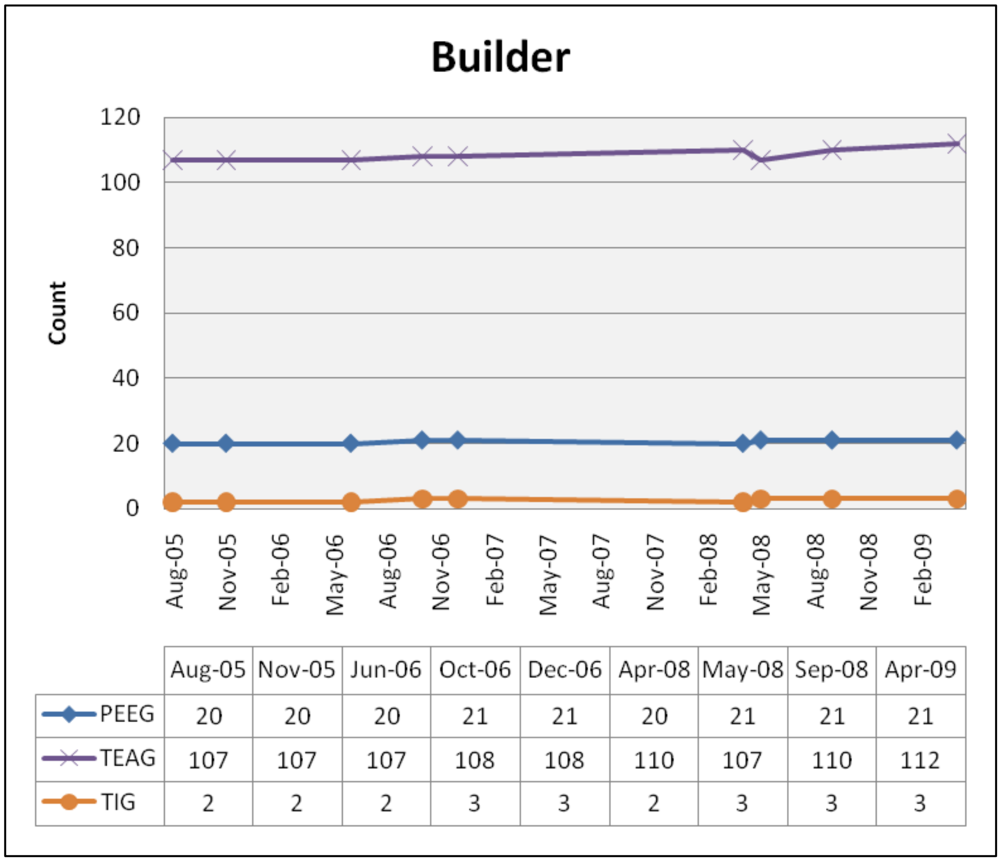
Figure 16 displays results for the Adapter pattern. There are no changes in PEEG, PIG, or TIG counts throughout. TEAG counts increase several times over the period of study. There are three categories (PEAG, TEAG, and TEEG) that show a simultaneous increase from September 2008 through March 2009. Overall, from August 2005 through September 2008, there is very little change in grime counts.



	Aug-05	Nov-05	Jun-06	Oct-06	Dec-06	Apr-08	May-08	Sep-08	Apr-09
■ PEAG	2	2	2	2	2	2	2	2	3
▲ TEEG	6	6	6	6	6	6	6	6	7
× TEAG	0	0	0	2	2	3	3	3	5

Figure 16. Grime counts for 1 Adapter pattern realization (Derby)

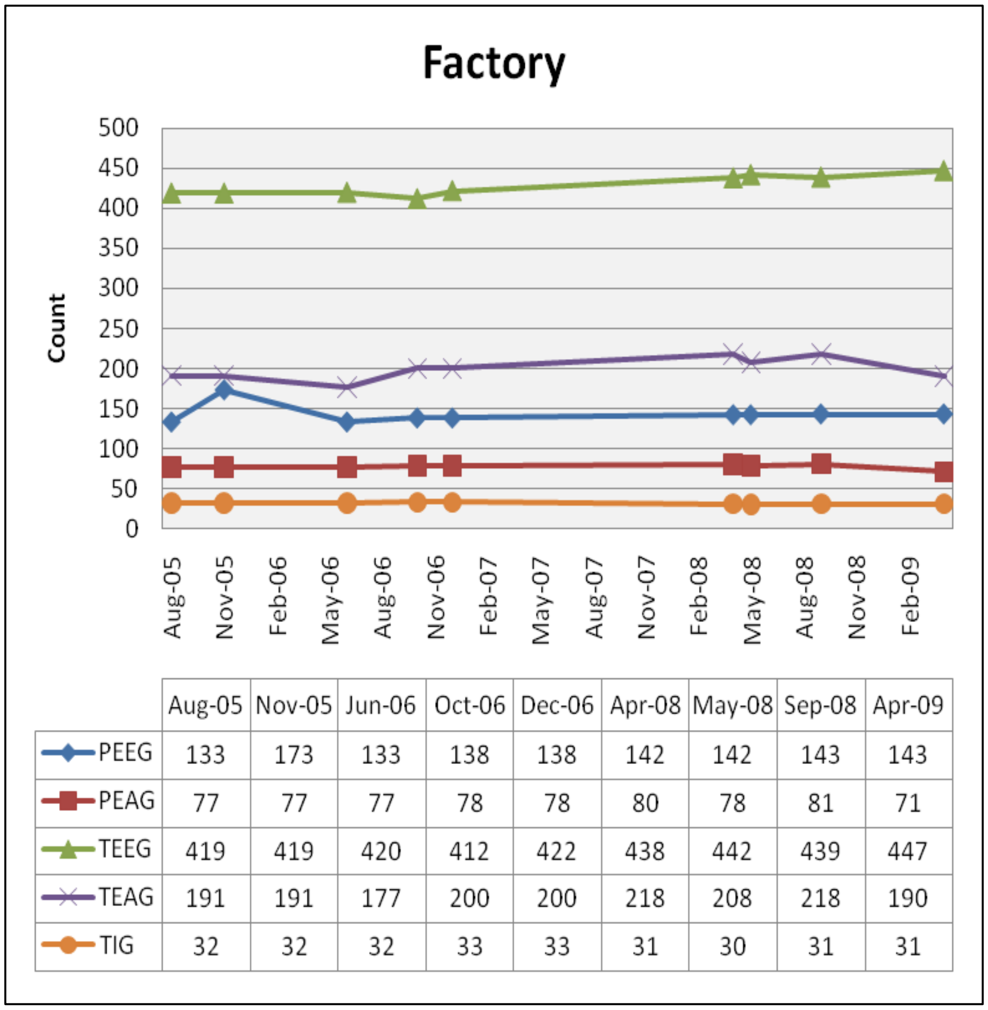
In the Builder pattern, most categories show at most a variation of one grime count over the entire study period. TEAG counts are the exception displaying a rise of 5 from August 2005 to April 2009.



	Aug-05	Nov-05	Jun-06	Oct-06	Dec-06	Apr-08	May-08	Sep-08	Apr-09
◆ PEEG	20	20	20	21	21	20	21	21	21
× TEAG	107	107	107	108	108	110	107	110	112
● TIG	2	2	2	3	3	2	3	3	3

Figure 17. Grime counts for 1 Builder pattern realization (Derby)

The Factory pattern data shown in figure 18 shows more variance than the previous data. PIG counts show no changes throughout while TIG counts varie by a magnitude of 2. PEAG and TEAG counts display many abrupt changes between releases, but they have a similar grime count in April 2009 as compared to August 2005. PEEG and TEEG counts increase and decrease throughout the studied interval eventually settling with greater grime counts over time.

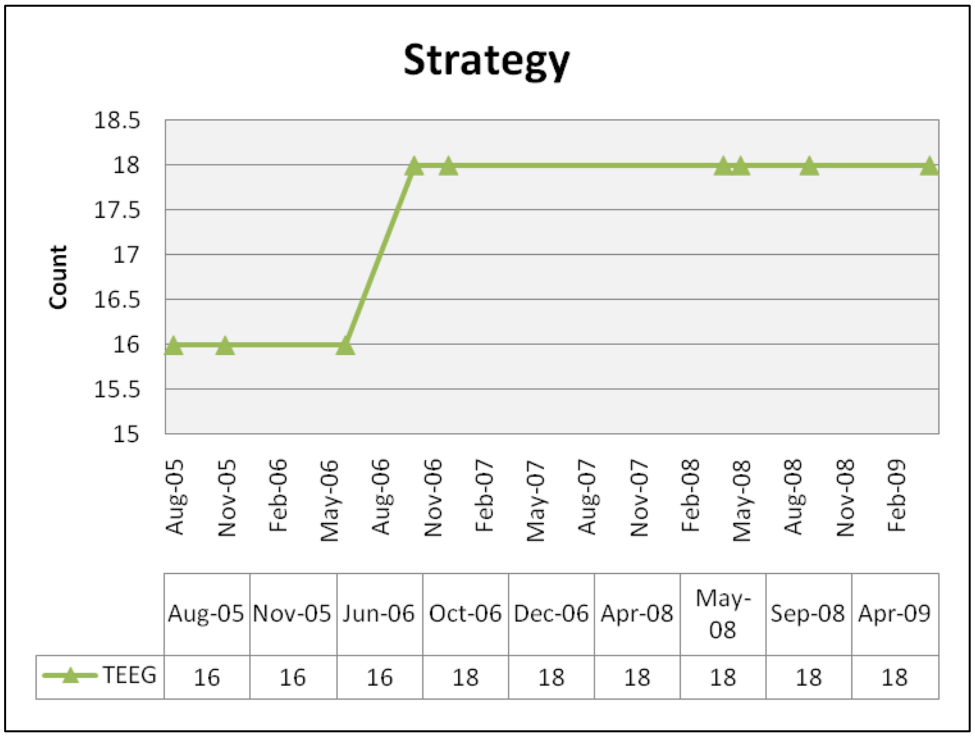


	Aug-05	Nov-05	Jun-06	Oct-06	Dec-06	Apr-08	May-08	Sep-08	Apr-09
◆ PEEG	133	173	133	138	138	142	142	143	143
■ PEAG	77	77	77	78	78	80	78	81	71
▲ TEEG	419	419	420	412	422	438	442	439	447
× TEAG	191	191	177	200	200	218	208	218	190
● TIG	32	32	32	33	33	31	30	31	31

Figure 18. Aggregate grime counts for 12 Factory pattern realizations (Derby)

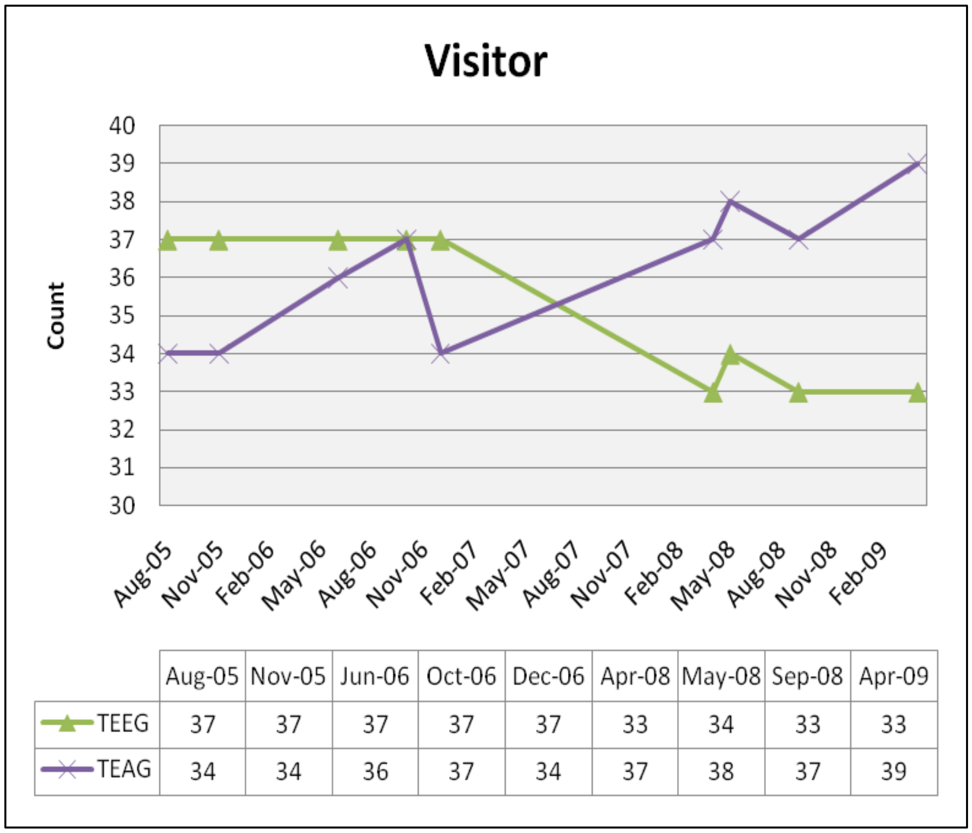
Data for the Strategy pattern shows stability in all categories with the exception of TEEG counts, where there is an increase between June 2005 and October 2006.

Visitor pattern data shows little variation as well. TEEG and TEAG counts are the only grime counts that change over the study interval. TEAG counts display up and down trends with an overall increase in grime. TEEG counts decrease from beginning to end except for an increase of 1 count in May 2008.



	Aug-05	Nov-05	Jun-06	Oct-06	Dec-06	Apr-08	May-08	Sep-08	Apr-09
TEEG	16	16	16	18	18	18	18	18	18

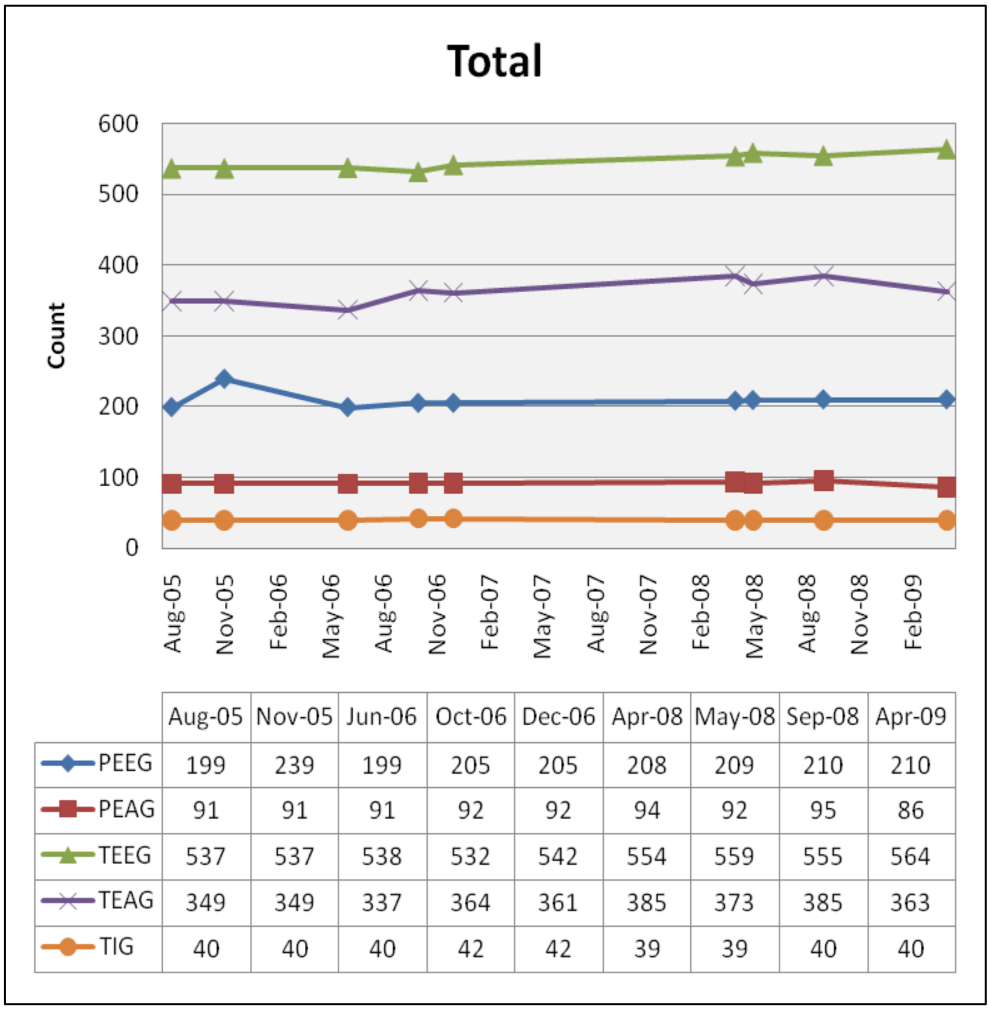
Figure 19. Grime counts for 1 Strategy pattern realization (Derby)



	Aug-05	Nov-05	Jun-06	Oct-06	Dec-06	Apr-08	May-08	Sep-08	Apr-09
TEEG	37	37	37	37	37	33	34	33	33
TEAG	34	34	36	37	34	37	38	37	39

Figure 20. Grime counts for 1 Visitor pattern realization (Derby)

The total grime counts for all design patterns are shown in figure 21. PIG data shows no changes throughout while TIG counts vary by at most 3. PEAG counts are the only data that show a consistent decrease, falling from 91 to 86 over the 4 year study period. PEEG, TEEG, and TEAG counts all increase by a total of 11, 27, 14 respectively over the period of study.



	Aug-05	Nov-05	Jun-06	Oct-06	Dec-06	Apr-08	May-08	Sep-08	Apr-09
PEEG	199	239	199	205	205	208	209	210	210
PEAG	91	91	91	92	92	94	92	95	86
TEEG	537	537	538	532	542	554	559	555	564
TEAG	349	349	337	364	361	385	373	385	363
TIG	40	40	40	42	42	39	39	40	40

Figure 21. Aggregate total grime counts for 16 pattern realizations (Derby) (1 Adapter, 1 Builder, 12 Factory, 1 Strategy, 1 Visitor)

6. ANALYSIS AND DISCUSSION

We have gathered data from three large open source software systems. The data was used to test the hypotheses proposed in section 4.4. We use statistical methods to determine whether the results support or contradict each hypothesis. The significance tests are also accompanied by a brief discussion.

Section 6.1 gives a description of the statistical methods used for hypothesis testing. Sections 6.1.1 through 6.1.3 contain test results and discussions for each system studied. A summary is given in section 6.1.4.

6.1 Trend Analysis

The time series nature of the data lends itself to trend analysis. Preliminary inspection of the data led to the choice of simple linear regression (SLR) as the method of analysis because raw data show evidence of linear trends. Previous research has revealed that non-linear models are appropriate for software growth [70]. However, we use a linear model for analysis for two reasons. First, the raw grime count data almost unanimously displays linear growth. Second, grime counts do not necessarily follow the same trends as software growth; there is no evidence to make this assumption.

Grime counts represent the dependent variable, and time, the independent variable measured in days, is the unit of association. These data fit neatly in the general SLR model because the hypotheses require a test for a linear relationship between time and grime counts.

The model shown below represents the grime count (denoted by y); x represents time, β_0 is the parameter representing the y-intercept, β_1 is the parameter for slope (also called the regression coefficient), and ε is the error.

$$y = \beta_0 + \beta_1 x + \varepsilon$$

Using SLR to model the data allows us to determine if a possible linear relationship between the independent variable (time in the form of release dates) and the dependent variable (grime count) exists. A significant relationship between the two variables for any modular grime category will cause a rejection of the hypothesis for said category because each hypothesis states that there is not a linear relationship between grime counts and time. A significant statistical result reveals a linear relationship between grime and time and therefore causes us to reject the null hypothesis.

In order to find statistically significant relationships between the variables, we tested whether the regression coefficient β_1 is equal to 0. Given the SLR model, a regression coefficient that is significantly non-zero indicates a non-trivial relationship between the independent and dependent variables. Changes in time cause changes in the grime estimate because the regression coefficient (when it is not equal to zero) is multiplied by time in the linear equation shown above. This in turn contradicts the hypothesis that grime counts do not grow linearly over time. Note that a slope of 0 may indicate a linear relationship between grime counts and time, but the relationship is non-growing and it confirms the null hypothesis because grime counts are unchanging over time.

A t-test was used to determine the significance of the regression coefficient. The t-value is the ratio of the regression coefficient and the standard error because we are testing against a regression coefficient of 0. The t-value is then compared to a t-distribution to obtain a p-value, which determines the likelihood that a t-value that large would occur in a sample in which the dependent and independent variables are not related. The p-value is the probability that the regression could occur due to random chance; a very small p-value indicates that the changes in grime count are correlated with increases in time and not simply random. For this analysis, a p-value less than .05 deems the regression coefficient statistically significant. A p-value of .05 is a common threshold used in empirical research.

There are some details of the resulting SLRs to note. The independent variable in the analysis is the number of days since the date 01/01/1900. This means the y-intercept represents the estimated grime count at that date. This makes the y-intercept value meaningless as we are only interested in the slope of the regression (change in grime over time) and not the estimated grime count at some point in the past. As a result of using days as the measure of time, the regression coefficients represent changes in grime count *per day*. A positive coefficient represents *increases* in grime count by day and a negative coefficient represents *decreases* in grime count by day.

In the following sections an SLR is run for each hypothesis using a least squares method. The regression equation, as well as the regression coefficient t-test is shown for each SLR, along with an interpretation about the validity of the hypothesis.

6.1.1 Apache Tomcat

There is one hypothesis to test for each grime category. Sections 6.1.1.1 through 6.1.1.5 include the analysis of statistical results. The results are summarized in section 6.1.1.6.

Table 11 shows the SLR information for Apache Tomcat. The regression coefficient and the standard error are shown. The t-value is calculated using the regression coefficient and the standard error. A small p-value (p < .05) indicates it is very unlikely that the regression coefficient could be equal to zero without a relationship between grime count and time.

Table 11. Regression data for Apache Tomcat.

	Regression coefficient	Standard error	t-value	p-value
PEAG	-0.00019	.000433	-0.44527	0.669569
TEAG	-0.00247	.003668	-0.67398	0.521950
PEEG	0.006589	.000758	8.690865	0.000054
TEEG	0.034949	.01303	2.682286	0.031435

6.1.1.1 PEAG. The p-value for PEAG is not significant. Therefore, we cannot make any claims about the true value of the regression coefficient. In turn, this supports the null hypothesis that PEAG buildup and time are not linearly related in Apache Tomcat.

6.1.1.2 TEAG. The calculations for TEAG are similar to those for PEAG. The small regression coefficient leads to a small t-value, which in turn leads to a large p-value. The p-value indicates the value of the regression coefficient may be due to chance. These results support the null hypothesis that TEAG buildup and time are not linearly related for the releases of Tomcat studied.

6.1.1.3 PEEG. The SLR for PEEG reveals the first indication of a significant relationship between time and grime buildup. The extremely low p-value indicates that the regression coefficient is very unlikely to occur if there truly was no linear relationship between PEEG count and time. This information allows us to reject the null hypothesis that PEEG does not show linear growth over time.

6.1.1.4 TEEG. The regression model for TEEG reveals a significant relationship. The t-test on the regression coefficient results in a p-value which is less than the threshold of .05. We reject the null hypothesis for TEEG. The positive regression coefficient reveals a positive relationship between time and grime count.

6.1.1.5 PIG and TIG. A regression is not necessary for the PIG and TIG data from Tomcat because the results show no changes throughout the study. A straight line indicates a regression coefficient of zero and statistically unchanging grime counts. These results are supportive of the null hypothesis for PIG and TIG; the data justify the notion that PIG and TIG do not display linear growth over time in this software.

6.1.1.6 Tomcat Analysis Summary. There is one hypothesis for each taxonomy category in Apache Tomcat. Each hypothesis predicts that grime buildup grows linearly over the releases studied. We used a simple linear regression to estimate a regression coefficient (slope) for the data. By testing whether the slope is equal to zero, we determine if there is a positive relationship between time and grime count (i.e. whether grime counts increase over time). In the case where the slope is not equal to zero, there is evidence to contradict the null hypothesis.

The data for PEEG and TEEG suggests we have evidence to contradict the null hypotheses. The data support the notion that PEEG and TEEG do buildup over the time period studied. The results for all other modular grime categories support the null hypotheses. There is no indication to suggest a linear relationship between time and grime count.

6.1.2 Dr. Java

The regression information for each category is presented and interpreted in this section. A summary of the analysis results is included in section 6.1.2.6. Table 12 contains all of the regression information.

Figure 12. Regression data for Dr. Java.

	Regression coefficient	**Standard error**	**t-value**	**p-value**
PEAG	0.00456	0.000226	20.1431	0.00000097
TEAG	-0.01006	0.00447	-2.248	0.065614
PEEG	-0.0007	0.004788	-0.14597	0.888728
TEEG	0.049522	0.006388	7.813994	0.000232
PIG	-0.00012	0.000217	-0.5514913	0.601216
TIG	-0.00012	0.000218	-0.55149	0.601216

6.1.2.1 PEAG. The extremely small p-value for PEAG supports the conclusion that the regression coefficient is significant. The positive regression coefficient represents a statistically significant linear increase in grime count over time. This is evidence to reject the null hypothesis that PEAG counts do not grow linearly over time in the Dr. Java versions studied.

6.1.2.1 TEAG. The p-value in table 12 has a value just above the threshold. The regression coefficient is negative, indicating grime decreases over time. The insignificant p-value supports the null hypothesis that TEAG buildup is not linearly related to time in the releases of Dr. Java studied. The negative coefficient indicates that grime does not grow over time in either case.

6.1.2.3 PEEG. The linear regression for PEEG shows no connection between time and grime count. The regression coefficient from table 12 is extremely small and the p-value indicates a lack of significance. This analysis supports the null hypothesis for PEEG.

6.1.2.4 TEEG. The linear model for TEEG in Dr. Java provides some interesting results. A regression coefficient of .0495 is quite large for this data. At first glance this seems like a significant increase. The t-test backs up this assertion with a t-value of 7.8 which reveals an extremely small p-value well below the threshold. This information contradicts the null hypothesis and supports the conclusion that TEEG counts are significantly increasing in a linear fashion over the releases studied.

6.1.2.5 PIG and TIG. The regression data for PIG and TIG are also shown in table 12. The analysis results are expected as the data for both PIG and TIG show very little variation throughout the study. Without any variation over time, linear growth of grime counts over time cannot be inferred. The large p-values support this notion as well as the null hypothesis; grime buildup (PIG and TIG) is not linearly related to time.

6.1.2.6 Dr. Java Analysis Summary. The analysis results for all six modular grime categories can be broken down as follows. Four categories (PEEG, TEAG, PIG, and TIG) show no evidence of linear grime buildup over time. The statistical analysis provides no evidence to reject the null hypothesis that grime buildup for these four categories is insignificant.

Two grime categories (PEAG and TEEG) show significant results after analysis. The low p-values for the regression coefficients indicate that grime builds up linearly over time for both types. A regression coefficient of .00456 for PEAG indicates a slow but steady rise in grime count. The regression coefficient for TEAG (0.01006) indicates an increase of 1 grime count every 10 days on average.

6.1.3 Apache Derby

Table 13 shows the analysis results for Apache Derby. Sections 6.1.3.1 through 6.1.3.5 elaborate on the results and section 6.1.3.6 summarizes all results for this system.

Table 13. Regression data for Apache Derby.

	Regression coefficient	Standard error	t-value	p-value
PEAG	-0.00029	0.00196	-0.14911	0.885671
TEAG	0.025331	0.008521	2.972756	0.020727
PEEG	-0.00234	0.009286	-0.25205	0.808242
TEEG	0.022215	0.003453	6.433066	0.000356
TIG	-0.00077	0.000806	-0.95379	0.371955

6.1.3.1 PEAG. An analysis for PEAG reveals a linear model that is of little significance. The small value for the regression coefficient (slope) indicates that there is little growth in PEAG. The large p-value further confirms this conclusion. The null hypothesis for PEAG is supported; PEAG counts do not display linear growth over time.

6.1.3.2 TEAG. The linear model for TEAG provides an interesting result. The slope indicates a rise of .025 for TEAG count per day. The p-value for a t-test on the regression coefficient attests to the fact that the slope is statistically significant. This analysis contradicts the null hypothesis, and supports the alternative hypothesis that TEAG builds up linearly over time.

6.1.3.3 PEEG. The analysis for PEEG produces similar results to those for PEAG. The regression coefficient is small and the p-value for the t-test analysis produces an insignificant result. The model shows no relationship between time and PEEG count. This is supportive of the null hypothesis.

6.1.3.4 TEEG. The analysis of the TEEG data provides some interesting results on par with those for TEAG. The slope shows that TEEG increases at a rate of .022 counts per day. The extremely small p-value eliminates any concern that the results are a statistical oddity. The significance of these results contradicts the null hypothesis while supporting the notion that TEEG does buildup linearly over time in the releases studied.

6.1.3.5 PIG and TIG. The data for PIG is unchanging. Therefore no regression is necessary for analysis because grime count does not grow over time . The regression coefficient for TIG is extremely small, indicating a slight decrease in TIG counts over time. The p-value supports the assumption that the small regression coefficient is insignificant. The data and analysis support the hypotheses that PIG and TIG in Apache Derby have no linear relationship with time.

6.1.3.6 Derby Analysis Summary. There are two modular grime sub types that show linear growth in the data gathered. TEEG and TEAG both have small p-values that show significance of their respective regression coefficients. Both regression coefficient values are positive which means that TEEG and TEAG are significantly increasing in a linear fashion. The analysis of the other modular grime categories shows insignificant changes throughout. The null hypotheses (that grime buildup does not grow linearly over time) cannot be rejected for PEEG, PEAG, PIG, and TIG.

6.1.4 Summary of statistical analysis

The statistical results are summarized in table 14. Each cell contains the p-value from a t-test on the regression coefficient for the linear model of each grime sub category. The meaning of the p-values is explained in further detail in section 6.1. Significant p-values are shown in bold.

Table 14. Slope significance (p-value) for all systems and modular grime categories.

	PEAG	**TEAG**	**PEEG**	**TEEG**	**PIG**	**TIG**
Apache Tomcat	0.669569	0.521950	**.0000535**	**0.031435**	n/a	n/a
Dr. Java	**.0000010**	0.065614	0.888728	**0.000232**	0.601216	0.601216
Apache Derby	0.885671	**0.020727**	0.808242	**0.000356**	n/a	0.371955

There are four significant p-values within all three systems studied. Most notably, the p-values support the notion that TEEG shows linear growth over time in every system. PIG and TIG show no significant p-values in any of the systems studied. There is significant linear PEEG buildup in Tomcat, linear PEAG buildup in Dr. Java, and linear TEAG buildup in Apache Derby. The raw data along with this corresponding analysis is discussed more thoroughly in section 6.2.

6.2 Discussion

This section contains an informal discussion of the data and analysis results found in the previous two sections. First we contemplate general implications of the results in section

6.2.1. Finally we discuss the results for each software system individually in sections 6.2.2 through 6.2.4.

6.2.1 General Discussion

The results for the Singleton pattern throughout all of the software studied have one thing in common: PIG and TIG do not change. Internal grime is likely nonexistent in the Singleton pattern realizations because the realizations are defined using only one class. Internal grime can only occur when new illegal relationships appear between pattern classes. In the case of the Singleton, a self-referential relationship is already defined by the pattern. It is unlikely that any new internal relationships will appear over the evolution of the pattern and internal grime will remain unchanged.

Results support rejection of six null hypotheses; PEEG and TEEG in Apache Tomcat, PEAG and TEEG in Dr. Java, and TEAG and TEEG in Apache Derby. We do not have evidence to reject other null hypotheses, including PIG and TIG in Apache Tomcat and PIG in Apache Derby where a statistical test was not needed to confirm that grime does not grow linearly over time.

The rejection of the TEEG hypothesis in all 3 systems provides the most evidence for generalization. In other words, of all taxonomy categories, TEEG most consistently shows linear growth over time in the open source software we studied. TEEG is caused by temporary relationships that originate within the design pattern classes.

Temporary coupling is likely to increase over time as functionality (methods) is added to classes. This could happen to all classes in the system, causing increases in counts for grime originating internally and externally. This is reflected in the data for TEEG and

TEAG in Apache Derby where temporary grime counts increase regardless of coupling direction. TEAG does not show a significant linear relationship with time in the other two systems studied. The differences between TEEG and TEAG indicate that temporary grime may be more likely to appear in design pattern classes. The addition of new and expanded functionality to pattern classes, not originally intended by the pattern, is likely the cause of significant count increases for TEEG. In other words, developers add to pattern classes to increase functionality instead of modularizing new features by creating new classes for functionality unrelated to the pattern. The result is an increase in TEEG counts for the pattern classes.

Overall, Persistent grime shows less significant growth in all of the software studied (only PEEG in Apache Tomcat and PEAG in Dr. Java show linear growth over time). One possible explanation is that developers tend to avoid modifications that cause classes to be strongly coupled. The motivation is to avoid highly coupled groups of classes because the relationships can become complex and difficult to comprehend, making maintenance more time consuming. In this case, developers augment classes by expanding or adding methods, which in turn increases temporary and not persistent coupling.

One notable result is that PIG and TIG show no linear relationship with time in any of the systems studied. This is an indication that internal grime is not an issue. One explanation is that the number of *candidate* internal grime relationships is limited. Candidate refers to a currently non-existing relationship that would contribute to modular grime if created. As an example, a pattern that consists of three classes already coupled in a specific way (as defined by the pattern definition) can only become more highly coupled in a limited number of ways. There are very few possible candidate internal grime relationships, making the odds of encountering one less likely than some other external grime relationships. The result is an insignificant buildup of internal grime.

6.2.2 Apache Tomcat

Some of the pattern realizations studied showed no changes during the entire study period. The Singleton, Adapter, and Façade patterns were stable in the sense that grime counts were invariant. This does not mean the classes went unmodified; their coupling with their environment and each other, as defined in section 3.2, was not altered. Intuitively this limits the types of changes that may occur because changes to attributes and method signatures would have an effect on grime counts if they are not compliant with the RBML. We can speculate that any changes made to the Singleton, Adapter, and Façade classes were minimal or their grime buildup was not harmful (i.e. the queries did not count it). In other terms, the change proneness of the pattern classes was low. This conclusion confirms previous research [25] and agrees with the intuition that design pattern classes should be less change prone that other classes because they have a clearly defined function that is unchanging [12].

Analysis of the Singleton, Adapter, and Façade patterns reveals a common characteristic; all of the pattern realizations studied were generally one class patterns, meaning they are usually realized with only one class. A pattern realization that contains fewer classes is statistically less likely to be changed because there are fewer classes to change.

The Factory and Observer pattern results account for the majority of the variance. Each shows an increase in TEEG counts over the first four releases. Beyond the fourth release, TEEG counts rise slowly but steadily. PEEG increases slowly over the studied time period for both patterns leading to the significance found in the analysis results. As opposed to the single class design pattern realizations discussed previously, the Factory and Observer pattern realizations contain multiple classes and also hold the most influence on the final results.

In regards to the main focus of this research, the grime counts for each category behave differently. Two categories (TEEG and PEEG) show significant linear growth over time in the Apache Tomcat system. The other four categories do not show linear trends in grime buildup. This result shows that while the aggregated modular grime count demonstrates significant buildup, it is mostly due to increases in the grime count of specific modular grime categories in the Apache Tomcat software.

While speculating as to why only TEEG and PEEG show significant increases, it may be pertinent to ask, what do they have in common? They both share an efferent coupling direction while they differ due to coupling strength. Specifically, TEEG and PEEG buildup originates in the pattern classes and not in the surrounding environment. This is counterintuitive to the idea that design pattern classes should be less change prone. Furthermore, this result implies that grime buildup occurs due to changes in pattern classes and the changes occur throughout the class in the form of changes to the state (attributes) and behavior (methods) of the class. Changes to the state and behavior of design pattern classes are likely the result of added functionality in addition to the known functions of the pattern.

6.2.3 Dr. Java

The grime data varies greatly by design pattern for the versions of Dr. Java in this study. Although the graphs for Apache Tomcat show general trends in the raw data easily identified without complex analysis, Dr. Java is much different. At different points in time, some patterns show increases in grime, while other patterns show decreases (e.g. the Observer and Factory patterns between May and October 2008). These inconsistencies have a cancelling effect on each other for the data of most interest, aggregate grime counts for all design patterns. This observation once again raises questions about grime buildup in different design patterns.

There are some common trends that should be noted in the data for different design patterns. The Iterator, Visitor, Factory, and State pattern data all show consistent increases in TEEG counts. These patterns are likely responsible for the analysis results which show the linear relationship between TEEG counts and time. The data for the Observer and Singleton pattern realizations is less consistent. Each shows decreases in TEEG counts during some releases, but a net increase in TEEG counts overall.

The data for PEEG best shows the unpredictable differences between design patterns. All patterns besides the Factory and State patterns show erratic increases and decreases over the development cycle. The only conclusion to draw from this is that there is no discernable growth pattern for PEEG buildup in Dr. Java. This is backed up by the aggregate data and analysis results. Similar conclusions can be made about the data for TEAG.

6.2.4 Apache Derby

The data for many of the grime types (PEEG, PEAG, PIG, and TIG) is mostly unchanging. The only noticeable change occurs for PEEG counts in the Factory pattern from

August 2005 to November 2005 where there is a large increase in counts which then subsides in the next release. One explanation is that a large modification was made to factory classes (possibly to add functionality unrelated to the pattern‘s purpose) during this span; developers may have refactored the factory classes in subsequent releases to cause the decreases in counts. In general, the data for PEEG, PEAG, PIG, and TIG are uninteresting.

The data and analysis show significant increases in both TEEG and TEAG. Both types of grime share one common characteristic: temporary coupling strength. Increases in temporary coupling for pattern classes, as well as the external classes coupled to the patterns, will cause this result. This may reflect a development trend where developers modify and add methods when adding functionality. The result is less persistent and more temporary coupling between classes. This effect was also observed in Apache Tomcat.

7. THREATS TO VALIDITY

Construct validity refers to the use of metrics and measures that meaningfully measure and capture the intended phenomenon. The code queries used to gather data were generated specifically to measure grime according to the formal definitions in the taxonomy, minimizing the threats to construct validity. One possible threat occurs during the checking of conformance of the grime candidate relationships to their corresponding RBML diagrams. The conformance check was done manually, due to the lack of automated tools, and human error is always a possibility.

To have content validity, the measures must completely represent the notions of modular grime coupling counts. The grime definitions proposed in this study could be subdivided further into specific coupling counts that could capture grime definitions at finer granularity levels; however these definitions would also fall under our higher level definitions and not threaten the content of our representation.

Internal validity refers to the causal connection between dependent and independent variables. In this case study there are 6 dependent variables —TEEG, PEEG, TEAG, PEAG, PIG, and TIG and one dependent variable, time. Our analysis provides evidence for a linear relationship between some of the modular grime types (mainly TEEG) and time, but no causal relationship can be inferred because of the experimental methodology.

External validity indicates that the study results can generalize to other systems. The criteria for software selection in this study represent a threat to external validity. Only open source systems with readily available source code for several releases and evidence for the existence of design patterns were chosen. Because this is a case study on 3 open source systems, generalization of the results to commercial or other open source software systems cannot be done. Repeated experimentation and increased sample size is necessary to eliminate the threats to external validity.

8. CONCLUSION AND FUTURE RESEARCH

Software decay is the study of changes in software over time that cause decreases in maintainability. Research on decay has progressed slowly over the last thirty years. More recently, the appearance of object oriented design patterns has lead to research questions about the decay of these micro-architectures within the software. Seminal research by Izurieta [39] uncovered the existence of modular grime, a type of design pattern decay. These results motivated this research to further explore the phenomenon of modular grime.

In this thesis we presented a taxonomy of modular grime based on three common characteristics of relationships: strength, scope, and direction. We also conducted empirical research to determine whether each grime type increased linearly over time in 3 open source software systems.

The results show that one type of modular grime, TEEG, has a statistically significant linear relationship with time in all systems studied. This confirms earlier results that found modular grime build up, but it also shows that one specific type of modular grime build up may be more prevalent.

The effects of TEEG buildup over time on other factors like system maintainability and developer effort are unknown. Future empirical research should study this in order to determine the tangible effects of TEEG buildup. Additional empirical research is also necessary to confirm the results of this study.

Some results revealed significant buildup of other modular grime categories such as PEAG, TEAG, and PEEG. These results were inconsistent between systems and warrant further investigation in other software systems, including commercial systems. The insignificance of results for internal grime (TIG and PIG) should be confirmed in future research as well.

Although we aggregated grime counts during our hypothesis testing, the results were presented by individual design pattern. Although individual differences between design patterns were found, no generalizations can be made about differences in similar patterns across systems. It is possible that modular grime affects each design pattern differently. This research question remains open.

REFERENCES CITED

[1] S. Ali, and O. Maqbool, “Monitoring software evolution using multiple types of changes,” *Int. Conf. Emerg. Technol.,* pp. 410-415, October 2009.

[2] G. Antoniol, R. Fiutem, and L. Christoforetti, “Design pattern recovery in object-oriented software,” Proc. Program Comprehension Workshop, pp. 153-160, June 1998.

[3] Apache Derby. http://db.apache.org/derby/

[4] Apache Software Foundation. http://www.apache.org/

[5] Apache Tomcat. http://tomcat.apache.org/

[6] Apache Web Server. http://httpd.apache.org/

[7] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato, “A comparison of reverse engineering tools based on design pattern decomposition,” *Proc. Australian Softw. Eng. Conf*., pp. 262-269, Mar. 2005.

[8] V.R. Basili, L.C. Briand, and W.L. Melo, “A validation of object-oriented design metrics as quality indicators,” IEEE Trans. Softw. Eng., Vol. 22, pp. 751, Oct. 1996.

[9] V. Basili, et al, “Understanding and predicting the process of software maintenance releases,” *Proceedings of the 18th International Conference on Software Engineering*, pp. 464-474, March 1996.

[10] L.A. Belady, and M.M. Lehman, “Characteristics of Large Systems,” *Proc. Conf. Research Directions in Software Technology*, pp 106-142, 1978.

[11] J. Bieman, D. Jain, and H. Yang, “OO design patterns, design structure, and program changes: an industrial case study,” *IEEE Int. Conf. Softw. Maint*., pp. 580-591, Nov. 2001.

[12] J. Bieman, et al, “Design patterns and change proneness: an examination of five evolving systems,” *Proceedings. Ninth International Software Metrics Symposium*, pp. 40-49, Sept. 2003.

[13] B. Boehm, “The changing nature of software evolution,” *IEEE Software*, Vol. 27, pp. 26-28, Aug. 2010.

[14] L.C. Briand, J.W. Daly, and J.K. Wust, “A unified framework for coupling measurement in object-oriented systems,” *IEEE. Trans. Software. Eng*., Vol. 25, pp. 91-121, Jan. 1999.

[15] L. Briand, P. Devanbu, and W. Melo, “An investigation into coupling measures for c++,”, *Proc. Intl. Conf. Softw. Eng*., pp. 412-421, May 1997.

[16] Browse-by-Query. http://browsebyquery.sourceforge.net/

[17] S.R. Chidamber, and C.F. Kemerer, “A metric suite for object-oriented design,” *IEEE Trans. Softw. Eng*., Vol. 20, pp. 476-493, June 1994.

[18] C.K.S. Chong Hok Yuen, "On analytic maintenance process data at the global and the detailed levels: A case study," *Conf. on Software Maint. Proc.*, pp 248-255, 1988.

[19] S. Cook, J. He, and R. Harrison, "Dynamic and static views of software evolution*," IEEE Int. Conf. Softw. Maint. ICSM*, pp 592-601, Nov. 2001.

[20] C. Cook, and A. Roesch, "Real-time software metrics," *Journal of Systems and Software*, Vol. 24, pp. 223-237, March 1994.

[21] K. Dae-kyoo, and L. Lu, "Inference of design pattern instances in UML models via logic programming," *Proc. IEEE Int. Conf. Eng. Complex Comput. Syst*., pp. 47-56, Aug. 2006.

[22] K. Dae-kyoo, and W. Shen, "An approach to evaluating structural pattern conformance of UML models," *Proc. ACM Symp. Appl. Computing*, pp. 1404-1408, March 2007.

[23] M. Dagpinar and J. Jahnke, "Predicting maintainability with object-oriented metrics – an empirical comparison," *Rev. Eng. Working Conf. Proc*., pp. 155-164, Nov. 2003.

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

[25] M. Di Penta, L. Cerulo, Y. Gueheneuc, and G. Antoniol, "An empirical study of the relationships between design pattern roles and class change proneness," *IEEE Int. Conf. Softw. Maint*., pp. 217-226, Sept. 2008.

[26] Dr. Java. http://drjava.org/

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

[28] S. Eick, et al, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. Software Eng*., Vol. 27, pp. 1-12, Jan. 2001.

[29] R.B. France, D.K. Kim, E. Song, and S. Ghosh, "A UML-Based Pattern Specification Technique," *IEEE. Trans. Software. Eng*., Vol. 30, pp. 193-206, March 2004.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reuseable Object-Oriented Software*, Addison Wesley, 1994.

[31] GNU GPL. http://www.gnu.org/licenses/gpl.html

[32] M. Godfrey, and D. German, "The past, present, and future of software evolution," *IEEE International Conference on Software Maintenance*, pp. 129-138, Sept. 2008.

[33] M. Godfrey, and Q. Tu, "Growth, evolution, and structural change in open source software*," Int. Workshop Princ. Softw. Evol*., pp. 103-106, Sept. 2001.

[34] Y.G. Gueheneuc and G. Antoniol, "DeMIMA: a multi-layered framework for design pattern identification," *IEEE. Trans. Software. Eng*., Vol. 34, pp. 667-684, Sept. 2008.

[35] J. Gustafsson, J. Paakki, L. Nenonen, and A. Verkamo, "Architecture-centric software evolution by software metrics and design patterns," *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pp. 108-115, March 2002.

[36] Halstead's Complexity Measures. http://en.wikipedia.org/wiki/Halstead_complexity_measures

[37] I. Herraiz, et al, "Comparison between SLOCs and number of files as size metrics for software evolution analysis," *Proc. Eur. Conf. Software Maint. Reeng*., pp. 206-213, March 2006.

[38] M. Hitz and B. Montazeri, "Measuring product attributes of object-oriented systems," *Proc. European Software Eng. Conf*., pp. 124-136, Sept. 1995.

[39] C. Izurieta, "Decay and Grime Buildup in Evolving Object Oriented Design Patterns," Ph. D. Dissertation, Dept. Comp. Sci., Fort Collins, CO, Colorado State University, 2009.

[40] C. Izurieta, and J. Bieman, "The evolution of FreeBSD and Linux," *Proc. ACM-IEEE Int. Symp. Empir. Softw. Eng.*, pp. 204-211, Sept. 2006.

[41] C. Izurieta, and J. Bieman, "Testing Consequences of Grime Buildup in Object Oriented Design Patterns," *Proc. Int. Conf. Softw. Test., Verif. Validation*, pp. 171-179, April 2008.

[42] C. Izurieta, and J. Bieman, "How Software Designs Decay: A Pilot Study of Pattern Evolution," *Proc. Int. Symp. Empir. Softw. Eng. Meas*., pp. 449-451, Sept. 2008.

[43] Java. http://www.java.com/en/

[44] JDBC. http://www.oracle.com/technetwork/java/javase/jdbc/index.html#corespec40

[45] JDepend. http://www.clarkware.com/software/JDepend.html

[46] JHawk. http://www.virtualmachinery.com/jhawkprod.htm

[47] JQuery. http://eclipse-plugins.info/eclipse/plugin_details.jsp?id=368

[48] C. Kemerer, and S. Slaughter, "Empirical approach to studying software evolution," *IEEE Trans. Software Eng*., Vol. 25, pp 493-509, Aug. 1999.

[49] W. Li and S. Henry, "Object-oriented metrics that predict maintainability,*" Journal of Systems and Software*, Vol. 23, pp. 111-122, Nov. 1993.

[50] W. Li, J. Talburt, and Y. Chen, "Quantitatively evaluate object oriented software evaluation," *Proceedings. Technology of Object-Oriented Languages*, pp. 362-367, Sept. 1997.

[51] L. Liu, "EVOLVE: adaptive specification techniques of object-oriented software evolution," *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, Vol. 5, pp. 396-405, Jan. 1998.

[52] M.M. Lehman, "Laws of program evolution – rules and tools for program management," *Proc. Infotech State of the Art Conf*., pp. 11/1-11/25, April 1978.

[53] M.M. Lehman, "On understanding laws, evolution, and conservation in the large program life cycle," *Journal of Systems and Software*, vol. 1, pp. 213-231, 1980.

[54] M.M. Lehman, “Programs, lifecycles, and laws of software evolution,” Proc. *IEEE Spec. Iss. on Softw. Eng*., vol. 68, pp. 1060-1076, Sept. 1980.

[55] M.M. Lehman, D.E. Perry, and J.F. Ramil, “On evidence supporting the FEAST hypothesis and the laws of software evolution,” *Int. Software Metrics Symp. Proc*., pp 84-88, Nov. 1998.

[56] M.M. Lehman, et al, “Metrics and laws of software evolution – the nineties view,” *Int. Software Metrics Symp. Proc*., pp 20-32, Now. 1997.

[57] C.S. Ma, C.K. Chang, and J. Cleland-Huang, “Measuring the intensity of object coupling in c++ programs,” *Proc IEEE Comput. Soc. Int. Comput. Software. Appl. Conf*., pp. 538-543, October 2001.

[58] T.J. McCabe, “Structured testing: A software testing methodology using the cyclomatic complexity metric,” Nat. Bur. Stand. Washington, DC, Rep. NBS-SP-500-99, 1982.

[59] G.J. Myers, *Composite/Structural Design*, New York: Nordstrand Reinhold, 1978.

[60] A.P. Nikora, and J.C. Munson, “Understanding the nature of software evolution,” *Proceedings International Conference on Software Maintenance*, pp. 83-93, Sept. 2003.

[61] ObjectAid. http://www.objectaid.com/

[62] M. Ohlsson, A. Mayrhauser, B. McGuire, & C. Wohlin, “Code decay analysis of legacy software through successive releases,” *Proc. IEEE Aerospace Conf*., pp. 69-81. March 1999.

[63] D.E. Perry, “Dimensions of software evolution,” *Proceedings. International Conference on Software Maintenance*, pp 296-303, Sept. 1994.

[64] L. Prechelt, et al, “A controlled experiment in maintenance comparing design patterns to simpler solutions,” *IEEE Trans. Software Eng.,* Vol. 27, pp. 1134-1144, Dec. 2001.

[65] J.F. Ramil, and M.M. Lehman, “Metrics of software evolution as effort predictors – a case study,” *Conference on Software Maintenance*, pp. 163-172, October 2000.

[66] T. Schanz and C. Izurieta, “Object oriented design pattern decay: a taxonomy,” *Proce. ACM-IEEE Int. Symp. Empir. Softw. Eng. Measur*., Sept. 2010.

[67] N. Shi and R.A. Olsson, “Reverse engineering of design patterns from Java source code,” *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng*., pp. 123-132, Sept. 2006.

[68] SourceForge. http://sourceforge.net/

[69] Structured Query Language. http://www.sql.org/

[70] W.M. Turski, “Reference model for smooth growth of software systems,” *IEEE Trans. Softw. Eng*., Vol. 22, pp. 599-600, August 1996.

[71] Tyruba. http://tyruba.sourceforge.net/

[72] M. Vokac, “Defect frequency and design patterns: an empirical study of industrial code,” *IEEE Trans. Software Eng*., Vol. 30, pp. 904-917, Dec. 2004.

[73] M. Vokac, et al, “A controlled experiment comparing the maintainability of programs designed with and without design patterns - a replication in a real programming environment,” *Empirical Software Engineering*, Vol. 9, pp. 149-195, Sept. 2004.

[74] Vuze. http://www.vuze.com/

[75] H. Zhang, and S. Kim, “Monitoring software quality evolution for defects,” *IEEE Software*, Vol. 27, pp. 58-64, Aug. 2010.

APPENDICES

APPENDIX A

DYNAMIC CHART DOCUMENTATION

This appendix contains the documentation for the current version of the Dynamic Chart program. It describes the strategy that Dynamic Chart uses to chart data as well as details on how to use the program.

<u>Introduction</u>

Dynamic Chart is a program used to display linear combinations of values in real time. The basic display of a Dynamic Chart is a time series data set. Each point in time contains the result of some linear combination of values. The user has the ability to change coefficients in the linear combination using sliders. Each point can be described mathematically by the following linear equation.

$$G_t = \alpha_1 PEAG_t + \alpha_2 PEEG_t + \alpha_3 TEAG_t + \alpha_4 TEEG_t$$

Where G_t represents a singular grime count at time t. $PEAG_t$, $PEEG_t$, $TEAG_t$, and $TEEG_t$ are the actual grime counts for each type of modular grime at time t and α_1 through α_4 are the linear coefficients that are independent of time. Each point in time has one corresponding value per group and there exists a G_t for each group at each time t.

<u>File input</u>

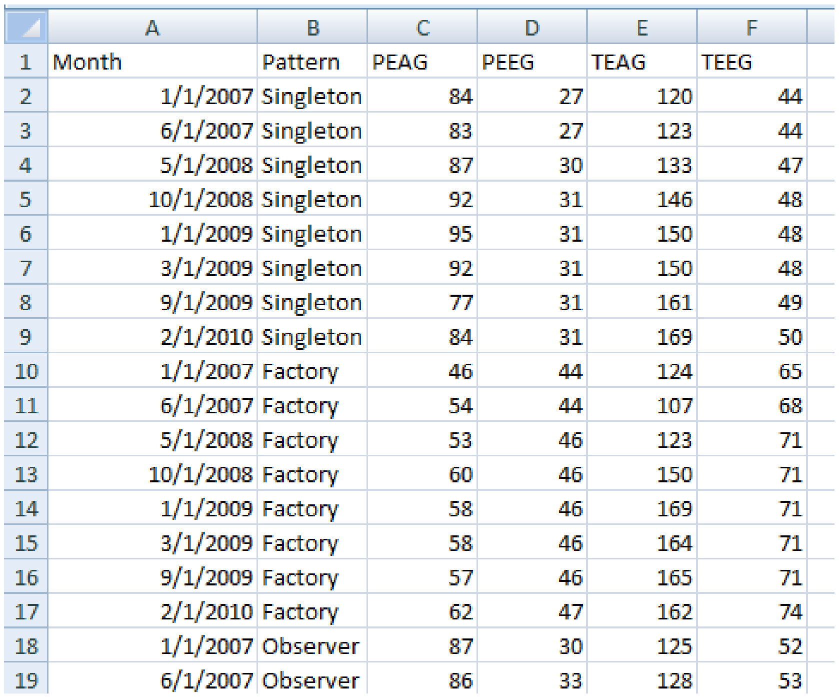
Data is input using File -> Load. Currently data is loaded only from .csv files. The format of the input file has a few requirements.

1. The first column contains dates of the form ―MM/DD/YYYY”

2. The second column contains the name for each time series. This is useful when inputting several different time series groups.

3. The remaining columns are composed of values for each variable in the linear combination. The name of the variable is denoted in a header.

Figure 22 shows an example input file. The first column contains dates. The second column contains the name for each pattern that contains a series of data. In this case there are several different groups of data taken for the same series of time. The remaining columns are used for each variable in the linear combination. In this case there is a linear combination of four variables: PEAG, PEEG, TEAG, and TEEG. When this file is loaded, a data set and group are created for each name. There is a data set named PEAG in group PEAG and so on. Other data, such as PIG and TIG, can be added easily in additional columns.



	A	B	C	D	E	F
1	Month	Pattern	PEAG	PEEG	TEAG	TEEG
2	1/1/2007	Singleton	84	27	120	44
3	6/1/2007	Singleton	83	27	123	44
4	5/1/2008	Singleton	87	30	133	47
5	10/1/2008	Singleton	92	31	146	48
6	1/1/2009	Singleton	95	31	150	48
7	3/1/2009	Singleton	92	31	150	48
8	9/1/2009	Singleton	77	31	161	49
9	2/1/2010	Singleton	84	31	169	50
10	1/1/2007	Factory	46	44	124	65
11	6/1/2007	Factory	54	44	107	68
12	5/1/2008	Factory	53	46	123	71
13	10/1/2008	Factory	60	46	150	71
14	1/1/2009	Factory	58	46	169	71
15	3/1/2009	Factory	58	46	164	71
16	9/1/2009	Factory	57	46	165	71
17	2/1/2010	Factory	62	47	162	74
18	1/1/2007	Observer	87	30	125	52
19	6/1/2007	Observer	86	33	128	53

Figure 22. A sample input

<u>The Main Display</u>

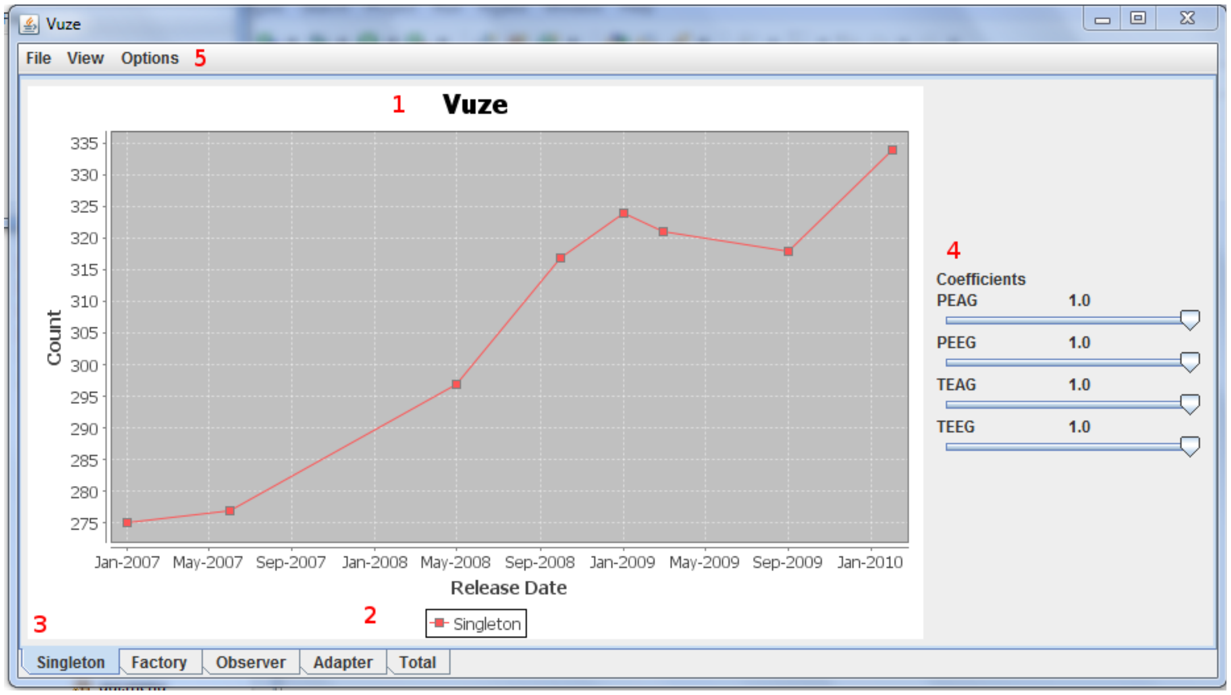


Figure 23. The main display

Once a data file is selected, the main display opens. Figure 2 shows the main display. The parts of the main display, marked by numbers, are each described below.

1. The title for the data collection given by the user during the file input process

2. The X-axis label and legend. The legend is useful in the case when multiple time series‘ are displayed together.

3. The chart tabs. Each tab contains a different chart. The tabs are determined by the grouping of the data. The title of each tab is a group name and the chart contained within contains all data sets in the group.

4. The slider panel is where the user can modify the linear combination. There is a slider for each variable (or column in the input file). Moving the slider modifies the coefficient for that variable as demonstrated in section 1.

5. The menus give the user more options:

File: The user can save the current chart or load more data.

View: Allows the user to changes tabs

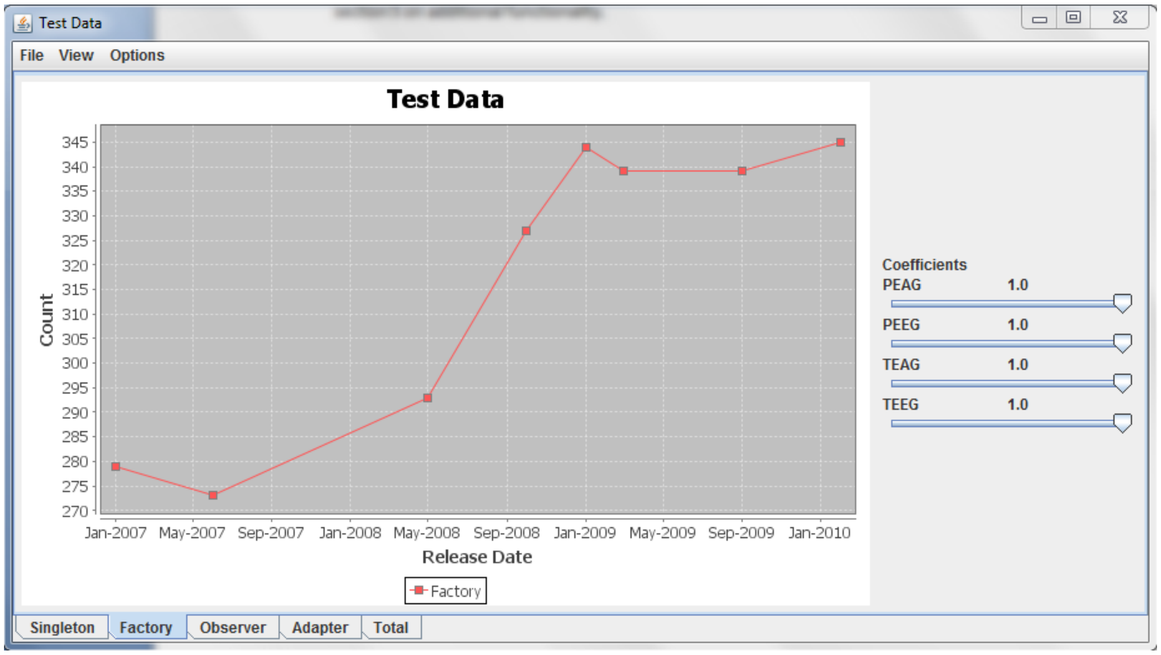
Options: Contains options to modify the data and the data groupings. These are discussed in section 5 titled additional functionality.

File Output

File -> Save allows charts to be saved as image (PNG) files. When this option is selected the current visible chart is saved.

Additional Functionality

The options menu contains some extra functions to modify the data as well as to modify the way data is displayed. There are two options: ―Modify data" and ―Modify chart groupings". ―Modify data" allows the user to modify the underlying input data while ―Modify chart groupings" allows the user to modify how the data sets are grouped and therefore how charts are composed (i.e. which data sets are displayed together). Each function is demonstrated next. For examples, the original test data display is shown in figure 24.


Test Data
File View Options
Test Data
Count
345
340
335
330
325
320
315
310
305
300
295
290
285
280
275
270
Jan-2007 May-2007 Sep-2007 Jan-2008 May-2008 Sep-2008 Jan-2009 May-2009 Sep-2009 Jan-2010
Release Date
Factory
Singleton Factory Observer Adapter Total

Coefficients

PEAG	1.0
PEEG	1.0
TEAG	1.0
TEEG	1.0

Figure 24. A typical chart display

<u>Modify Data</u>

When the user wishes to modify the data, the following screen is displayed:

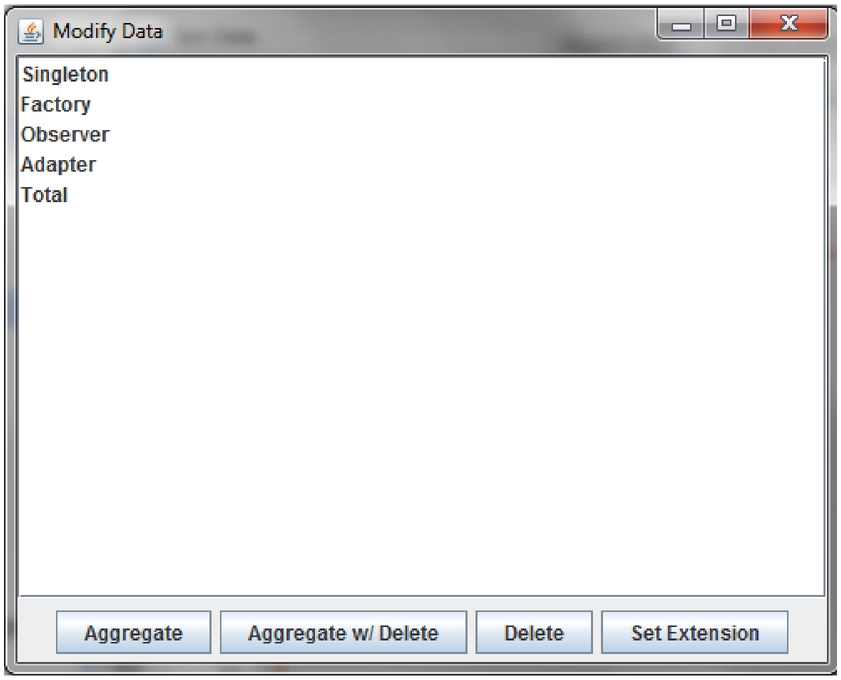


Figure 25. The Modify Data display

All current data sets are displayed in the selection box. The user may select any number of them and perform the following operations.

1. *Aggregate*: this aggregates the values for multiple data sets by date. For each date in the data sets, the values of each variable are added together. As an example, if the user wants to aggregate the Singleton (S) data and the Factory (F) data into a new data set (F, S), this can be shown mathematically as follows, where the subscript t represents a single date.

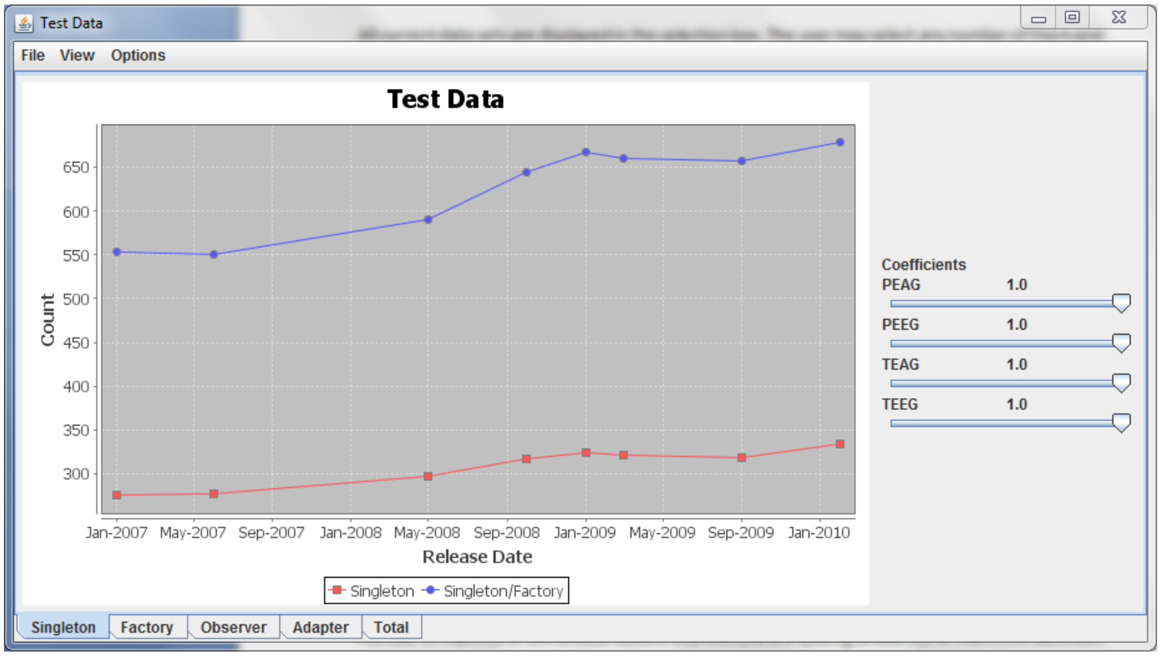
$G(S)_t = \alpha_1 PEAG(S)_t + \alpha_2 PEEG(S)_t + \alpha_3 TEAG(S)_t + \alpha_4 TEEG(S)_t$; $G(S)_t$ represents the Singleton data

$G(F)_t = \alpha_1 PEAG(F)_t + \alpha_2 PEEG(F)_t + \alpha_3 TEAG(F)_t + \alpha_4 TEEG(F)_t$; $G(F)_t$ represents the Factory data

$$G(F, S)_t = \alpha_1[PEAG(F)_t+PEAG(F)_t] + \alpha_2[PEEG(F)_t+PEEG(F)_t] + \alpha_3[TEAG(F)_t+TEAG(F)_t] + \alpha_4[TEEG(F)_t+TEEG(F)_t]$$

The sets that were aggregated continue to exist after this operation. NOTE: Each data set chosen must contain the same series of dates, otherwise an error occurs.

Figure 26 shows the results of the preceding aggregation after application to the test data shown in figure 24. A new data set titled ―Singleton/Factory” is included in the chart.



Coefficients

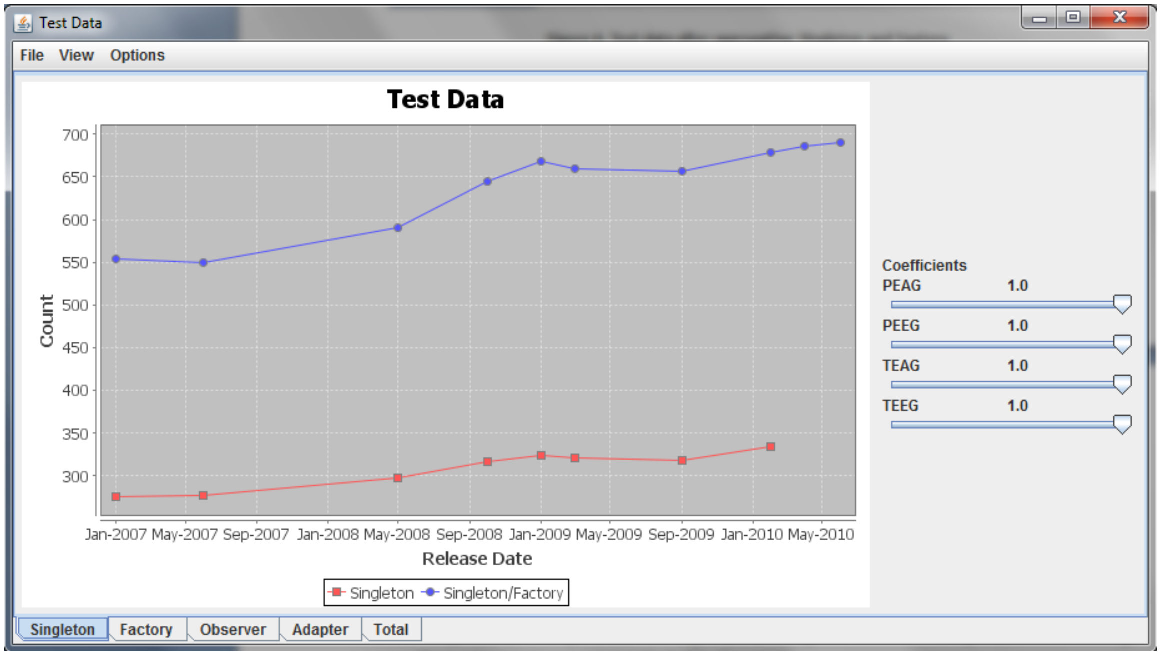
PEAG	1.0
PEEG	1.0
TEAG	1.0
TEEG	1.0

Figure 26. Test data after aggregating Singleton and Factory

2. Aggregate w/ Delete: The same as aggregate, but the original data sets are deleted after aggregation.

3. Delete: Removes a data set from the data collection

4. Set Extension: This sets a predictive extension for the data set(s). An extension is defined by an extra number or marks (# of extra observations to predict) and a spacing amount (# of months in between each extra mark). The program fits several different regressions to the existing data. The most accurate one is used to predict the extra data points (also called marks), which are then displayed along with the original data. Figure 27 shows an extension of the ―Factory/Singleton" data from figure 24. Two extra data points were predicted, each separated by 2 months, resulting in a new data points in April and June 2010.



Coefficients

PEAG	1.0
PEEG	1.0
TEAG	1.0
TEEG	1.0

Figure 27. Test data after extension.

Modify Chart Groupings

When the user chooses to modify how different data sets are displayed together, the following menu appears.

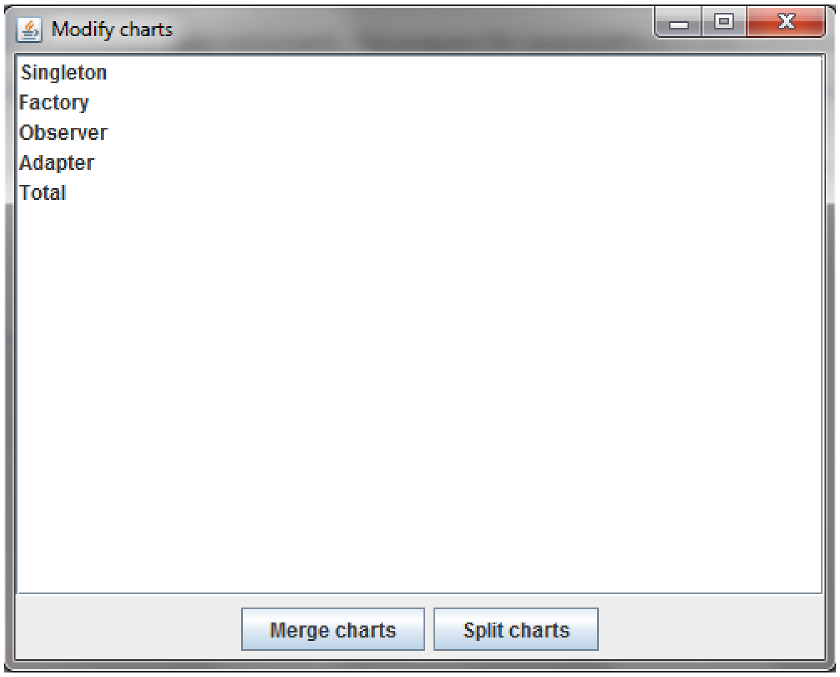


Figure 28. The modify chart groupings display

The selection box contains the group names for the charts. Each one corresponds to a tab in the main display. Since there may be multiple data sets in a group (i.e. displayed together on one chart), these choices can differ from those in the ―modify data" display. In this example, the Singleton data group actually contains two data sets: Singleton and Singleton/Factory, which was created earlier through aggregation. There are two types of operations in this display:

1. Split Charts: Splits a group(s) by allowing the user to extract data sets from existing groups and put them into a new group. When multiple groups are selected the user can extract data sets from each and form a new group.

Extending the ongoing example, the user may wish to put the Singleton/Factory data set in its own chart. After selecting the Singleton data group and choosing —Split charts", the following display is shown:

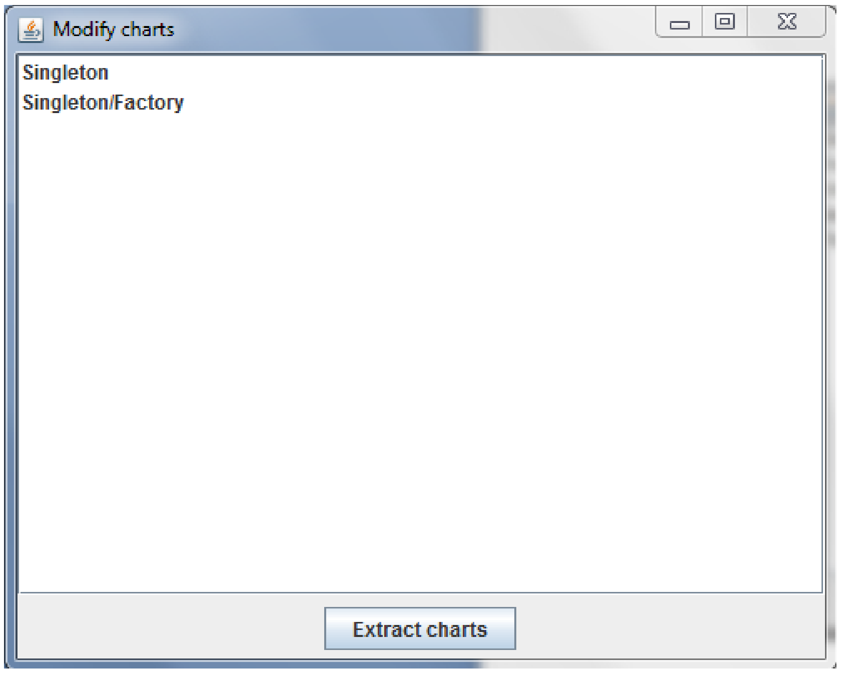


Figure 29. A window displaying the members of a data group

After selecting Singleton/Factory and clicking —Extract charts", the user is prompted for a new group name. The result of this action is shown in figure 29. Singleton/Factory is now displayed alone in a group named —New chart" while the Singleton group now contains only the Singleton data.

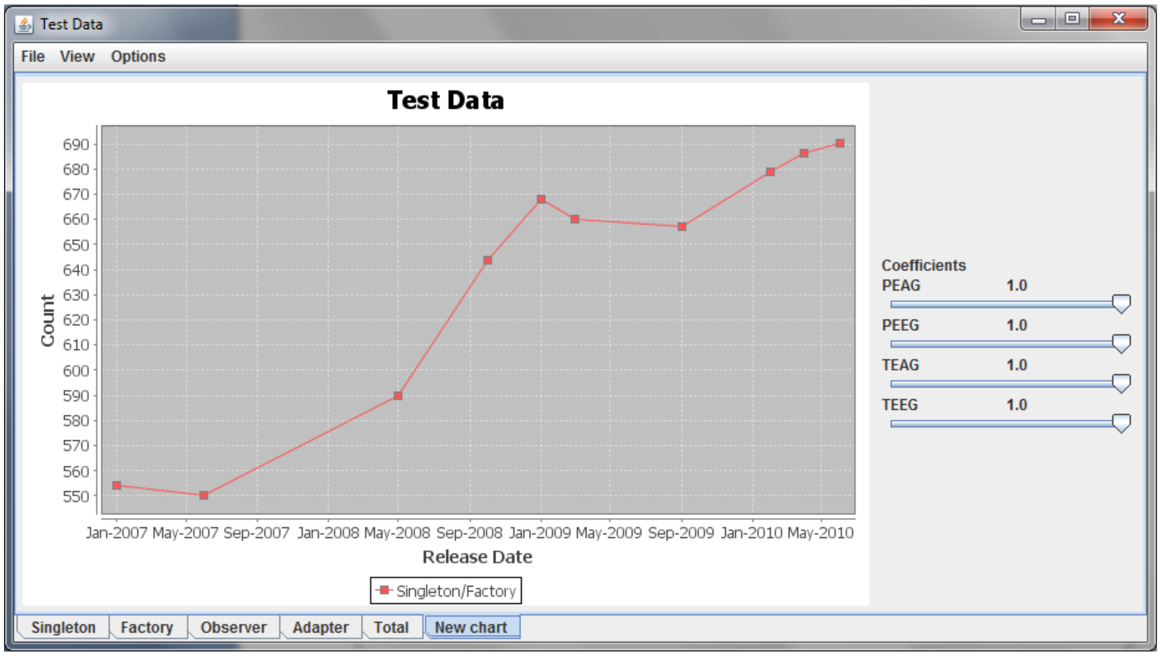
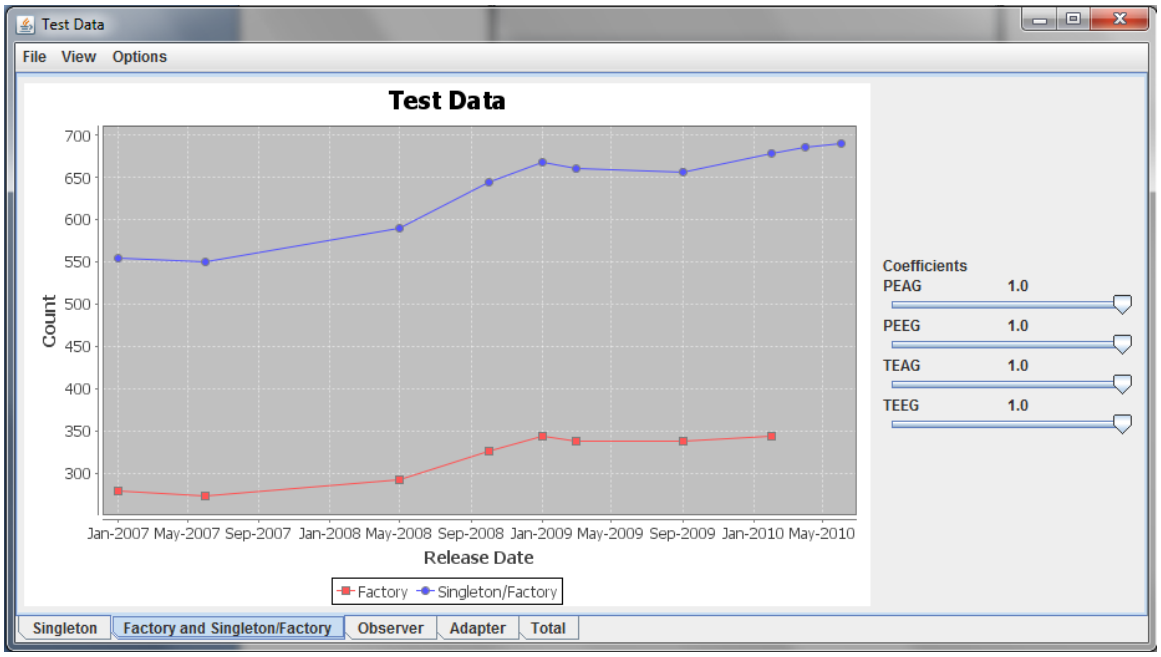


Figure 30. The results of splitting a data group

2. Merge Charts: Merges the selected groups by taking each data set in the groups and combining them into one new group as selected by the user. It provides a convenient way to compare different data sets.

The user may wish to now group the Singleton/Factory data in the group ―New chart" with the Factory data. By selecting ―Modify Chart Grouping" from the menu, the user can now select the groups of interest and click the ―Merge charts" button shown in figure 28. The resulting group, with the name ―Factory and Singleton/Factory" is shown in figure 31.



Coefficients
PEAG 1.0
PEEG 1.0
TEAG 1.0
TEEG 1.0

Figure 31. The results of merging two groups.

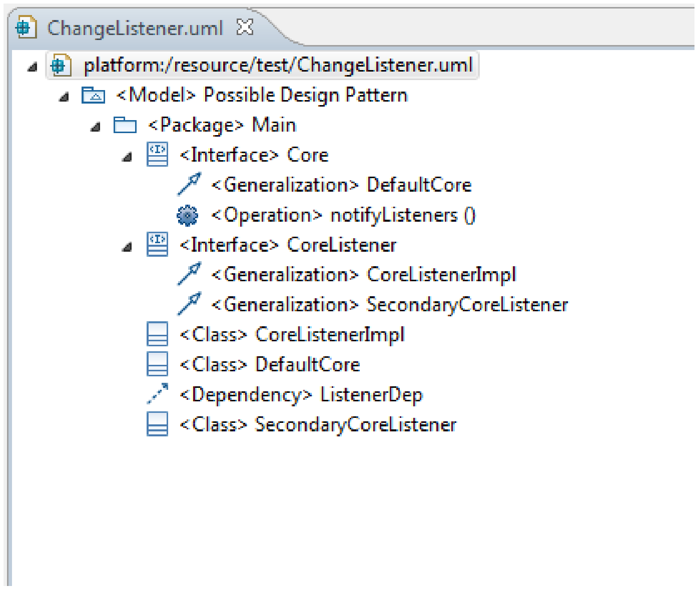
APPENDIX B

RBML CONFORMANCE CHECKER DOCUMENTATION

This appendix contains a summary of a new and unfinished tool used to check the conformance of a candidate design pattern to an RBML diagram. Automatic checking of design pattern conformance to an RBML diagram is more time efficient and less error prone than manual checking. The tool is an Eclipse plug-in which implements the RBML conformance checking algorithm given in [22].

Input

The RBML conformance checker takes as input an RBML diagram along with a UML class diagram. The RBML diagram provides a formal definition of a design pattern. In the current version, RBML diagrams are hard coded into the software. The UML class diagram is encoded under the Eclipse Modeling Framework (EMF) UML2.0 specification. Figure 32 shows an example input.



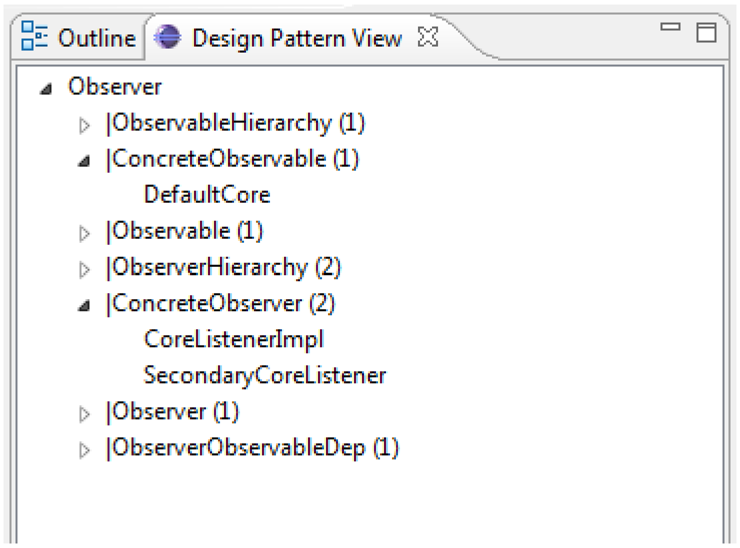
```
ChangeListener.uml
platform:/resource/test/ChangeListener.uml
  <Model> Possible Design Pattern
    <Package> Main
      <Interface> Core
        <Generalization> DefaultCore
        <Operation> notifyListeners ()
      <Interface> CoreListener
        <Generalization> CoreListenerImpl
        <Generalization> SecondaryCoreListener
      <Class> CoreListenerImpl
      <Class> DefaultCore
      <Dependency> ListenerDep
      <Class> SecondaryCoreListener
```

Figure 32. Eclipse UML input

The example shows a candidate Observer pattern realization created for demonstration purposes. There are two interfaces, Core and CoreListener, which are candidates for the Observable and Observer roles respectively. There are also concrete classes that implement the interfaces.

Demonstration

When the user right clicks on the file ChangeListener.uml shown in figure 32, the menu reveals an option to ―Map Design Patterns". This causes the plug-in to attempt a conformance check between the UML class diagram and all RBML pattern definitions (an Observer RBML definition for this demonstration). If a conforming pattern is found within the UML file, the ―Design Pattern View" is shown. Figure 33 shows the results of checking conformance of ChangeListener.uml to the RBML definition for the Observer Pattern.



```
Observer
    |ObservableHierarchy (1)
    |ConcreteObservable (1)
        DefaultCore
    |Observable (1)
    |ObserverHierarchy (2)
    |ConcreteObserver (2)
        CoreListenerImpl
        SecondaryCoreListener
    |Observer (1)
    |ObserverObservableDep (1)
```

Figure 33. The Design Pattern View

The results from a conformance check are show in a tree view. The root of the tree displays the pattern name. Each child of the root is a role in the pattern‘s RBML definition. In this case there are 7 roles that must be fulfilled from ―ObservableHierarchy” to ―ObserverObservableDep”. The children of each role are parts in the UML class diagram that play the corresponding role in the conformance map. For example, the class ―DefaultCore” from the UML diagram plays the role of ―ConcreteObservable” in the RBML pattern definition. The number in parentheses next to each role displays the number of mappings for said role.

APPENDIX C

RBML DIAGRAMS

This appendix includes RBML diagrams for each design pattern studied. The diagrams formally represent pattern definitions and were used to determine conformance of candidate realizations found in the software studied.

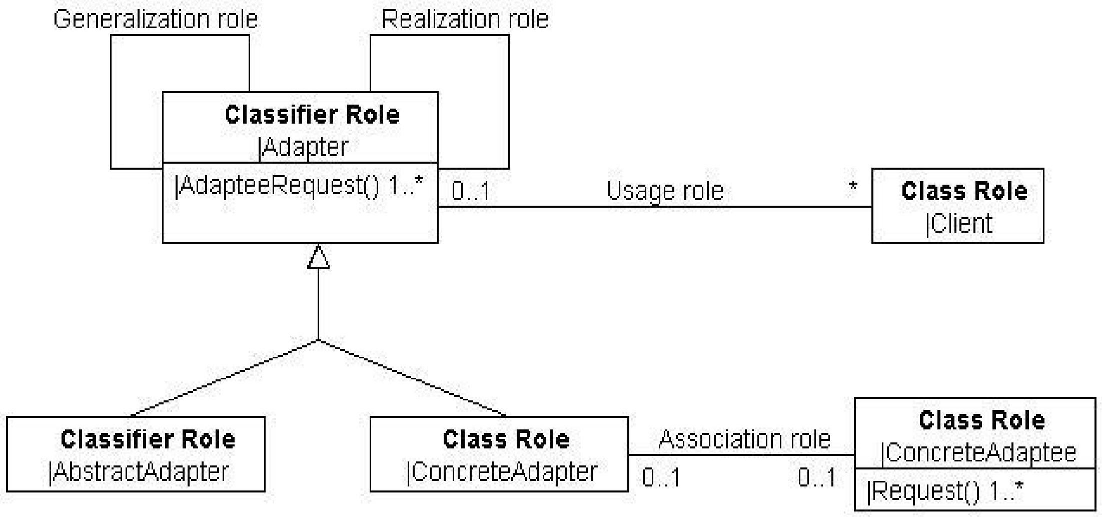


Figure 34. The Adapter Pattern

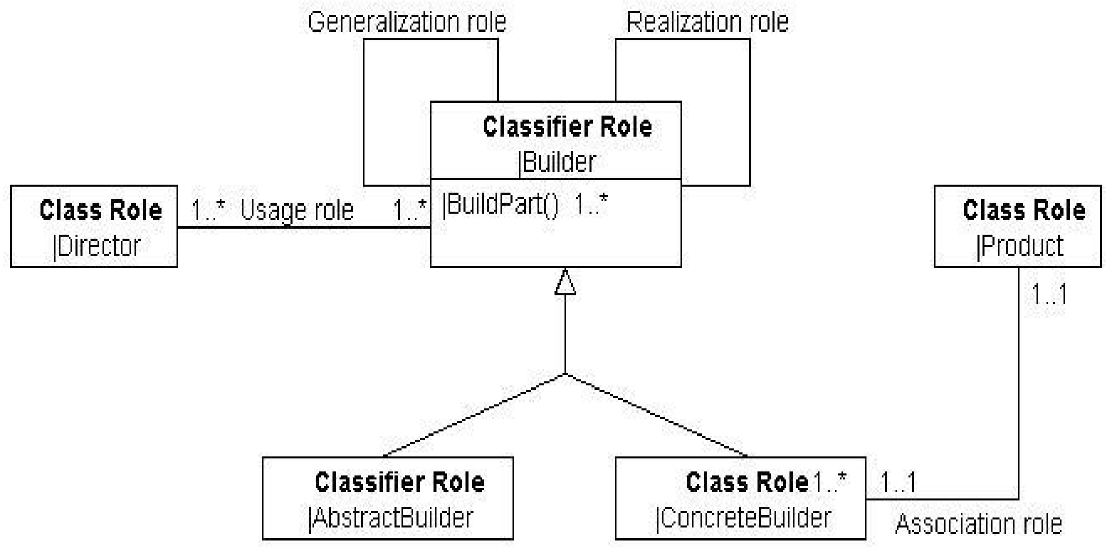


Figure 35. The Builder Pattern

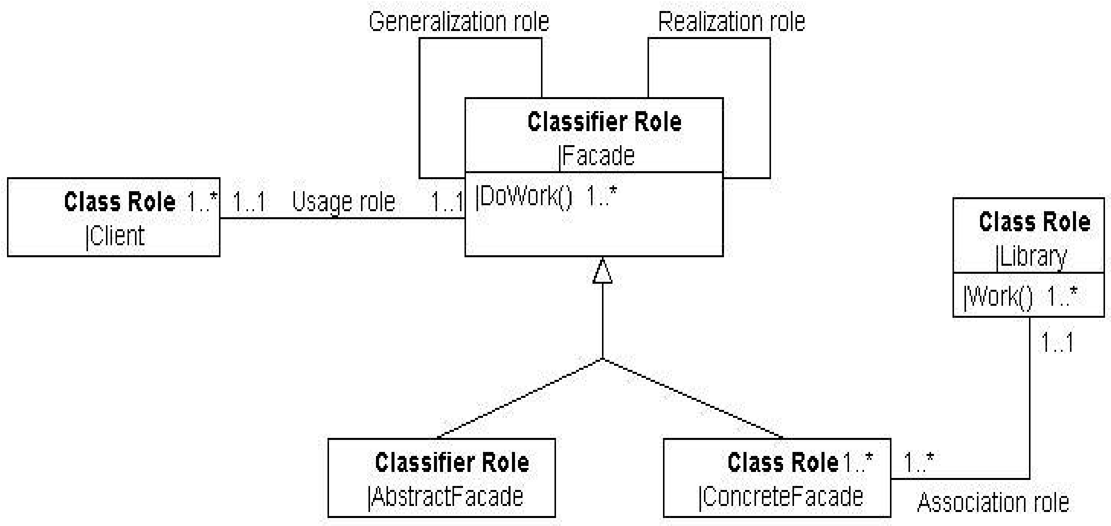


Figure 36. The Façade Pattern

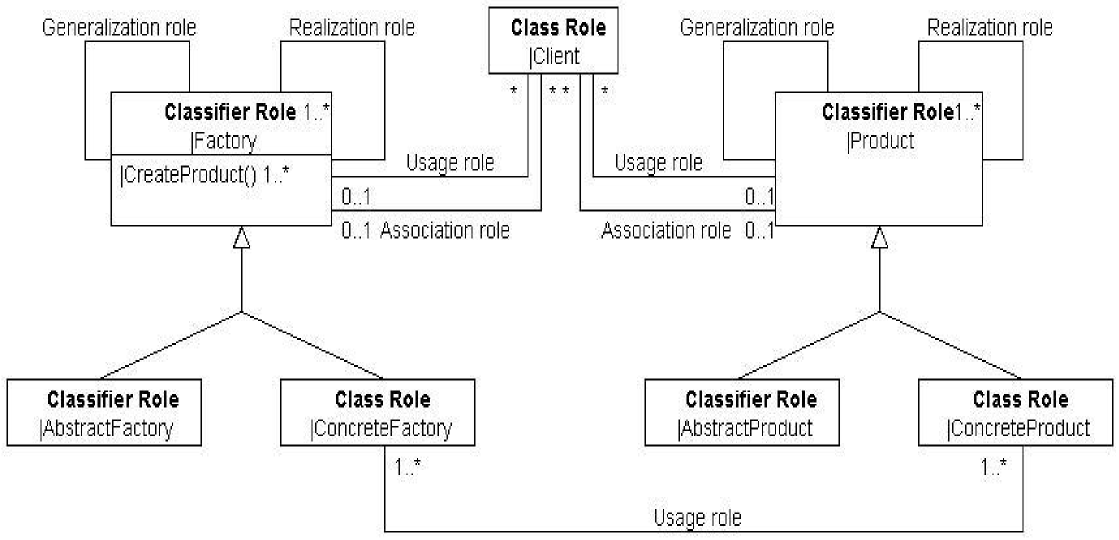


Figure 37. The Factory Pattern

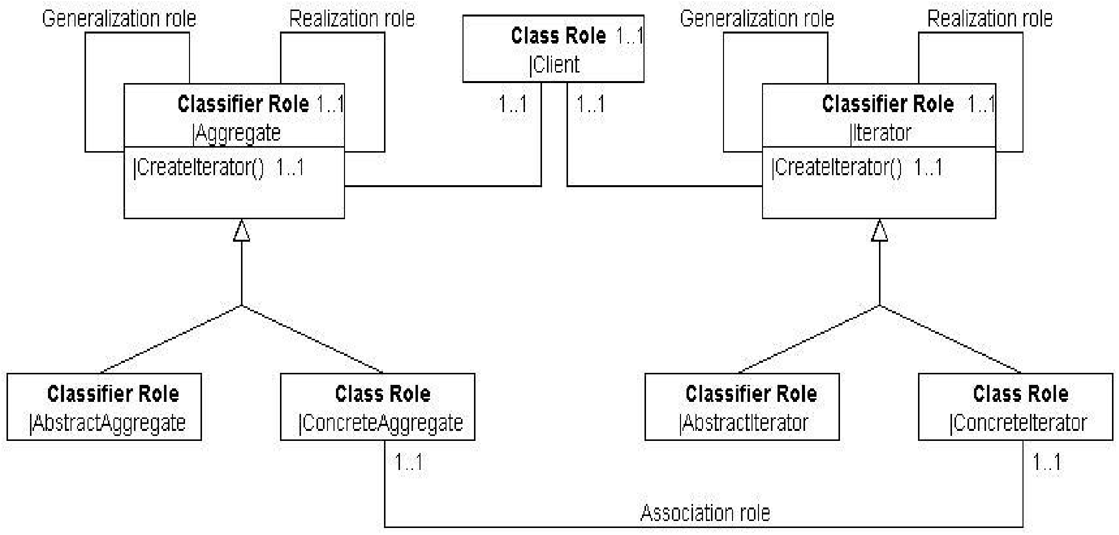


Figure 38. The Iterator Pattern

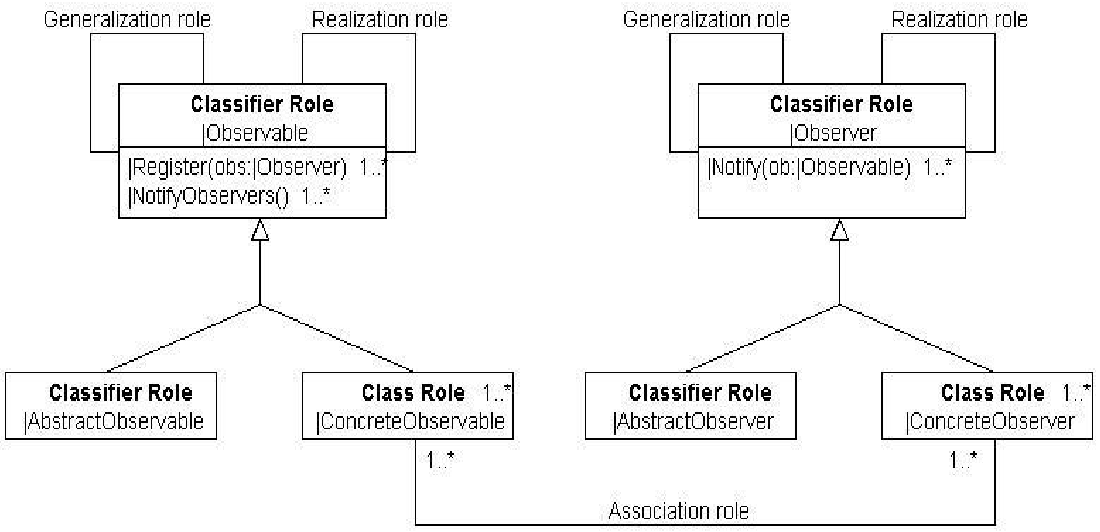


Figure 39. The Observer Pattern

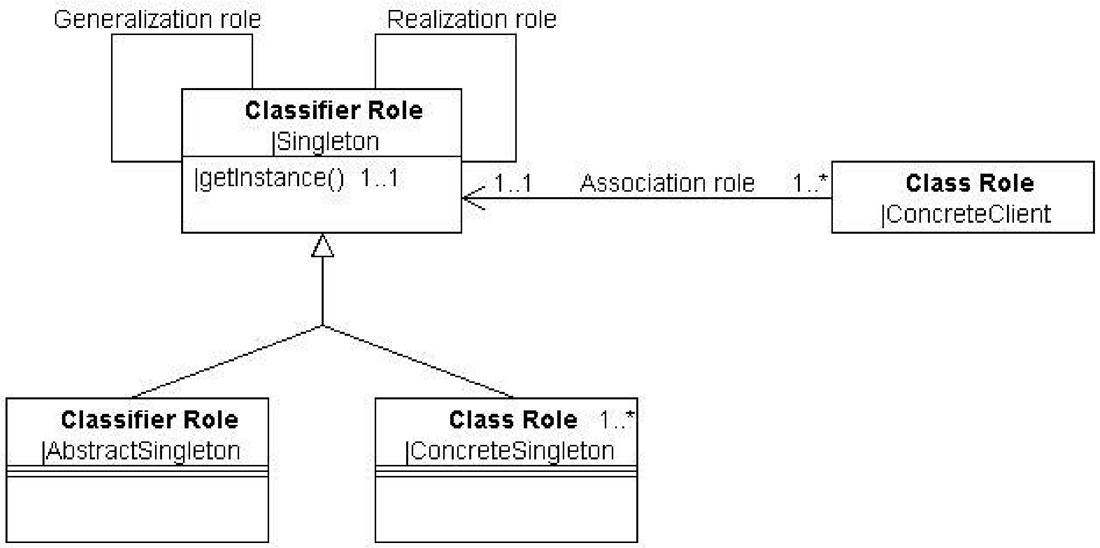


Figure 40. The Singleton Pattern

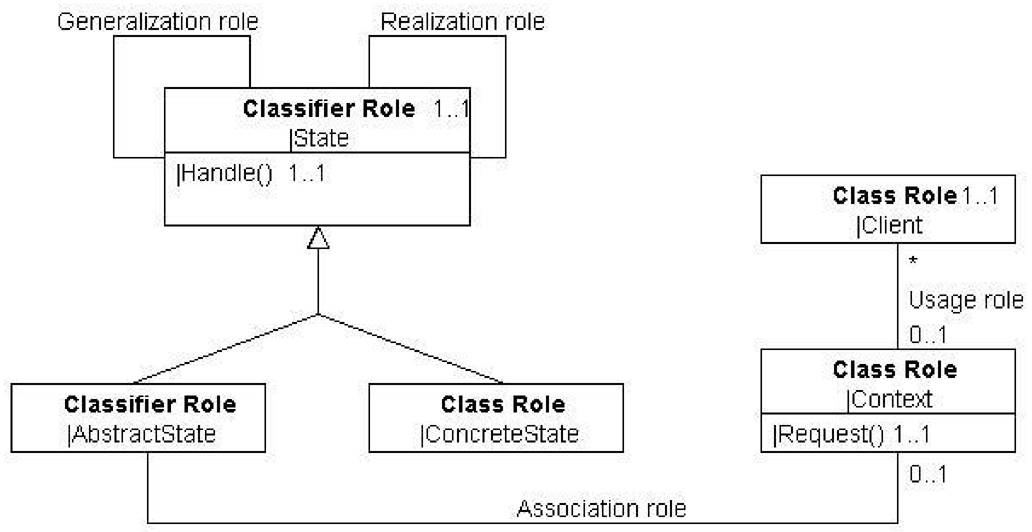


Figure 41. The State Pattern

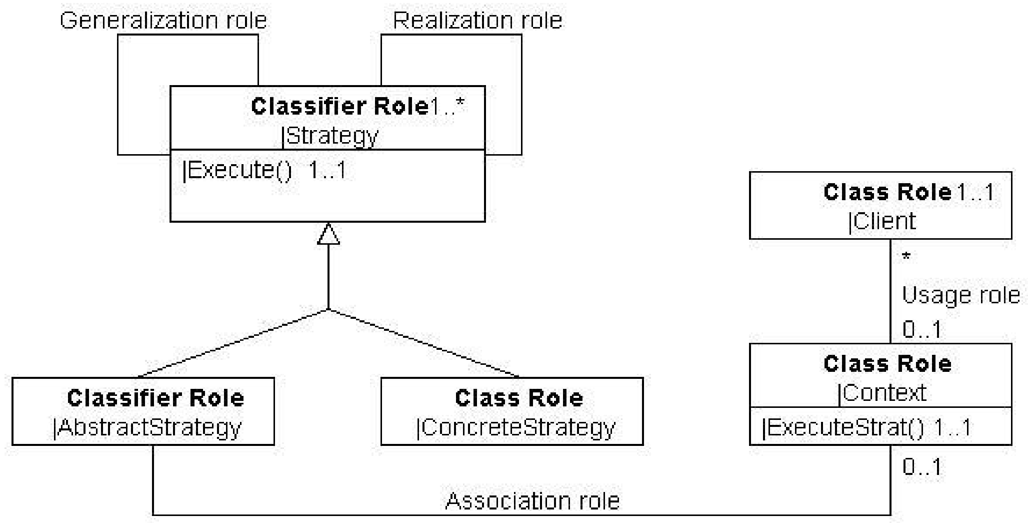


Figure 42. The Strategy Pattern

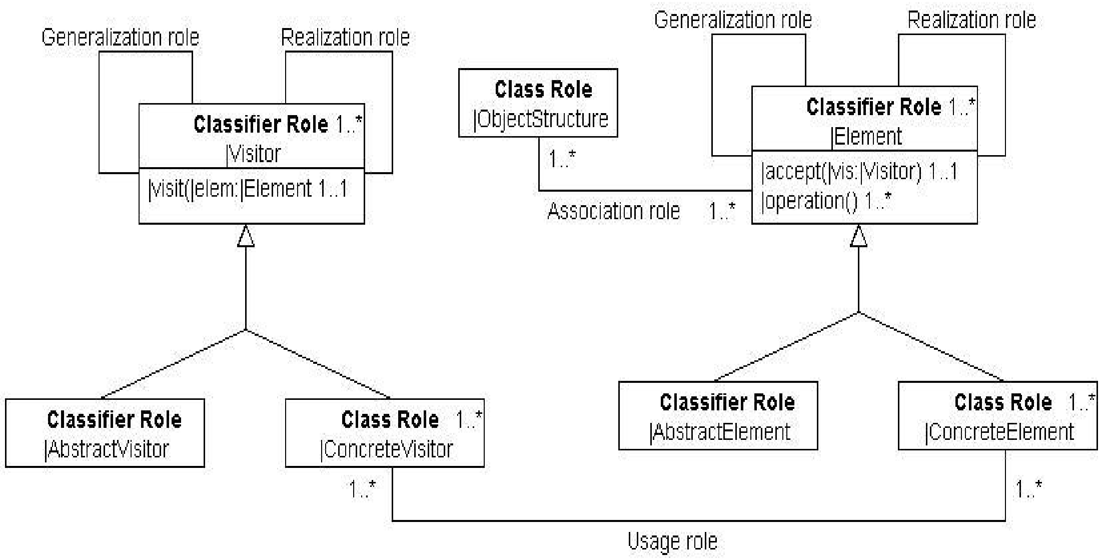


Figure 43. The Visitor Pattern