Object Oriented Design Pattern Decay: A Taxonomy

Travis Schanz
Montana State University
Department of Computer Science
Bozeman, MT 59717
1-406-661-3718
travis.schanz@cs.montana.edu

Clemente Izurieta
Montana State University
Department of Computer Science
Bozeman, MT 59717
1-406-994-3720
clemente.izurieta@cs.montana.edu

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 occur 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. Illustrative examples of grime build-up are provided to demonstrate the taxonomy. A pilot study is used to validate the taxonomy and provide initial empirical evidence of the proposed classification.

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Design — *Design Concepts, Object-oriented design methods*; D.2.11 [**Software Engineering**]: Software Architectures — *Patterns*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement — *Enhancement, Extensibility, Maintainability, Maintenance measurement*.

General Terms
Measurement, Design, Experimentation.

Keywords
Software Architectures, Object Oriented Design Patterns, Software Decay, Software Evolution

1. Introduction
Software systems evolve over time and studies [6] suggest that decay of designs occurs as a result of changes to its functionality and structure. A consequence of decay is an increase in test requirements and an increase in adaptability and maintainability efforts [11]. Studies in software decay focus on the overall design of a system [14]. Measuring decay is thus a difficult problem because surrogate measures [2] must be used to quantify external quality attributes. Attempts to measure decay have been proposed [6]; however indices used by prior studies make it difficult to compare the relative decay of system designs to each other. Izurieta and Bieman [9] however; suggest using design patterns as the underlying **micro-architectures** to study. Design patterns have a well understood form that can be described using formal pattern languages (e.g. RBML [7], PADL [8]), thus providing an agreed upon structure that can be used to measure against. As design patterns evolve, changes to the pattern can be measured to see if the pattern is evolving in the manner in which it was intended. Deviations indicate decay. Empirical studies by Izurieta and Bieman demonstrate a form of decay; **grime**. Their studies suggest that "design patterns do not structurally breakdown, but as designs evolve, design pattern realizations tend to be obscured as new associations develop between classes."

Whilst empirical evidence of design pattern decay and grime buildup is available [10], taxonomy is a natural progression and is essential. A taxonomy promotes the classification of grime into ordered groups that are disjoint and complete while preserving natural relationships between categories. The classification, description and naming of various forms of grime as applicable to each individual design pattern is proposed. This research goes beyond the initial definitions of decay and grime by proposing a taxonomy of design pattern grime.

The paper is organized as follows. Section 2 discusses the details of the taxonomy. Section 3 provides illustrative examples of couplings that contribute to each taxonomical category. Sections 4 and 5 present and discuss data from an initial pilot study focused on validating the taxonomy. Section 6 examines the threats to validity. Conclusions and direction for further research are provided in section 7.

2. Taxonomy of grime
Izurieta and Bieman [9] define three levels of grime; class grime is defined as changes to software classes that belong to a design pattern, but whose functional value is not derived from the way the pattern was meant to be extended. For example, new code added to design pattern classes (e.g. methods or attributes) that are not necessary for pattern function will increase class grime. Modular grime is defined as increases in the internal and external coupling of classes that belong to a pattern. As designs evolve pattern classes can develop new relationships that are unnecessary for pattern operation. Organizational grime refers to the physical distribution of pattern classes throughout software packages and namespaces. Empirical studies suggest that modular grime tends to increase as software designs evolve [9]. Evidence of class and organizational grime is inconclusive [9]. This research proposes a

ESEM'10, September 16-17, 2010, Bolzano-Bozen, Italy.

preliminary taxonomy of modular grime that goes beyond the original definitions by providing examples in proposed categories.

Modular grime builds when the classes of design pattern realizations grow new relationships not described by the pattern's RBML. A pattern's RBML is a precise description of the classifiers and associations that belong to said pattern. RBML was chosen to describe design patterns because it provided an intuitive UML-like format that can be used to detect and specify any pattern. We define criteria that determine when a new coupling involving a pattern class contributes to modular grime. Section 2.1 describes the criteria that define these couplings.

2.1 Coupling between classes

There exist many metrics and measures that distinguish between various dimensions of coupling between classes [4]. We use **strength**, **scope** and **direction** to classify modular grime.

2.1.1 Strength of coupling

Coupling can be classified on an ordinal scale according to strength [13]. Strength is determined by the difficulty of removing the coupling relationship. We use persistent and temporary coupling because they are the most common forms in object oriented systems [13]. For example, two classes A and B have a persistent (strong) association when class A contains an attribute of type B. The classes have a temporary (weak) association when class A contains a method with a parameter, a return value, or a local variable of type B. The relative strength of the coupling relationship is approximated by the amount of effort required to refactor the relationship. Persistent relationships are considered *strong* because they are likely to persist throughout the lifetime of the design pattern realization, while temporary relationships are considered *weak* because of their provisional nature. The sets **Persistent** = {class_attribute} and **Temporary** = {method_local_variable, method_return_value, method_formal_parameter} define ordinal sets for permanent and temporary coupling types respectively. Each set contains (in increasing level of refactoring complexity) the types of coupling considered. Currently, we do not have enough ordinal categories to justify using a five point Likert scale, however as new members of the Persistent or Temporary sets are identified we can easily accommodate such changes. These sets can be augmented with other less common forms of coupling in object oriented designs (i.e., sharing of global variables, data flow couplings, etc.). Extensive research in identifying different coupling types has been performed by [4].

2.1.2 Scope of coupling

Scope demarcates the boundary of a coupling relationship and can be internal or external. A class belonging to a design pattern develops a relationship with external scope if another class (not in the design pattern) is coupled with the former. A relationship has internal scope if the coupling involves two classes belonging to the same realization of a 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 $r_{ci, cj}$

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

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

2.1.3 Direction of coupling

We use afferent (**Ca**) and efferent (**Ce**) coupling to refer to the direction of a coupling relationship [12]. The afferent coupling count (number of in-bound relationships) of a set of classes increases when an external class c_{ext} references a member of the set ***C(P)***. A reference can be a new attribute, method parameter, return value, or local variable. Similarly, the efferent coupling count (number of out-bound relationships) of a set of classes increases when any class ci $\in C(P)$ references an external class c_{ext}.

2.2 Grime categories

The strength, scope, and direction of coupling relationships are used as the primary coupling factors that influence the construction of a modular grime taxonomy. As realizations of design patterns evolve, not all new relationships developed by classes that belong to the design pattern are considered grime. In section 2.2.1 we discuss relationships that do not contribute to grime build-up of design patterns. Section 2.2.2 characterizes relationships that contribute to grime build-up and provides definitions of modular grime classifications.

2.2.1 Non-grime coupling

As design pattern realizations evolve, external **Ca** counts due to usage relationships grow (as expected). Additionally, the appearance of new internal coupling relationships as a result of allowed RBML extensions are expected if the design pattern evolves as intended by the designer. New relationships that evolve as a result of such circumstances are not categorized as grime. For example, design patterns are extended through generalization and specialization of pattern classes. Classes that extend patterns through conformant RBML inheritance relationships are not considered grime. In addition to allowed inheritance relationships, unidirectional internal coupling increases are also expected if a pattern evolves via intended extensibility mechanisms. For example, the Visitor pattern creates such a relationship between the client side hierarchy *visitor* methods and the server side hierarchy *accept* methods.

2.2.2 Grime

Coupling relationships that violate the pattern's RBML contribute to grime build-up. Violations depend on the design pattern realization and the RBML that characterizes such realizations. If the RBML of a design pattern is strict, then the number of initial realizations found in a design will be smaller and the evolution of the pattern will be constrained. Alternatively, if the RBML of a design pattern is too lenient, then any set of coupled classes can be made to match the pattern's description, yielding too many false positives. The evolution of the pattern would be largely unrestricted.

Couplings that cause modular grime are classified according to our definitions of strength, scope and direction. The strength of coupling is an important dimension in the taxonomy because it helps determine the difficulty of grime removal (via refactoring) by developers. Grime resulting from the accumulation of strong coupling relationships requires additional effort to refactor. For

example, persistent coupling relationships are more difficult to remove than temporary coupling relationships. Coupling direction indicates the source of the grime. An increase in non-conformant **Ce** counts of a pattern implies that pattern classes must be refactored to remove grime build-up. An increase in external **Ca** counts to pattern classes also indicates possible grime build-up if the relationships are made to concrete classes. Usage relationships are intended to be made with the abstract classes of a pattern.

Using these criteria, we classify modular grime into six disjoint groups listed in sections 2.2.2.1 through 2.2.2.6. Figure 1 displays the taxonomy.

2.2.2.1 Persistent internal grime (PIG)

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

2.2.2.2 Temporary internal grime (TIG)

This set contains invalid temporary relationships involving two pattern classes $\in$ ***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** $\in$ **Temporary** and $|\boldsymbol{R(P)}|$ increases when **r** is invalid.

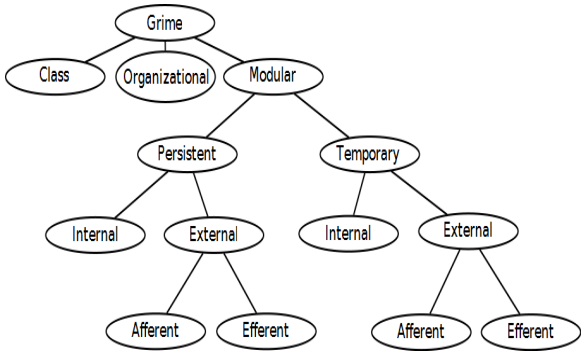


Figure 1. A Taxonomy of Object Oriented Design Pattern Grime

2.2.2.3 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** $\in$ **Persistent** and **Ce** increases when **r** is invalid.

2.2.2.4 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** $\in$ **Temporary** and **Ce** increases when **r** is invalid.

2.2.2.5 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 $\notin$ ***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. PEAG is observed when **r** $\in$ **Persistent** and **Ca** increases when **r** is invalid.

2.2.2.6 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 $\notin$ ***C(P)***. TEAG is observed when **r** $\in$ **Temporary** and **Ca** increases when **r** is invalid.

3. Taxonomy examples

In this section we provide illustrative examples of how grime build-up on creational, structural and behavioural design patterns is classified using the proposed taxonomy. Example pattern realizations are depicted with representative invalid couplings. Invalid couplings are labelled as "violations" that develop over time. Classification of invalid relationships is driven by the criteria defined in section 2.1.

3.1 PIG and TIG of the Observer behavioral pattern

PIG occurs when an invalid persistent association develops between internal pattern classes $\in$ ***C(P)***. Figure 2 depicts an example where the relationship $\mathbf{r}_{\text{ConcreteObservable,ConcreteObserver}}$ violates the pattern's RBML. Under governance of valid RBML, the concrete classes are indirectly coupled via inheritance through the parent class relationship, and by ConcreteObserver classes with unidirectional references to ConcreteObservable classes. The association $\mathbf{r}_{\text{ConcreteObservable,ConcreteObserver}}$ is an example of a violation and how the pattern is not meant to be extended.

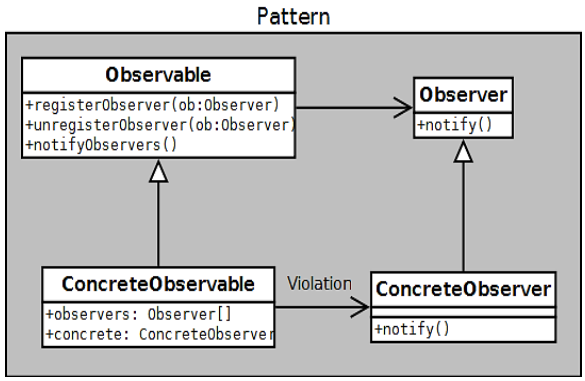


Figure 2. Observer Pattern Realization

Similarly, if $\mathbf{r}_{\text{ConcreteObservable,ConcreteObserver}} \in$ Temporary (i.e., a use dependency) and is an invalid relationship, then **r** belongs to the TIG set.

3.2 PEEG and TEEG of the Singleton creational pattern

Modular grime build-up in the Singleton pattern is only possible by means of external relationships. The RBML description allows one class or one inheritance hierarchy as valid realizations, thus eliminating the possibility of internal relationships. Figure 3 depicts an occurrence of PEEG. The Singleton class develops an invalid external, persistent relationship $\mathbf{r}_{\text{Singleton,Other\_Class}}$. The

UML diagram shows that **r** is invalid because it violates its RBML. The **Ce** of the pattern increases with the addition of **r**. We classify **r** as a member of the PEEG set.

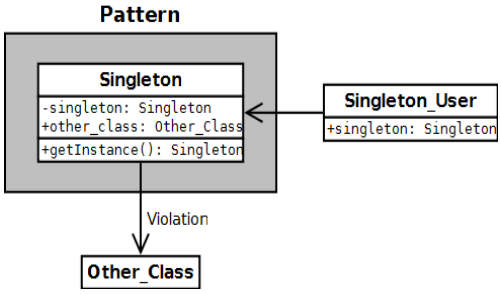


Figure 3. Singleton Pattern Realization

Similarly, if $\mathbf{r}_{Singleton,Other\_Class} \in$ Temporary (i.e. a use dependency) and is an invalid relationship, then **r** belongs to the TEEG set. The relationship **r** shares the same characteristics as if it belonged to the PEEG set with one major distinction; the strength of coupling.

3.3 PEAG and TEAG of the Adapter structural pattern

Figure 4 depicts an example of PEAG. The relationship $\mathbf{r}_{Client,ConcreteAdapter}$ in the realization of the Adapter pattern is an invalid external relationship not supported by the pattern's RBML. The figure indicates that **r** ∈ Persistent, and causes the **Ca** of the pattern to increase. Thus, **r** ∈ PEAG.

Similarly, a change in coupling strength differentiates PEAG from TEAG. If $\mathbf{r}_{Client,ConcreteAdapter}$ is modified so that **r** ∈ Temporary, then **r** ∈ TEAG if it is also invalid.

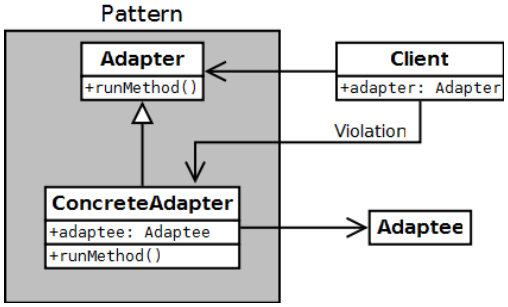


Figure 4. Adapter Pattern Realization

4. Pilot study

The purpose of the pilot study is to validate and refine each grime category in the proposed taxonomy. A case study was conducted on one open source software system to examine the evolution of modular grime involving design pattern classes. We test the following hypotheses to determine if grime, as determined by the proposed taxonomy, does occur in software design patterns.

$\mathbf{H_{1,0}}$: There is inconsequential PEEG buildup over the evolution of a software design pattern.

$\mathbf{H_{2,0}}$: There is inconsequential TEEG buildup over the evolution of a software design pattern.

$\mathbf{H_{3,0}}$: There is inconsequential PEAG buildup over the evolution of a software design pattern.

$\mathbf{H_{4,0}}$: There is inconsequential TEAG buildup over the evolution of a software design pattern.

In section 4.1 we describe the case study methodology. Section 4.2 displays observed raw results. Results are analyzed and discussed in section 5.

4.1 Methodology

4.1.1 Software studied

Vuze (formally Azuereus) [21] is a peer-to-peer file sharing client that uses the bittorrent protocol [16]. Vuze is written in Java, and is distributed under the Open Source license. The most recent release has more than 2 million downloads and it is considered a successful project. We study eight versions of the software spanning thirty eight months of development. Table 1 displays the basic statistics of each release.

Table 1.Vuze versions and release dates

Version	Date	LOC	# of Classes
2.5.0.4	01/2007	339,736	3,451
3.0.1.6	06/2007	349,946	3,518
3.0.5.2	05/2008	396,754	3,809
4.0.0.0	10/2008	484,969	4,354
4.1.0.0	01/2009	507,050	4,511
4.2.0.0	03/2009	519,396	4,608
4.2.0.8	09/2009	540,618	4,771
4.3.1.2	02/2010	547,203	4,682

4.1.2 Tools used

The software was analyzed under the Eclipse [19] Integrated Development Environment (IDE) using the Browse-by-Query (BBQ) [17] plug-in. BBQ creates an object oriented database of Java constructs from byte-code and allows users to query the resulting database for various metrics. All grime metrics are gathered with these tools.

We used three basic BBQ query statements to gather data about class attributes (persistent efferent coupling), class method return types, and method parameter types (temporary efferent coupling). In BBQ, executable queries are specified with English like sentences that allow for a more user friendly and accessible method for specifying arbitrarily complex expressions. To obtain all unique types of attributes in any class **C,** we use the query "unique types of attributes in class **C"**. All method return types are obtained using the query "unique types of methods in class **C**". All method parameter types are obtained with the query "unique types of arguments of methods in class **C**". We use the union operation with the last two queries to obtain the total temporary efferent coupling count of a class. The queries used to obtain afferent coupling counts are slightly different. To obtain all of the classes with an attribute of type **C** we use the query, "unique classes containing fields with type of class **C**". All classes with a method parameter of type C are obtained with the query "unique classes containing methods of arguments matching class

C". The set of classes with a method return type of class C is obtained with the query, "unique classes containing methods matching class **C**". We use the union operation with the last two queries to obtain the total temporary afferent coupling of a class.

To gather data about a set of pattern classes, we simply union the corresponding query type for each class in the pattern. For example, to find PEEG for the classes **C1** and **C2**, we use the query: "count((unique types of attributes in class **C1**) union (unique types of attributes in class **C2**))".

4.1.3 Data collection

Data was gathered from 8 versions of Vuze spanning 38 months of development. All design patterns were found with the aid of various tools and were manually validated. We use PatternFinder [18] to provide initial guidance for the location of potential design patterns. We then manually examine pattern structure and naming semantics to find patterns that were clearly intended by software designers. We check for RBML conformance to verify each pattern realization. There are currently no automated tools available that check for structural RBML conformance. Efforts to automate RBML conformance are still being developed [5]. The Java programming language provides several built-in constructs that allow a class to be considered part of a design pattern. For example, you can decorate a class to be observable. While these can be considered design patterns, they are out of the scope of this research.

We collected all coupling count metrics for each pattern realization under study. The taxonomy definitions from section 2 provide the basic information necessary to construct the BBQ queries. We count all couplings that appear during the evolution of each realization of every pattern. BBQ did not allow us to differentiate between internal and external scope of new couplings. Therefore, any changes in PIG are reflected in PEAG and PEEG, while changes in TIG will be reflected in TEAG and TEEG. We have identified more powerful tools [20] to help with the generation of refined queries beyond this seminal pilot study. For each pattern **P,** we used the following surrogate definitions to generate the data collection:

PEEG: Count of unique types of attributes of class **C** where **C** ∈ **C(P)**.

TEEG: Count of unique types of return values and parameters of methods in class **C** where **C** ∈ **C(P)**.

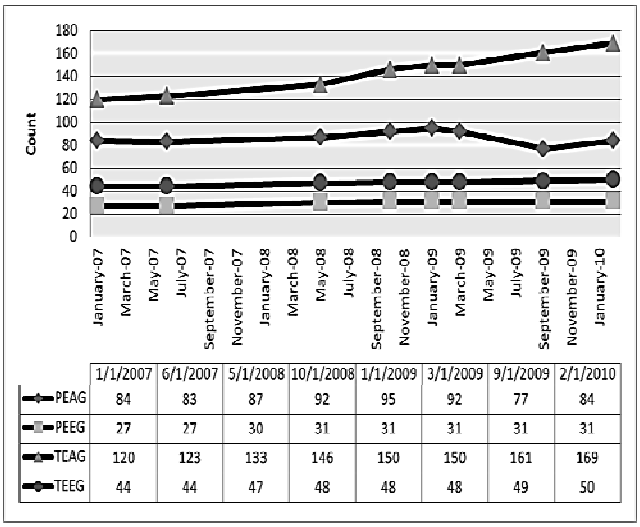
PEAG: Count of unique classes that have an attribute of type **C** where **C** ∈ **C(P)**.

TEAG: Count of unique classes that have a method return value or method parameter of type **C** where **C** ∈ **C(P)**.

We gathered data from 9 design pattern realizations. Each pattern realization exists in each of the 8 releases studied, and is RBML compliant in every version of the system. We study 3 Singleton, 3 Observer, and 3 Factory pattern realizations.

4.2 Results

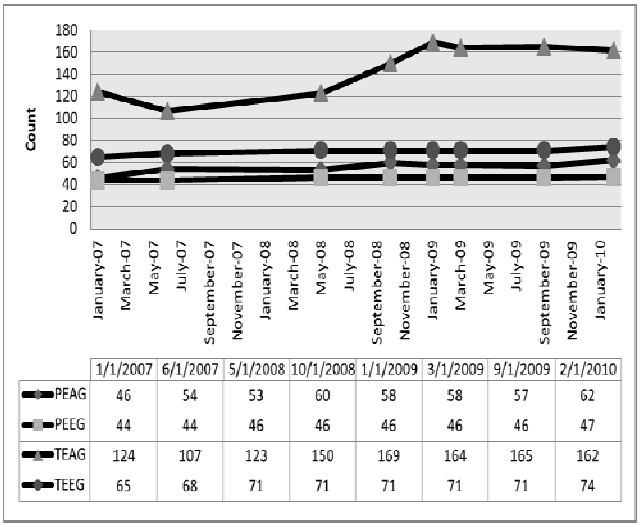
The modular grime counts of the Singleton pattern realizations are shown in Figure 5. We observe an abated, yet steady increase in all grime counts with the exception of PEAG. The latter begins to decrease after the January '09 release and continues to decrease until the January '10 release where a slight increase is observed. A refactoring event can cause this decrease, and we discuss this further in section 5.2. No other categories experience a similar decline.



	1/1/2007	6/1/2007	5/1/2008	10/1/2008	1/1/2009	3/1/2009	9/1/2009	2/1/2010
PEAG	84	83	87	92	95	92	77	84
PEEG	27	27	30	31	31	31	31	31
TEAG	120	123	133	146	150	150	161	169
TEEG	44	44	47	48	48	48	49	50

Figure 5. Singleton results

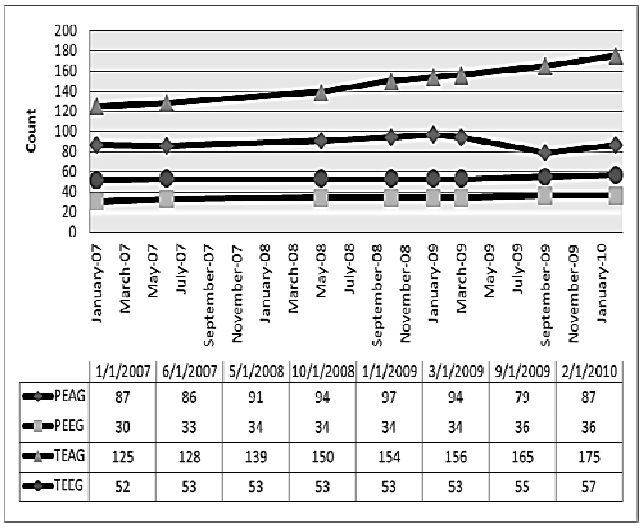
Results of the Factory pattern are shown in Figure 6. The metric counts for each grime category indicate slight increases over the three years of revisions. A notable exception is the marked growth observed in TEAG counts between the June '07 and January '09 releases. Grime counts for PEAG do not display similar trends to those observed in the Singleton pattern realizations.



	1/1/2007	6/1/2007	5/1/2008	10/1/2008	1/1/2009	3/1/2009	9/1/2009	2/1/2010
PEAG	46	54	53	60	58	58	57	62
PEEG	44	44	46	46	46	46	46	47
TEAG	124	107	123	150	169	164	165	162
TEEG	65	68	71	71	71	71	71	74

Figure 6. Factory results

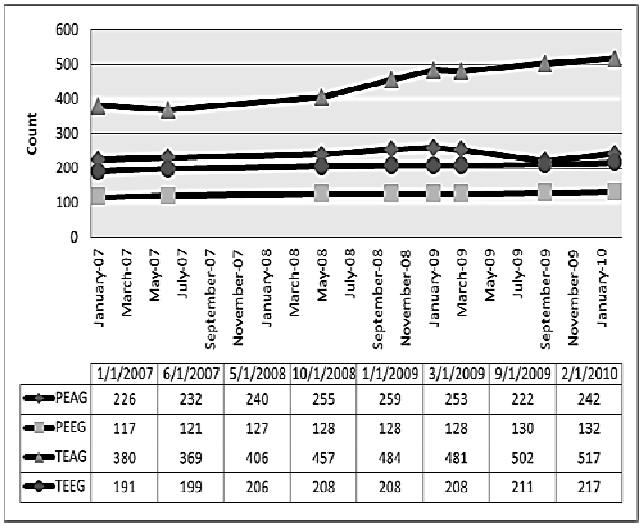
Results for all 3 realizations of the Observer pattern are shown in Figure 7. Results for each grime category are analogous to those observed in the Singleton pattern. There is a steady increase in coupling counts for three grime categories, with the exception of PEAG counts. PEAG begins to decrease after the January '09 release and continues to decrease until the January '10 release before a slight increase is observed.



	1/1/2007	6/1/2007	5/1/2008	10/1/2008	1/1/2009	3/1/2009	9/1/2009	2/1/2010
PEAG	87	86	91	94	97	94	79	87
PEEG	30	33	34	34	34	34	36	36
TEAG	125	128	139	150	154	156	165	175
TEEG	52	53	53	53	53	53	55	57

Figure 7. Observer results

In Figure 8 we display the aggregate totals of each grime category for all design pattern realizations. As reflected in the earlier results for each individual pattern, TEAG shows the greatest change from the first to last release with a net increase of 137. TEEG increases by 26, PEAG increases by 16, and PEEG increases by 15. Together these results support earlier findings by Izurieta and Bieman [9] that modular grime does occur.



	1/1/2007	6/1/2007	5/1/2008	10/1/2008	1/1/2009	3/1/2009	9/1/2009	2/1/2010
PEAG	226	232	240	255	259	253	222	242
PEEG	117	121	127	128	128	128	130	132
TEAG	380	369	406	457	484	481	502	517
TEEG	191	199	206	208	208	208	211	217

Figure 8. Results for all patterns

5. Analysis and discussion

In section 5.1 we do a non-parametric statistical analysis to determine the significance of grime buildup for each category of the proposed taxonomy. Section 5.2 discusses trends reflected in the study results and their impact on the taxonomy.

5.1 Statistical analysis

We use Wilcoxon's Signed-Rank test [15] to evaluate our hypotheses. The Signed-Rank test is a non-parametric test that determines whether the differences between measurements on sequential releases are significantly positive. The limited sample size of the data renders alternative parametric tests less reliable. However, while the Singed-Rank test can help determine if changes between revisions are statistically significant, it has no power to determine the magnitudes of these changes. All measured differences are converted to ranks at the beginning of the test and analysis is then performed on the ranks. Any large differences in grime counts will only be reflected in the test as a difference in, at most, several ranks. P-values in the Signed-Rank test represent the probability that changes between releases are zero. A low p-value implies it is very unlikely that increases in grime counts are due to random chance. To determine if grime buildup is statistically significant, the p-values for all of the tests are displayed in Table 2.

The p-values for PEEG are not significant at the 0.05 level except in the general case, where all results are aggregated regardless of pattern type. There are several cases in the data where the change in PEEG between releases is zero. A zero cannot be used in the Signed-Rank test because the data (difference between releases) is separated into groups by sign. Statistical analysis is done on the ranks of the values in each group and zeros must be discarded because they belong to neither group. This decreases the sample size and lessens the effectiveness of the test for each pattern. The significance ($p < .05$) obtained in the general case for all patterns can be explained by the fact that PEEG rarely decreases and usually increases regardless of pattern. The significant p-value suggests there is sufficient evidence to reject $\mathbf{H_{1,0}}$ in the general case, but not in the case of individual design patterns.

TEEG tests reveal low p-values for the Singleton pattern realizations as well as for the combination of all pattern realizations. The significance level of the p-value for the general case of all patterns, as well as the low p-values ($p < .1$) for all individual pattern realizations, suggests there is sufficient evidence to reject $\mathbf{H_{2,0}}$.

P-values for PEAG are not significant. Large decreases in grime counts for the Singleton and Observer pattern realizations between March and September of 2009 cause the results to be insignificant. These results suggest that there is no evidence to reject $\mathbf{H_{3,0}}$.

Table 2. P-values for the Signed-Rank test, significant values (<.05) in bold.

	PEEG	TEEG	PEAG	TEAG
Singleton	.186	**.044**	.306	**.018**
Factory	.186	.074	.200	.290
Observer	.091	.087	.335	**.011**
All	**.029**	**.030**	.250	**.039**

Test results for TEAG reveal that this type of grime is not significant in the Factory pattern realizations only. The significance of the Singleton, Observer, and All p-values also suggests we have sufficient evidence to reject $\mathbf{H_{4,0}}$.

Correlation analysis was performed on the results shown in Figure 8 to confirm an apparent relationship observed between PEEG and TEEG. The value calculated for the Spearman rank correlation coefficient is 1. The value of the coefficient is attributed to the grime counts for both PEEG and TEEG; which are both monotonically increasing over the release history. A

parametric alternative, the Pearson correlation coefficient, is calculated at 0.989. These coefficients provide evidence of a positive correlation between the two categories. Although this does not imply causality, the result confirms that these two types of grime *move* together.

5.2 Discussion

The purpose of the proposed modular grime taxonomy is to suggest a possible organization of couplings that have negative effects on the evolution of design pattern realizations. The purpose of the pilot study is to empirically show the extent to which each grime classification contributes to the decay of software pattern realizations. The results present some interesting trends.

PEEG and TEEG show similar trends in every dataset. Both categories show very little variation. These findings indicate that PEEG and TEEG may not be independent and have a positive correlation as reported in 5.1. This would suggest that coupling strength as defined in the taxonomy plays a minor role as a modular grime classifier when applied to external efferent coupling. This conclusion is different for external afferent coupling; where PEAG and TEAG appear to be distinct. Both show varying degrees of variability and there are instances (September 2009) in the Singleton and Observer patterns realizations where TEAG increases while PEAG shows a marked decrease. This supports the notion that PEAG and TEAG are independent. TEAG/PEAG and TEEG/PEEG differ only by coupling strength, yet the data suggests the former pair is independent while the latter is not. Pattern usage is a possible explanation for this result. Changes in usage will not be reflected in grime counts for TEEG and PEEG because usage must originate in non-pattern classes, thus affecting afferent coupling only. However, a refactoring event to reduce strength of coupling between patterns and classes using said patterns could explain the difference between TEAG and PEAG. Developers aware of the dangers of strong coupling might refactor classes that use patterns to reduce persistent couplings to temporary couplings. The result would be a simultaneous increase in TEAG and decrease in PEAG. Excluding the Factory pattern realization, this possibility is best reflected in the changes observed in TEAG and PEAG between March and September of 2009.

Results observed in the Singleton and Observer patterns are similar. Metrics for each grime category show similar trends over the evolution of the software in both patterns. The possible refactoring event discussed earlier is reflected in both. A possible explanation is that some of the pattern realizations are coupled [4]. Coupled patterns share common classes. Changes to shared classes are reflected in individual grime counts for each pattern realization involved in the coupling. Further investigation of the Singleton and Observer pattern realizations used in the pilot study shows that there is one example of pattern coupling. One of the Singleton pattern realizations is embedded within an Observer pattern realization.

Factory pattern grime counts show little similarity to those observed in other patterns. In general, the results provide no evidence to suggest that different types of patterns (creational, observational, or behavioral) develop grime in a different manner. There appears to be no discernible relationship between pattern type and grime buildup.

The statistical results reveal that, in general, PEEG, TEEG, and TEAG tend to increase throughout the evolution of design patterns while PEAG does not. The division between grime categories supports the use of coupling direction as the most relevant criteria for classification. Pattern usage may have an effect on the results for TEAG, but PEEG and TEEG cannot increase as the result of usage. It is also apparent that different design patterns may experience changes in grime differently. Further research is necessary to determine if this is true.

6. Threats to validity

We assess construct, content, internal and external validity of the case study.

Construct validity refers to the use of meaningful metrics and measures. The measures used for releases and grime coupling counts must actually quantify the notion of releases and the various grime categories. We use BBQ to build queries that when executed, collect grime counts. The queries serve as surrogates to capture the metrics whose formal definitions are given in section 2. BBQ does not have the ability to differentiate between internal and external grime counts, and this limitation forced us to combine the two. This threatens construct validity because changes in TIG will have an effect on measures for TEEG and TEAG, while changes in PIG will affect PEAG and PEEG. Though this may cause concern over the validity of the results, it should be noted that we were unable to manually find an example of internal grime during the data gathering phase of the research. Additionally, the lack of automated RBML tools limits our ability to differentiate between some types of compliant relationships and grime buildup. Increases in usage of the design patterns studied can inflate some modular grime results.

To have content validity, the measures must completely represent the notions of grime coupling counts. The definitions proposed in this case 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 4 independent variables —TEEG, PEEG, TEAG, and PEAG, and one dependent variable, grime count. There is a rationale to consider design pattern types (creational, structural, or behavioral) or Open Source software versus commercial software as a possible dependency, where the type of pattern or system accumulates different types of grime at different rates. However, we did not have enough data to support a causal relationship.

External validity indicates that the study results can generalize to other systems. Clearly, the size limitations of the pilot study have an effect on external validity. Only one Open Source system was studied and the number of distinct pattern types and realizations of those types were small. It is not possible to generalize from these preliminary results and thus we cannot assert whether patterns in other systems decay and buildup grime in a similar manner. Additional data needs to be gathered from commercial and Open Source systems to increase external validity.

7. Conclusion and further research

A modular grime taxonomy is presented that uses three basic underlying criteria as classification factors: strength, scope, and direction. Strength helps us identify the relative difficulty of

refactoring invalid coupling relationships. Scope helps us determine if the source of grime build-up comes from within pattern classes or from changes made to the surrounding design. Finally, direction defines the source of the grime. Grime originating from external classes is harder to remove because of the heightened responsibility of the pattern. Grime originating from within the pattern causes the testability of the pattern to increase because the dependencies of said pattern are higher.

The pilot study confirms earlier research that modular grime does occur. The results observed in the evolution of the Vuze software system confirm that TEEG, TEAG, and PEEG show increases in grime counts. Analysis of the data shows no apparent relationship between design pattern type and modular grime; although a more formal study is needed to confirm this conclusion.

This pilot study is another data point in generating a body of evidence to continue our understanding of how design patterns decay. Further studies will explore the prevalence of each grime category in Open Source software and serve as validation of proposed categories.

Additional research is also planned to compare grime build up against the total coupling of objects in a software system. This ratio will help us determine if grime grows at a different rate than the coupling of all objects, including objects not associated with design pattern realizations. Future studies will also investigate the appearance of grime in coupled design patterns. A thorough statistical trend analysis is also necessary in future studies to help predict the build up of grime over time. A significant number of coupling measures have been proposed that have not been taken into consideration in this seminal work. This threatens the construct validity of the taxonomy; however we expect to expand the taxonomy to accommodate a higher order space.

8. References

[1] Arisholm, E. and Sjoberg, D.I.K. 2000. Towards a Framework for Empirical Assessment of Changeability Decay. J. Syst. Software. 53, 1 (July. 2000), 3-14.

[2] Basili, V.R., Briand, L.C. and Melo W.L. 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE. Trans. Software. Eng. 22, 10 (October 1996), 751-761.

[3] Bieman, J. and Wang, H. 2004 Evaluating the Strength and Impact of Design Pattern Coupling. Working paper.

[4] Briand, L.C., Daly, J.W. and Wust J.K. 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems IEEE. Trans. Software. Eng. 25, 1 (January 1999), 91-121.

[5] Dae-Kyoo, K. and Shen W. 2008. Evaluating Pattern Conformance of UML Models: a Divide and Conquer Approach and Case Studies. Software Quality Control. 16, 3 (September 2008), 329-359.

[6] Eick, S.G. et al. 2001. Does Code Decay? Assessing the Evidence from Change Management.Data. IEEE. Trans. Software. Eng. 27, 1 (January 2001), 1-12.

[7] France, R.B., Kim D.K., Song, E. and Ghosh S. 2004. A UML-Based Pattern Specification. Technique. IEEE. Trans. Software. Eng. 30, 3 (March 2004), 193-206.

[8] Gueheneuc, Y-G. and Antoniol, G. 2008. DeMIMA: A Multi-Layered Framework for Design Pattern Identification. IEEE. Trans. Software. Eng. 34, 5, (September 2008) 667-684.

[9] Izurieta, C. and Bieman, J.M. 2008. Testing Consequences of Grime Buildup in Object Oriented Design Patterns. In First ACM-IEEE International Conference on Software Testing, Verification and Validation (Lillehamer, Norway, April 9-11, 2008). ICST '08, 449-451.

[10] Izurieta, C. and Bieman, J.M. 2007. How Software Designs Decay: A Pilot Study of Pattern Evolution. In First ACM-IEEE Symposium on Empirical Software Engineering and Measurement (Madrid, Spain, Sept. 20 – 21, 2007).ESEM '07, 171-179.

[11] Izurieta, C. 2009.Decay and Grime Buildup in Evolving Object Oriented Design Patterns. Ph.D. Dissertation, Colorado State University.

[12] Martin, R. 1994. OO Design Quality Metrics-An Analysis of Dependencies. Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA ' 94.

[13] Myers, G.J. 1978. Composite/Structural Design. Van Nostrand Reinhold, New York.

[14] Ohlsson, M., Mayrhauser, A., McGuire, B. & Wohlin, C. 1999. Code Decay Analysis of Legacy Software through Successive Releases. In Proceedings of the IEEE Aerospace Conf (Aspen, CO., March 6-13, 1999). 69-81.

[15] Wilcoxon, F. 1945. Individual Comparisons by Ranking Methods. Biometrics, 1, 80-83.

[16] Bittorrent Protocol. http://www.bittorrent.org/beps/bep_0003.html

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

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

[19] Eclipse Integrated Development Environment. http://www.eclipse.org/

[20] SemmleCode. http://semmle.com/semmlecode

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