Computer Science Technical Report, 2006.

# Design Pattern Coupling, Change Proneness, and Change Coupling: A Pilot Study

James M. Bieman and Huixia Wang
Software Assurance Laboratory
Computer Science Department
Colorado State University
Fort Collins, CO 80523.
Email: bieman@cs.colostate.edu, wanghu@cs.colostate.edu

## Abstract

*A design pattern realization consists of a cluster of classes that work together to solve a particular problem using a well known, named solution. Developers may build systems out of several pattern realizations, and these pattern realizations may be interconnected, or, in other words, coupled. Coupled pattern realizations may represent a reasonable solution to software design problems, however the coupling can introduce dependencies that increase fault-proneness and lower adaptability. We identify mechanisms that can couple pattern realizations, and evaluate the relative tightness of the connections. An examination of pattern coupling in five systems provides initial evidence that pattern coupling is common. In addition, we find initial evidence that classes in pattern realizations that are coupled via associations are (1) more change prone and (2) exhibit higher change coupling — classes that are modified together in response to one required change — than those in pattern realizations that are coupled by other mechanisms*

**Keywords:** Patterns, coupling, change proneness, object-oriented design methods, quality analysis and evaluation, measurement, maintainability, enhancement, extensibility, maintenance measurement.

## 1 Introduction

A principle objective of software design is to determine a "good" arrangement of program entities, whether those entities are procedures, functions, or classes. A long standing design goal is to maximize program entity strength or cohesion while minimizing program entity coupling [31].

The notion of *coupling* refers to the relationships between program entities. Designers aim for low coupling to minimize dependencies between entities. With low coupling between program units, a change in one unit is less likely to affect another unit. Also, low coupling should ease debugging, since, with low coupling, a fault in one entity is less likely to propagate to other entities. In several case studies, Briand et al. found a significant relationship between coupling and fault proneness of program classes [7, 10].

Myers' definition of seven ordered categories of coupling tightness between modules — from (1) no direct coupling (best case) to (7) content coupling (worst case) — is the basis for most coupling measures for procedural software [31]. These tightness categories represent an ordinal scale that indicates the relative interdependence of the coupled modules.

Coupling measures for object-oriented software use classes as the program unit or entity. Briand, Daly, and Wust survey and evaluate a comprehensive suite of coupling measures for object-oriented systems [8]: coupling between objects (CBO) and response set for a class (RFC) [13], message passing coupling (MPC) and data abstraction coupling (DAC) [25], efferent copuling (Ce) and afferent coupling (Ca) [27], coupling factor (COF) [1], information flow (ICP) [24], 11 counts of various interactions [9]. Other coupling measures include counts of the number of associations between classes and their peers (NAS) [18] and couplings between modules (CBM) where a module is a Java package [26]. All of these coupling measures indicate the strength of coupling between classes. Rather than treat an individual class as a design elements, our work views a cluster of classes as one design element.

Design patterns that represent known solutions to common software design problems are now commonly employed in commercial and open source development [5]. Software design activities are moving towards a model-driven development process [6] where developers will build systems by composing pattern realizations together, rather

than by connecting individual classes. Thus, systems become collections of interacting pattern realizations and individual classes that are not part of a pattern.

With design patterns becoming a primary design entity, the assessment of design quality should take into account the attributes of pattern realizations. In this paper, we examine the notion of pattern coupling [28], which is one possible pattern attribute.

Figure 1 shows a UML class model, from a real system, of two coupled pattern realizations from Gamma et al [16]. A realization of a Factory Method pattern is coupled to a realization of a Strategy Pattern. In this example, interface Product and class ProductA play roles in both pattern realizations.

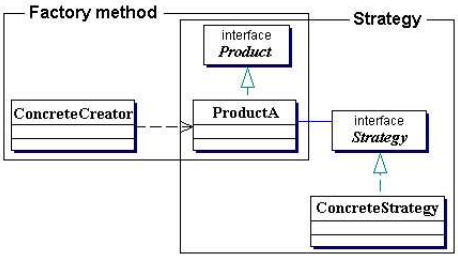



**Figure 1. Intersection Coupling between a Factory Method pattern realization and Strategy pattern realization in proprietary System B.**

Pattern coupling can potentially affect the adaptability of a system. A system of pattern realizations with more strong connections between individual pattern realizations may be more difficult to understand and thus more difficult to maintain than a system of weakly connected pattern realizations.

Adaptability can be measured in terms of the actual labor required to modify a system as it evolves, perhaps in terms of person-hours. Unfortunately, such information can be very difficult to obtain. Thus, we use change proneness as a surrogate for adaptablity. Change proneness can be measured by counting the number of changes to particular system elements as they evolve. An element that requires relatively more changes is, in general, more difficult to adapt. Of particular concern is that due to coupling, changes to one element may in turn force changes to another coupled element. Such *change coupling* of elements [3] indicates lower adaptability.

The coupling of pattern realizations may, in fact, have positive benefits. That is, coupled pattern realizations may help to make a coherent design. In the case of the coupled Factory Method and Strategy pattern realizations in Figure 1, we can argue that it is natural to use a factory to create concrete products. The connection between the ConcreteCreator and ProductA is transient and there is minimal likelihood of interference between the Factory Method pattern realization and the Strategy pattern realization, and little or no chance of the Strategy pattern realization interfearing with the Factory Method.

As in the coupling of procedures and functions, some couplings are good and necessary — data coupling in the Myers hierarchy — and others lead to trouble — common coupling and content coupling [31]. Even with the preferred forms of coupling, too many couplings can make a system unmanageable. Thus, we are concerned with both the mechanism for coupling and the number of couplings.

Our goal in this work is to understand the mechanisms that can couple design pattern realizations, and evaluate the relative impact of the different mechanisms on change proneness and change coupling. Strong coupling indicates greater interdependencies between pattern realizations, while weak coupling indicates greater independence. We select changeproneness and change copuling as evaluation criteria, since ease of modification is a primary reason for using design patterns [16].

First we identify and classify methods of coupling design pattern realizations, and propose an ordering of pattern coupling mechanisms in terms of coupling strength. We examine coupled pattern realizations in available commercial and open-source software to further understand the reasons for pattern couplings. The specific coupling mechanisms in these systems provide insights into the effect of the coupling on hypothetical changes. To further understand the relative strength of pattern coupling mechanisms we observe the change proneness of classes that play roles in coupled pattern realizations in four case studies of evolving systems. Change proneness is an indicator of the strength of a connection — tightly coupled elements are more likely to require modifications due to ripple effects.

## 2 Pattern Coupling Mechanisms and Coupling Strength

Since a design pattern realization consists of a cluster of one or more classes, pattern realizations can be linked in the same manner as individual classes — via associations, inheritance, use dependencies, and interactions with global data. Pattern realizations can also be linked through classes that play roles in several pattern realizations.

A classification of the coupling mechanisms can be based on a set of criteria including the following:

- The number of connections.
- The abstraction levels of the connections. Pattern realizations may be coupled via links that appear in class diagrams of detailed designs or conceptual models.

The links may represent interactions that occur in code, interaction diagrams, or in role models that serve as pattern specifications [15].

- The persistence of a connection between pattern realizations. A link may last for the lifetime of the pattern realizations involved or exists during a single method activation.

- Direction of the coupling.

We might expect coupling strength to increase with the number of links between pattern realizations, persistent links are likely to be stronger than transient links, and two-way coupling is likely to be stronger than one-way. The specific mechanisms used to link two patterns is a key factor in determining the strength of coupling.

Coupling has historically been evaluated using an ordinal scale, from weakest coupling to the strongest coupling [31]. We propose a similar ordinal scale for design pattern coupling, with the following preliminary coupling classifications:

0. No coupling. Two pattern realizations are not linked.

1. Transient (use dependency) coupling.

2. Persistent coupling via associations.

3. Persistent coupling via sharing of common objects via one of two categories:

   (a) Embedded. A parent pattern realization contains some realizations of the embedded pattern(s).

   (b) Intersection. Each pattern realization contains both the common objects that play roles in two or more pattern realizations, and some independent objects that play roles in only one pattern realization.

4. Sharing common superclasses.

5. Common coupling (also persistent). Two pattern realizations are common coupled when they communicate by sharing the same global data entity.

The proposed ordering of pattern coupling mechanisms represents a hypothesis, and needs validation. The rankings represent a suggested ordering, and should be treated as a classification or nominal measure until the ordering is validated.

# 3 Potential Impact of Pattern Coupling

Pattern coupling can potentially have a negative impact on on external quality factors. Both errors and changes in classes in one pattern realization can propagate to classes in the other pattern realization. We examine the impact of pattern coupling via the notion of *interference*, where behavior in one pattern realization interferes with that of another pattern realization. The analysis is based on examples of couplings found in the five systems studied. We look at each category of coupling mechanism in order.

## 3.1 Transient (Use Dependency) Coupling

Objects of classes that play roles in two design pattern realizations can be coupled via use dependencies. The relationship between the coupled pattern realizations is temporary. The pattern with import coupling has a use dependency on the pattern with export coupling. To reuse pattern realizations with import coupling, developer must import the pattern realizations or parts of the pattern realizations to which it is coupled. Changes to classes in pattern realizations with export coupling are likely to affect coupled pattern realizations with import coupling. In contrast, modifications to pattern realizations with import coupling should be less likely to affect the pattern realizations with export coupling. Because the use dependency connections are transient and are in effect only while a method is active, they represent looser coupling than connections implemented via associations, which persist between method invocations.

Fig. 2 shows an example of transient (use dependency) coupling between realizations of the Singleton pattern and the Strategy pattern in System A. In this example, class SingletonA has a method with an output parameter that is an object of type Context. Class SingletonA, which implements the Singleton pattern, imports class Context and ContextA, which play roles in a realization of the Strategy pattern. A change to class ContextA that either modifies the signature of the constructors that SingletonA invokes or adds code that conflicts with the implementation of class SingletonA, requires that class SingletonA be changed. In addition, reuse of SingletonA (Singleton pattern) requires that one import class Context and ContextA (Strategy pattern).

## 3.2 Persistent Coupling Via Associations

Associations that connect classes in different pattern realizations can be easily identified as associations in a UML class diagram of the software system. The relationship between the coupled pattern realizations is persistent — the connection continues between method activations. The coupling is explicit, and it is fairly easy to separate such coupled pattern realizations.

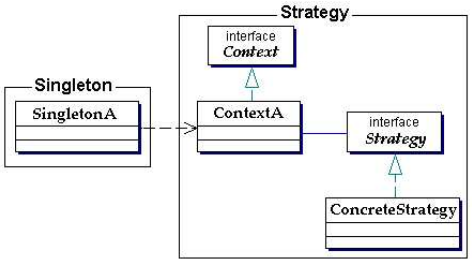



**Figure 2. Transient (use dependency) coupling between a Singleton pattern realization and Strategy pattern realization in System A.**

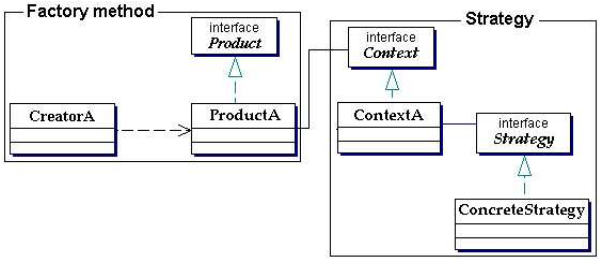



**Figure 3. Associations between realizations of the Factory Method and Strategy patterns in System B.**

A class in the pattern realization with import coupling has a reference to a class in the pattern realization with export coupling. So to reuse pattern realizations with import coupling, one must import the pattern realizations or some classes in the pattern realization(s) to which it is coupled. Modifications in the pattern realizations which have export coupling are likely to affect the pattern realizations with import coupling. However, modifications in the pattern realizations with import coupling are unlikely to affect the pattern realizations with export coupling.

Figure 3 shows an example of association coupling between realizations of the Factory Method and Strategy patterns from System B. In this example, class ProductA has an attribute whose type is Context. When ProductA object initializes an object of type Context, the constructor of ContextA will be invoked. In this case, ProductA (in the Factory Method pattern) imports class Context (in the Strategy pattern). So if the class Context or ContextA changes something on which the class ProductA relies, or adds something that raises a conflict with something in class ProductA, then class ProductA will need to be changed. Changes in ProductA will probably make it necessary to modify other classes in the factory method pattern realization. To reuse this realization of the Factory Method pattern, we must also import the strategy pattern realization to which it is coupled.

## 3.3 Persistent Coupling Via Sharing Common Objects

A class may play a role in more than one pattern realization. When all classes in one pattern realization also play roles in a second pattern realization, the two pattern realizations exhibit embedded coupling. Intersection coupling occurs when one or more classes in two pattern realizations, but not all classes in either pattern realization play a role in the other pattern realization.

### 3.3.1 Embedded Coupling

When there is embedded coupling, one pattern realization, the parent realization, contains all of the classes in the embedded pattern realization. Modifications in embedded pattern realizations can cause a series of changes in the parent pattern realization. Changes to classes in one pattern realization are more likely to propagate to the other pattern realization with embedded coupling than with association coupling. Implementations in embedded pattern realizations are shared with the parent pattern realization, while, in coupling implemented via associations, the only connection between the pattern realizations is through public interfaces — implementations are not shared.

Embedded coupling defines a hierarchy; the parent pattern realization contains the embedded pattern realizations. Thus, the whole structure can be referenced through the parent, it can be treated as a unit and easily reused later. Thus, we expect that the portion of a design that makes use of embedded coupling will be more reusable than the portion of a design that uses intersection coupling.

Fig. 4 shows a Factory Method pattern realization with an embedded Singleton pattern realization from System B. In this example, the whole structure is realization of a factory method pattern. Class ConcreteCreator plays a role in the Factory Method pattern realization and is also an realization of a Singleton pattern — there can only be one instance of ConcreteCreator. Modifications to methods that create objects of class ProductA in class ConcreteCreator will likely require, in turn, that new constructors be added to class ProductA. Changes, except for changes to ProductA constructor interfaces, are less likely to force modifications in the class ConcreteCreator.

### 3.3.2 Intersection Coupling

With intersection coupling, each coupled pattern realization consists of common classes and independent classes. Interactions between the coupled pattern realizations may be

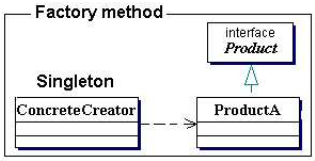



**Figure 4. Embedded coupling — a Singleton pattern realization inside a Factory method pattern realization in System B.**

complicated, and the effects of modifications to one pattern realization can ripple through the other pattern realization. Fig. 5 and Fig. 6 show two examples of intersection coupling between design pattern realizations in System B. There are two design pattern realizations in Fig. 5: realizations of the Factory Method and Strategy patterns. These two design realizations share two common classes: Product and ProductA. The Factory Method pattern realization consists of the two common classes and class ConcreteCreator. The strategy pattern realization consists of the two common classes and two independent classes: Strategy and ConcreteStrategy. In this example, class ConcreteCreator has a method which returns a parameter whose type is ProductA. ProductA also has an attribute whose type is ConcreteStrategy. To modify the constructor of ProductA, one may have to change ConcreteCreator, Strategy and ConcreteStrategy. Modifications in the ConcreteCreator class, for example, a change in the method that creates an object of ProductA, will require changes to the constructors of class ProductA and ConcreteStrategy. These changes will, in turn, require changes to ProductA and ConcreteCreator to keep them consistent.

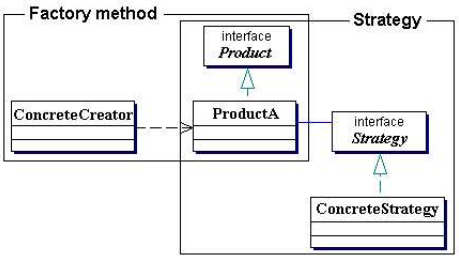



**Figure 5. Intersection coupling between Factory Method and Strategy pattern realizations in System B.**

Fig. 6 displays three coupled adapter pattern realizations: a1, a2 and a3. Pattern realization a1 and a2 share a common interface — AdapteeC(TargetB); a2 and a3 share a common interface — TargetA. In this example, adding new methods to the common interfaces will require adding new implementations to the adapter classes of both coupled pattern realizations. However, modifications to either of the adapter classes will be less likely to affect the other pattern realization. The reason is that both adapter classes implement the common interfaces separately and the modifications to either of the adapter classes will not affect the interface if there are only modifications in the body of the methods rather than the method name and signature.

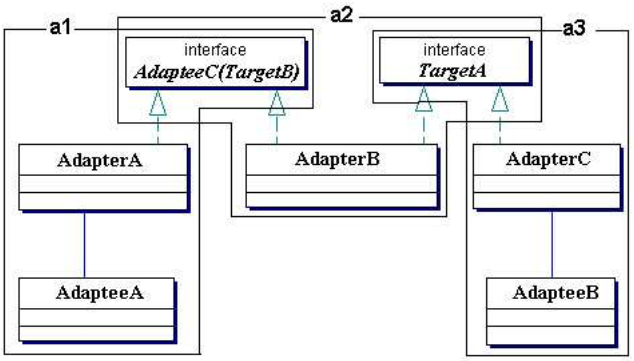



**Figure 6. Intersection coupling between 3 realizations of the Adaptor pattern in System B.**

### 3.4 Sharing Common Superclasses

The relationship between pattern realizations coupled by sharing common superclasses can be complex. Subclasses in the coupled pattern realizations may implement abstract methods, inherit the public/protected variables/methods, or override some of the public/protected methods in the superclass. The subclasses remain dependent on the implementation of the superclass, and a change in the superclass can affect the subclasses. A host of *fragile base class problems* can occur when method implementations in a base class are changed, even without affecting the interface of the superclass [30].

Clearly, modifications in the superclass will affect both of the coupled pattern realizations. However, changes in either of the coupled pattern realizations are unlikely to affect the other realization. Sharing common superclasses between design pattern realizations seems to represent simple and weak coupling. There is no direct connections between the coupled realizations and it will be easy to break apart the coupled pattern realizations. Changes in one pattern realization will not affect the other realization. However, coupling via global variables reference in the superclasses may effec-

tively hide common coupling from the sublasses, resulting in implicit common coupling.

Fig 7 shows an example of two realizations of the Factory Method pattern that are coupled by sharing a common superclass in System B. In this example, the Creator classes (CreatorA and CreatorB) and ConcreteProduct classes (ProductA and ProductB) in both pattern realizations have the same superclass (ClassA) and none of the subclasses have methods which override the methods in the superclass. There are public static methods in ClassA that access a single object. Class ProductA and ProductB inherit these static methods and invoke them in the implementations of some other methods. The single object referenced by the static methods acts as a global variable.

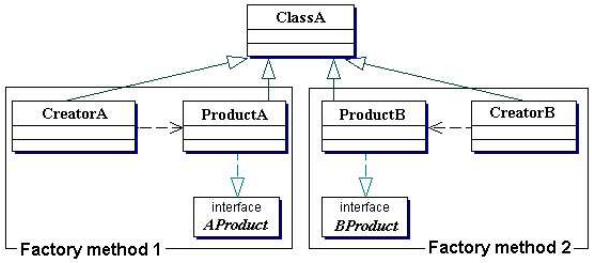



**Figure 7. Two Factory Method pattern realizations share a common superclass in System B.**

The real danger of coupling via a common superclass is that the dependencies between the pattern realizations cannot be identified by examining the pattern classes in isolation.

### 3.5 Common Coupling

Common coupling occurs when entities communicate through shared global structures. Myers describes common coupling as one of the tightest or worst forms of coupling, because it is implicit [31]. Entities may be coupled through common access to shared data, even though there is no direct connection between the coupled entities.

It is difficult to identify common couplings between design pattern realizations, because the coupling is implicit. The coupled pattern realizations have 'write' or 'read' access to the same variable. Any operations that write to the shared data must have the access controlled via synchronization. In addition, any defects or changes in the global variable will affect all of the pattern realizations that access the variable. Any changes in the patterns which access the global variable can affect the global variable, which, in turn, affects the other pattern realizations. Thus, maintaining a system with common coupling can be difficult. Fig. 8 shows an example from System B of common coupling between realizations of the Factory Method and Singleton (SingletonA) patterns through a common class (SingletonB) that acts as a global data entity. In this example, there is only one realization of class SingletonB. Both ProductA and SingletonA use SingletonB to log error messages and warning messages. Such coupling is hidden and can only be found by checking pattern realizations. In this example of common coupling, if class SingletonB changes something in methods or fields that ProductA and SingletonA use to log messages, then class SingletonA and ProductA must be changed. In this example, both realizations have 'write' access to the global object and the calling order does not need to be controlled. Thus, there is only weak coupling between these two pattern realizations. Coupling is strong when two design pattern realizations communicate through a global variable and both realizations 'read' and 'write' to the global variable. A change to a global variable or one of the coupled classes can ripple through to all of the pattern realizations or classes that access the variable, increasing the maintenance effort.

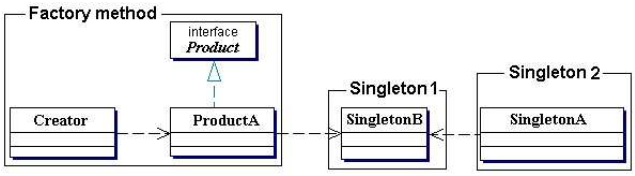



**Figure 8. Common coupling between a realization of the Factory Method pattern and a Singleton pattern in System B.**

## 4 Pilot Study: Observing Pattern Coupling and Changes

McNatt and Bieman found numerous examples of coupled pattern realizations in a set of 16 papers from the research literature [28]. Examples described in the papers include 99 realizations of patterns from Gamma et al [16]; 64 of the pattern realizations are coupled to other pattern realizations forming 25 connected pattern realization groups.

This prior work does not indicate the relative likelyhood of pattern coupling, since McNatt and Bieman included only examples that exhibit pattern coupling. Also, the systems studied are only partial systems. Thus, they did not determine the full extent of pattern coupling, the reasons for coupling, and consequences of pattern coupling.

Observations of multiple versions of full systems are required to identify overall occurrences of pattern coupling

and to examine the relationship of various pattern coupling mechanisms to change-proneness.

We examine five systems to identify and classify coupled patterns, determine the reasons for the coupling, and evaluate the potential of both positive and negative consequences. Table 1 provides high-level information about the systems that we studied. For each system, we examined the design structure in detail of a base system — an early version of the system. We extracted the object models of an early versions of each system using the Together tool from TogetherSoft. We identified design pattern realizations and coupled pattern realizations by analyzing the object models. Then we examined changes as the system evolved through several later versions. The systems are implemented in Java and are briefly described as follows:

- System A. System A is a proprietary system that is an early development in Java by a commercial organization. We examined 17 versions of System A.
- System B. System B is a more mature Java development project from the organization that developed System A. We examined 17 versions of System B.
- JRefactory. JRefactory is an open source system designed to refactor or restructure Java programs. It was developed and maintained by the Open Source community and is available through SourceForge.net. Version 2.6.38 of the system contains 699 classes and more than 47,000 lines of program code. We used JRefactory release 2.6.24 as the base version and studied the changes through version 2.6.27, 2.6.30, 2.6.31, 2.6.32, 2.6.34, 2.6.35, and 2.6.38, which were all the publicly released versions available at the time of the study.
- DrJava. DrJava is an open source software development environment for Java designed to foster test-driven software development. It was developed and maintained by the Open Source community and is available through SourceForge.net. Stable version 20030822 contains 564 classes with more than 49,000 lines of program code. We used DrJava version drjava-stable-20020703 as the base version and studied the changes through releases drjava-stable-20020814, drjava-stable-20021127, drjava-stable-20030113, drjava-stable-20030203, drjava-stable-20030313, drjava-stable-20030724, and drjava-stable-20030822.

We identified the pattern realizations, and pattern coupling, in base versions of all of the systems, and we conducted a detailed study of program changes in all of the systems. we have only one version of the Java implementation. Note that we changed the system names and class names for the proprietary systems (System A and System B) in this paper.

**Table 1. System Level Measurements.**

| System | Version | Num. of Classes | Lines of Code |
|---|---|---|---|
| Commercial Java System A | 2 | 384 | ∼23,000 |
| | 18 | 404 | ∼23,000 |
| Commercial Java System B | 2 | 101 | ∼7,500 |
| | 18 | 201 | ∼17,000 |
| JRefactory | 2.6.24 | 546 | ∼43,000 |
| | 2.6.38 | 699 | ∼47,000 |
| DrJava∗ | 20020703 | 311 | ∼25,000 |
| | 20030822 | 564 | ∼49,000 |
| OpenEJB | 0.8 | 95231 | 861 |
| | 0.9.2 | 61489 | 841 |

∗Only stable versions of DrJava are included.

Although it is possible to use automated methods to find some design pattern realizations [2, 20, 22], a manual approach can also effectively find pattern realizations [33]. In this research, we are looking for *intentional patterns*, pattern realizations that developers use in a deliberate, purposeful manner. These pattern realizations should be documented, and they should have a effect on the number of changes, since adaptability is the primary reason for using patterns — the indirection inherent in design patterns should reduce the number of changes to existing classes. Changes should be limited to adding new subclasses or other new classes that were not part of the original pattern realization. Because we seek to find only intentional patterns, we apply the following manual approach for pattern recognition with the following steps [4, 5]:

1. Search for pattern names in the documentation of the system. Developers are likely to document the pattern functionality/role of the class or method so that a pattern realization can be treated as a pattern realization during later development or maintenance.
2. Identify the context of the classes identified in step 1 by analyzing the object models. Once we find the classes whose documentation specifies something relating to a pattern name/role, we can look at the object models to identify all the classes required to constitute a pattern. We look for the links between classes that implement the pattern.
3. Verify that the candidate pattern realization is really a pattern realization. We examine the pattern realization to look for lower level details, for example, required delegation constructs.
4. Verify the purpose of the pattern realization. We examine each group of classes that represent a pattern candidate to confirm that the classes and relations have the same purpose as described by an authoritative pattern reference. We use the Gamma et al. [16] and

**Table 2. Patterns Identified in the Base Version of each System. The patterns are from Gamma et al. [16] and Grand [17].**

| Pattern | Number of Realizations | | | | |
|---|---|---|---|---|---|
| | A | B | JRefactory | DrJava | OpenEJB |
| Adaptor | 1 | | 16 | 3 | |
| Builder | | 1 | 2 | 1 | |
| Factory Method | 1 | 4 | | | 15 |
| Filter | | | 2 | 4 | |
| Iterator | | | | 1 | |
| Singleton | 1 | 3 | 1 | | 1 |
| State | 2 | | 3 | 1 | |
| Strategy | 1 | 1 | | 1 | 1 |
| Visitor | | | 2 | 3 | |
| Model View Controller | | | | 2 | |
| Master Slave | | | | 1 | |
| Interpreter | | | | 2 | |
| Command | | | | | 1 |

Grand [17] books as the authoritative references for this study.

Table 2 lists the patterns identified in each system and number of realizations of each pattern; In System A, 49 classes out of a total of 384 classes played roles in six pattern realizations of five design patterns — Adapter, Factory Method, Singleton, State, and Strategy patterns. In System B, 38 of the 102 classes played roles in nine pattern realizations of four pattern — Builder, Factory Method, Singleton, and Strategy patterns. We found 176 pattern classes in JRefactory out of a total of 699 classes that played roles in 26 pattern realizations of six patterns — Adapter, Builder, Filter, Singleton, State, and Visitor patterns. DrJava contains at least 19 pattern realizations of ten patterns — Adapter, Builder, Filter, Iterator, State, Strategy, Visitor, Model View Controller, Master Slave, and Interpreter.

Coupled pattern realizations are common in these systems. Table 3 shows the number of coupling occurrences in each category; each coupling involves two pattern realizations. We found four coupled pattern realizations among the six pattern realizations in System A. The pattern realizations in System B are extensively coupled — we found 52 couplings among only ten pattern realizations. Most of these couplings are due to shared common superclasses. The largest system, JRefactory has 26 pattern realizations and 82 pattern realization couplings.

Our empirical study focuses on the relationship between adaptability and pattern coupling. The notion of adaptability is difficult to define unambiguously and objectively.

**Table 3. Pattern coupling occurrences in systems.**

| Coupling Mechanism | | A | B | JRefactory | DrJava |
|---|---|---|---|---|---|
| Use Dependencies | | 3 | 6 | 12 | 4 |
| Associations | | | 2 | 16 | 2 |
| Sharing common classes | Embedded | 1 | 5 | 1 | |
| | Intersection | | 2 | 53 | 5 |
| Sharing common superclasses | | | 35 | | 7 |
| Common coupling | | | 12 | | |

Rather than measure adaptability directly, we use class change-proneness as a surrogate measure. To measure change-proneness, we examine the number of changes to program entities, primarily classes, over multiple versions of an evolving system.

## 4.1 Hypotheses.

We define our empirical hypotheses as follows:

$H1_0$ (Null Hypothesis 1): Pattern coupling is rare.

$H1_A$ (Alternate Hypothesis 1): Pattern coupling is common.

$H2_0$ (Null Hypothesis 2): There is no difference between the change-proneness of classes in design pattern realizations that are coupled to other patterns and pattern realizations that are not coupled.

$H2_A$ (Alternate Hypothesis 2): Classes in design pattern realizations that are coupled to other pattern realizations will be more change-prone than those in pattern realizations that are not coupled.

$H3_0$ (Null Hypothesis 3): There is no difference between the change-proneness of classes in design pattern realizations that are coupled via the following mechanisms: use dependency coupling, association, embedded, intersection, sharing common super classes, and common coupling between design pattern realizations.

$H3_A$ (Alternate Hypothesis 3): The relative change-proneness of coupled pattern realizations can be ordered in terms of coupling mechanism using the following ordering from least change-prone to most change-prone: use dependency coupling, association, embedded, intersection, sharing common super classes, and common coupling between design pattern realizations.

We evaluate the hypotheses by examining pattern coupling and change-proneness in four systems — the systems in Table 1 using the following process:

1. Find the pattern realizations in a base version of each system.

2. Find the coupled pattern realizations and identify the coupling mechanisms in each base version.

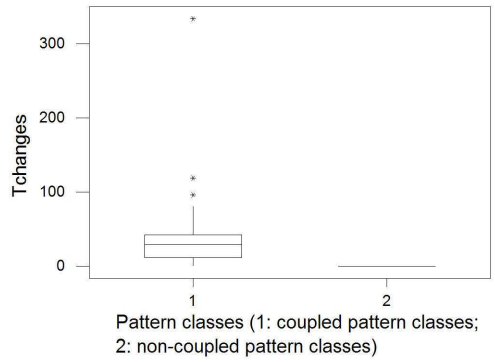
3. Identify all changes to each pattern realization, from the base version of each system through the last version in our data.

### 4.2 Evaluating H1 and H2.

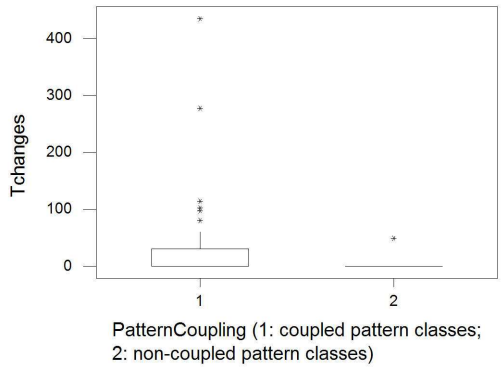
Virtually all of the design pattern realizations are coupled to other pattern realizations. The only exceptions are in System A and DrJava. In System A, only two of the six pattern realizations are not coupled. In DrJava, 16 of the 19 pattern realizations are coupled. Clearly, pattern coupling is common in these systems, providing support for rejecting $H1_0$ and accepting $H1_A$.

Due to the limited number of non-coupled patterns, it is difficult to evaluate H2. However, in System A, the two uncoupled pattern realizations had no changes, while the six coupled pattern realizations were changed as shown in Figure 9 (a). A Kolmogrov-Smirnov two-sample test shows that the difference in change-proneness of coupled versus non-coupled pattern realizations is significant at the 0.0000 level. These results hold if we normalize the indicator of change-proneness by class size by using changes per operation rather than the total changes per class. Figure 9 (a) shows that coupled pattern realizations are more change-prone than non-coupled pattern realizations in DrJava. A Mann-Whitney test shows that the difference in change-proneness of coupled versus non-coupled pattern realizations is significant at the 0.0367 level. This difference is less significant (0.0775) if we normalize the indicator of change-proneness by class size by using changes per operation rather than the total changes per class.

Nonparametric statistical tests are appropriate, because the data is not normally distributed. Usually, to compare two samples concerning average value for some variable of interest, the following tests are appropriate for nonparametric data: the Wald-Wolfowitz runs test, the Mann-Whitney U test, and the Kolmogorov-Smirnov two-sample test. To be consistent with the analyses used in our prior studies [4,5], we apply the Mann-Whitney when possible. However, the Mann-Whitney does not work to analyze the difference in change proneness between pattern and non-pattern classes in System A, because all changes for non-coupled pattern classes are identical — the number of changes for all non-coupled pattern classes is zero. The Kolmogorov-Smirnov two-sample works for this data.




(a) System A




(b) DrJava

**Figure 9. The number of changes in coupled non-coupled pattern realization classes in System A and DrJava.**

### 4.3 Evaluating H3

Only System B, JRefactory, and DrJava contain enough cases of coupled patterns in multiple categories to evaluate H3. JRefactory includes patterns coupled via associations, use dependencies and shared classes. Since JRefactory has only one only case of embedded coupling, we included it with the 53 cases of intersection coupling in the more general category of coupling via shared common classes.

Figure 10 shows the distribution of changes for each category of coupled patterns in System B, JRefactory, and DrJava. In all three systems, classes that are coupled via associations appear to be more change-prone than those coupled by other mechanisms. However, a Man-Whitney test shows that this difference is significant at the 0.05 level only in

JRefactory — a Man-Whitney test gives a significance of .0000 that we can reject the null hypothesis ($H3_0$) that the association coupled pattern classes are not more change-prone than those coupled via either use dependencies or shared classes. The distributions for all of the other mechanisms in JRefactory appear to be quite similar. In DrJava, the difference between the number of changes in of association coupled pattern classes and (1) use dependency coupled pattern classes is significant at the 0.1093 level, and (2) coupling via shared common classes are significant at the 0.1738 level. This level of significance (for DrJava) indicates a relationship that is not quite strong enough to reject $H3_0$. Again, we get similar results when we use changes per operation as the indicator of change-proneness.

We get somewhat different results by comparing the number of changes in patterns that are coupled via associations against changes in patterns coupled by other mechanisms, when the other mechanisms are treated as a single category. Figure 11 shows that classes in patterns coupled via associations are more change-prone in System B, JRefactory, and DrJava. This difference is significant in JRefactory and DrJava at the 0.000 and 0.0046 level respectively via a Mann-Whitney Test. However, the relationship is not significant in System B. The results are similar using changes per operation as the indicator of change-proneness.

Our data, though limited, does not support $H3_0$, but we cannot convincingly reject this null hypothesis. It does appear that patterns coupled through associations are more change-prone than those coupled through other mechanisms. However, the results are not significant in all cases. Additional data is needed.

### 4.4 The Ripple Effect and Change-coupling

One indication of the strength of the coupling mechanisms is the ripple effect [36,35] — how changes in one pattern realization cause changes in other pattern realizations. These entities are *change-coupled* [3]; they are modified as elements of one required change. We evaluated changes and identified the classes that were changed in response to the each required change in JRefactory and DrJava. Two key criteria allowed us to group class changes as elements of one change: matching comments documenting the changes, and matching check-in time stamps — check-ins witin one minute of each other. We did not rely on change-logs, which can be incomplete [12]. Similar data was not available for System B. Table 4 and Table 5 shows the number of coupled pattern realization pairs in JRefactory and DrJava respectively that are changed together in response to one or more change request. Patterns coupled via associations had the greatest tendency to be changed together — - 62.5% of these coupled pattern pairs in JRefactory and both of these pairs in DrJava are change coupled. Patterns coupled through shared classes had the second greatest tendency to be changed together — 46.3% of these coupled pattern pairs in JRefactory and 42.9% in DrJava are change coupled. Only DrJava contains pattern realizations coupled through sharing common superclasses; 42.9% of these coupled patterns are change coupled. None of the pairs that were coupled via use dependencies were changed together, and there were no cases of common coupling.

In JRefactory the difference in change-coupling of pattern realizations coupled via associations versus those coupled via sharing common classes is significant via a Mann-Whitney test at the 0.0204 level. That is, pattern realizations coupled via associations are more likely to be change coupled than those coupled via common classes.

We do not have enough data on change-coupled pattern realizations in DrJava to apply statistical tests. However, the preliminary results suggests that pattern realization pairs with association couplings are more likely to be change-coupled than pairs with use dependency couplings. Further data is needed to confirm this relationship.

### 4.5 Threats to Validity.

This empirical work is preliminary, and the results are not conclusive. We assess four types of threats to the validity of the empirical study: construct validity, content validity, internal validity and external validity. Construct validity refers to the meaningfulness of measurements [21,32] — do the measures actually quantify what we want them to? To validate the meaningfulness of measurements, we need to show that the measurements are consistent with an empirical relation system, which is an intuitive ordering of entities in terms of the attribute of interest [14, 23, 29]. A primary objective is to determine whether or not there is support for the proposed empirical relation system representing the strength of pattern coupling types, the independent variable. The dependent variable in this study, a count of changes, and change coupling, is an intuitive measure of an aspect of maintenance effort. However, not all changes are equal, but a large number of changes over a many versions should minimize the impact of change effort variability. Further study can determine the distribution of effort per changes; actual change effort data was not available for this study.

Content validity refers to the "representativeness or sampling adequacy of the content ... of a measuring instrument" [21]. The content validity of this research depends on whether the set of mechanisms of pattern coupling and maintainability adequately cover the notion of design quality and maintainability respectively. As the notion of pattern coupling is relatively new, we risk missing important coupling mechanisms. The count of changes and change coupling quantifies only two aspects of maintenance effort in our empirical study.

**Table 4. Coupling mechanisms and the number of pairs of coupled pattern realizations that are changed together in JRefactory.**

| Coupling Mechanism | | Number of coupled pattern realization pairs | Number of coupled pattern realization pairs that are changed together | Ratio |
|---|---|---|---|---|
| Use Dependencies | | 12 | 0 | 0 |
| Associations | | 16 | 10 | .625 |
| Sharing common classes | Embedded | 1 | 25 | .463 |
| | Intersection | 53 | | |
| Sharing common superclasses | | 0 | | |
| Common coupling | | 0 | | |

**Table 5. Coupling mechanisms and the number of pairs of coupled patterns that are changed together in DrJava.**

| Coupling Mechanism | | Number of coupled pattern realization pairs | Number of coupled pattern realization pairs that are changed together | Ratio |
|---|---|---|---|---|
| Use Dependencies | | 4 | 2 | .5 |
| Associations | | 2 | 2 | 1.0 |
| Sharing common classes | Embedded | 0 | 4 | .8 |
| | Intersection | 5 | | |
| Sharing common superclasses | | 7 | 3 | .429 |
| Common coupling | | 0 | | |

Internal validity focuses on cause and effect relationships. The notion of one thing leading to another is applicable here and causality is critical to internal validity. The statistical results are not consistent. Classes that are association-coupled are more change-prone than those coupled by other means in all cases. However, this result is significant (at the .05 level) in only three of the six cases. Even if the results were significant, they do not demonstrate causality, on their own. To show causality, we need to show temporal precedence — evidence that cause precedes effect, and demonstrate a theory that defines a mechanism for the relationships [11, 34]. In our study, pattern coupling data were collected from software versions created before the change activity, and we provide causal explanations for the effect of pattern coupling on changes, which were expressed as hypothesis. The results were strong enough to allow us to reject the null hypothesis for H1 and H2. Results supported hypothesis H3, but they were not strong enough to reject the null hypothesis for H3.

External validity refers to how well the study results can be generalized beyond the study data. The data used to evaluate H1 is based on five systems, the data used to evaluate H2 is based on one system, and H3 is evaluated using data from three systems. These systems are all implemented in Java and include two proprietary systems, one student implementation, and two open source systems. They represent a variety of systems. However, we do not have a good enough characterization of the universe of object-oriented systems to make a claim concerning external validity. Prior work by Bieman et al. demonstrates that that the design characteristics and change-proneness can vary greatly between systems [5].

## 5 Conclusions

The use of design patterns as design constructs leads to concerns about coupling between design pattern realizations and the strength of such couplings. Our study of six systems indicates that realizations of design patterns are commonly coupled with other design pattern realizations. Most of the pattern realizations in these systems are coupled with other pattern realizations. We have identified a set of coupling mechanisms and proposed an ordering of these mechanisms in terms of the tightness or coupling strength implied by the mechanism. An examination of the evolution of five of the systems indicates that classes in pattern realizations that are coupled through associations, usually created via references in instance variables, are among the most change-prone classes in the systems. Further, evidence from two open source systems suggests that classes that play roles in pattern realizations that are coupled via associations tend to be change-coupled — these coupled classes are changed simultaneously. Further research is needed to confirm these results.

## Acknowledgements

This material is based on work supported by the U.S. National Science Foundation under grant CCR-0098202. Storage Technology Corporation provided software, tools, and computer resources for this study. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] F. Abreu, M. Goul ao, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. *Proc. Fifth Int. Conf. Software Quality*, October 1995.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using metrics to identify design patterns in object-oriented software. *Proc. IEEE-CS Software Metrics Symp. (Metrics'98)*, 1998.

[3] J. Bieman, A. Andrews, and H. Yang. Understanding change-proneness in OO software through visualization. *Proc. Int. Workshop on Program Comprehension (IWPC 2003)*, pages 44–53, May 2003.

[4] J. Bieman, D. Jain, and H. Yang. Design patterns, design structure, and program changes: an industrial case study. *Proc. Int. Conf. on Software Maintenance (ICSM 2001)*, pages 580–589, 2001.

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

[6] G. Booch. Growing the UML. *Software and Systems Modeling*, 1(2):157–160, December 2002.

[7] L. Briand, J. Daly, V. Porter, and J. Wüst. A comprehensive empirical validation of design measures for object-oriented systems. *Proc. Int. Software Metrics Symp. (Metrics'98)*, pages 246–257, 1998.

[8] L. Briand, J. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*, 25(1):91–121, 1999.

[9] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. *Proc. 19th Int.*

*Conf. Software Engineering (ICSE'97)*, pages 412–421, 1997.

[10] L. Briand, J. Wüst, S. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. *Proc. Int. Conf. Software Engineering (ICSE'99)*, pages 345–354, 1999.

[11] D. Campbell and J. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Co., Boston, 1966.

[12] K. Chen, S. Schach, L. Yu, J. Offutt, and G. Heller. Open-source change logs. *Empirical Software Engineering*, 9:197–210, 2004.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[14] N. Fenton and S.L. Pfleeger. *Software Metrics - A Rigorous and Practical Approach Second Edition*. Int. Thompson Computer Press, London, 1997.

[15] Robert France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, March 2004.

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

[17] M. Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, 2nd Edition, Volume 1*. John Wiley and Sons, New York, 2002.

[18] R. Harrison, S. Counsell, and R. Nithi. Coupling metrics for object-oriented design. *Proc. Proc. Int. Symp. Software Metrics (Metrics 1998)*, pages 150–157, 1998.

[19] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object oriented systems. *Proc. Int. Symp. Applied Corporate Computing*, 1995.

[20] R. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design concepts. *Proc. Int. Conf. on Software Engineering (ICSE'99)*, pages 226–235, 1999.

[21] F. Kerlinger. *Foundations of Behavioral Research, Third Edition*. Harcourt Brace Jovaonvich College Publishers, Orlando, Florida, 1986.

[22] C Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. *Proc. Working Conf. on Reverse Engineering*, pages 208–215, 1996.

[23] D. Krantz, R. Luce, P. Suppes, and A. Tversky. *Foundations of Measurement*, volume I Additive and Polynomial Representations. Academic Press, New York, 1971.

[24] Y.S. Lee, B.S. Liang, S.F. Wu, and F.J. Wang. Measuring the coupling and cohesion of an object-oriented program based on information flow. *Proc. Int. Conf. Software Quality*, 1995.

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

[26] M. Lindvall, R. Tesoriero Tvedt, and P. Costa. An empirically-based process for software architecture evaluation. *Empirical Software Engineering*, 8:83–108, 2003.

[27] R. Martin. Oo design quality metrics — an analysis of dependencies. *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*, October 1994.

[28] W. McNatt and J. Bieman. Coupling of design patterns: Common practices and their benefits. *Proc. COMPSAC 2001*, pages 574–579, 2001.

[29] J. Michell. *An Introduction to the Logic of Psychological Measurement*. Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1990.

[30] Leonid Mikhajlov1 and Emil Sekerinski. A study of the fragile base class problem. *Proc. European Conf. Object-Oriented Programming (ECOOP'98)*, 1998.

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

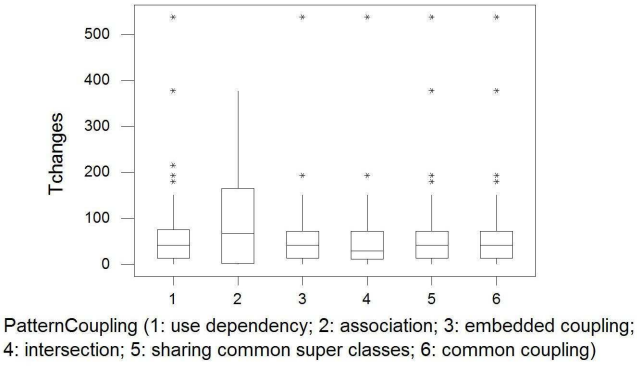
[32] J. Nunnally. *Psychometric Theory, Second Edition*. McGraw-Hill, New York, 1978.

[33] F. Shull, W. Melo, and V. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical Report UMCP-CSD CS-TR-3597 or UMIACS-TR-96-10, University of Maryland, Computer Science Dept., 1996.

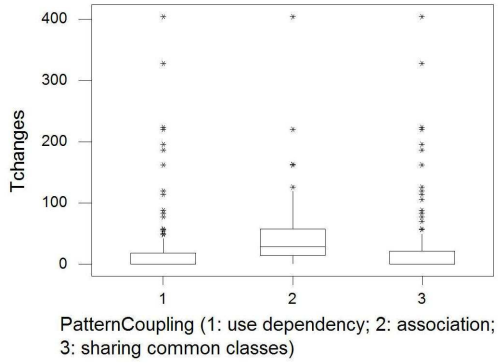
[34] L. Votta and A. Porter. Experimental software engineering: A report on the state of the art. *Proc. 17th Int. Conf. Software Engineering (ICSE'95)*, 1995.

[35] S.S. Yau and J.S. Collofello. Some stability measures for software maintenance. *IEEE Trans. Software Engineering*, SE-6(6):545–552, November 1980.

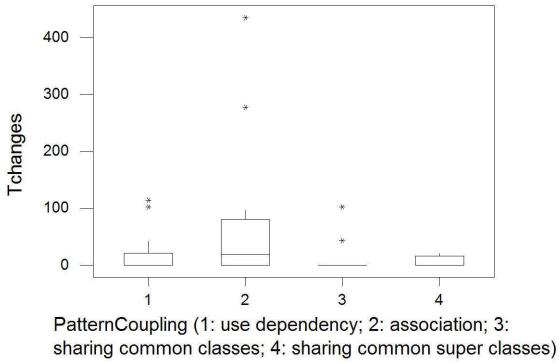
[36] S.S. Yau, J.S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. *Proc. COMPSAC 78*, pages 60–65, 1978.




(a) System B

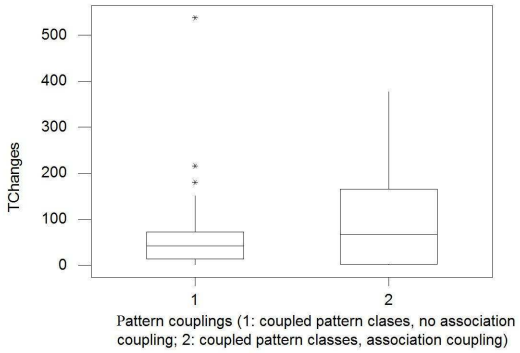



(b) JRefactory

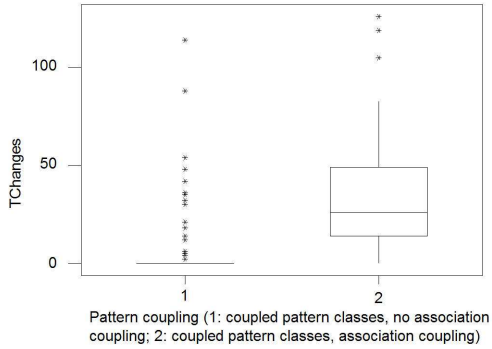



(c) DrJava

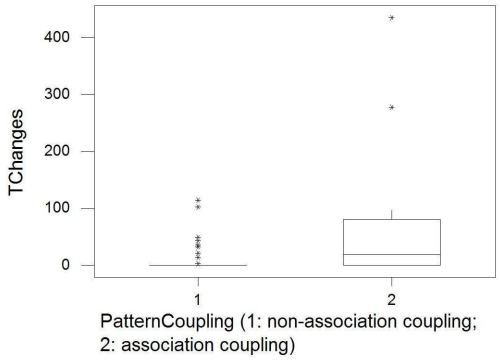
**Figure 10. Number of changes for coupling mechanisms in System B, JRefactory, and DrJava.**




(a) System B




(b) JRefactory




(c) DrJava

**Figure 11. Number of changes in pattern realizations coupled via associations and other coupling mechanisms in System B, JRefactory, and DrJava.**