

An Empirical Study on the Influence of Pattern Roles on Change-Proneness

Daryl Posnett · Christian Bird · Prem Dévanbu

Abstract Identifying change-prone sections of code can help managers plan and allocate maintenance effort. Recently, *design patterns* have been used to study change-proneness, and are widely believed to support certain kinds of changes, while inhibiting others. Recently, several studies have analyzed recorded changes to classes playing design pattern roles and find that the patterns “folklore” offers a reasonable explanation for the reality: certain pattern roles *do* seem to be less change-prone than others. We push this analysis on two fronts: first, we deploy W. Pree’s *metapatterns*, which group patterns purely by structure (rather than intent), and argue that metapatterns are a simpler model to explain recent findings by Di Penta et al. (2008). Second, we study the effect of the *size* of the classes playing the design pattern and metapattern roles. We find that size explains more of the variance in change-proneness than either design pattern or metapattern roles. We also find that both design pattern and metapattern roles were strong determinants of size. We conclude, therefore, that size appears to be a stronger determinant of change-proneness than either design pattern or metapattern roles, and observed differences in change-proneness between roles might be due to differences in the sizes of the classes playing those roles. The size of a class can be found much more quickly, easily and accurately than its pattern-roles. Thus, while identifying design pattern roles may be important for other reasons, as far as identifying change-prone classes, sheer size might be a better indicator.

Keywords Design Patterns · Empirical Software Engineering

CR Subject Classification D.2.8 · D.2.11

1 Introduction

Change-proneness is often thought of as a proxy for maintainability. Classes that are more change-prone have been shown to require more maintenance effort (Güneş Koru and Liu,

Daryl Posnett · Christian Bird · Prem Dévanbu

University of California Davis
Tel.: +530-752-7004
Fax: +530-752-4767
E-mail: darylp,cabird,ptdevanbu@ucdavis.edu

2007). Since reducing maintenance effort is a key goal of software engineering, it is important to understand the impact of software design choices on change-proneness. Design patterns (Gamma et al., 1995) promise to help make code easier to evolve. Specifically, patterns allow classes to be assembled into a design unit, or motif, where each class plays a specific pattern *role*. Patterns are thought to allow classes playing certain roles to evolve more easily than others. It is believed that this flexibility is partly responsible for avoiding reimplementations and client modifications (Gamma et al., 1995). A recent study by Di Penta *et al.* tends to confirm intuitions about the change-proneness of classes playing classic Gamma *et al.* design pattern roles (Di Penta et al., 2008).

A related question of interest is whether change-proneness is simply a consequence of the inheritance and association/aggregation *structure* of a design pattern, or if it is related to other design pattern properties.

Object-oriented programming (OOP) has at its core the idea of programming to an interface, i.e. declaring objects of a parent interface type rather than a particular child class type. This practice decouples the clients of the parent interface and the indirectly referenced class hierarchy. Pree presented *metapatterns*, which can be viewed as a purely structural view of design patterns (Pree, 1994). Metapatterns arise from this core OOP principle of *object composition* of the clients to the parent interface, and the *class composition* of the parent interface into the children. Most design pattern motifs can be viewed as one or more instances of metapatterns; structurally similar design pattern motifs instantiate the same metapatterns. Gamma *et al.* assert that “designers overuse inheritance as a reuse technique and, designs are often made more reusable ... by depending more on object composition.” (Gamma et al., 1995) Consequently, we observe that metapatterns model a subset of design pattern properties that are believed to facilitate reuse and limit change-proneness.

The findings of Di Penta *et al.* suggest that observed differences in change-proneness in classes playing design pattern roles could be explained more simply, just by structure, *viz.*, metapattern roles. We expect for example, a STRATEGY role to change less often than a CONCRETE STRATEGY role because changes to the former may ripple through both the CONTEXT hierarchies where overridden methods are defined and the STRATEGY class hierarchies where these overridden methods are called. This intuition is not dependent on any property of design patterns that are not also captured by metapatterns. The intent of the STRATEGY pattern does not affect reported intuition regarding change proneness, rather, the reported intuition is simply a consequence of the class and object relationships modeled by metapatterns.

Since metapatterns are defined by a subset of design pattern properties, they are more easily detected. With the exception of SINGLETON, MEMENTO, and FACADE, every Gamma *et al.* design pattern motif inherently contains a metapattern; thus, we will find at least one metapattern for each such design pattern. Furthermore, since metapatterns almost always have fewer detection requirements than design patterns, we typically find some additional metapatterns where no design pattern exists. Consequently, detected metapattern instances are usually at least as numerous as the design pattern instances that include metapatterns, up to any limitations in detection tools.

Earlier studies of change-proneness indicate that *size* influences change-proneness (Bieman et al., 2001, 2003) of classes that participate in design patterns. The authors expected that classes involved in design patterns would be less change-prone, however, they observed that such classes were still more change-prone after accounting for size than those not participating in design patterns.

In a later study, Aversano et al. (2007) observed that classes participating in patterns that “play a very important role for a(n) ... application” are more change-prone and that classes not participating in patterns are often not key participants in the application’s design. Consequently, their results support the work of Bieman but do not shed any further light on

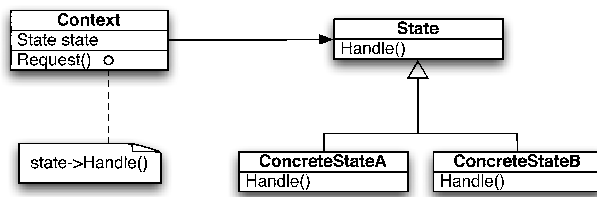


Fig. 1: *The STATE pattern*

the question of *whether the various design pattern roles actually offer the stability suggested by the literature.*

In this paper we study classes playing pattern roles within the same pattern and application to shed further light on the question of the relationship between design patterns and class stability

1.1 Outline

We present the following results:

1. We observe that differences in change-proneness of *design-pattern roles* reported by Di Penta *et al.* (Di Penta et al., 2008) can also be explained by the purely structural notion of *metapattern roles*. We observe similar patterns of change-proneness among design patterns that employ the same metapattern model as well as in the independently measured metapatterns.
2. However, when controlling for the *sizes* of the classes playing these (pattern and metapattern) roles, we find that the roles add very little explanatory power.
3. We also find that *sizes* of the classes are strongly associated with the metapattern roles played by the classes; leading to the conclusion that while pattern and metapattern roles do partially explain change-proneness, the dominant effect is indirectly through size, i.e. classes playing certain metapattern roles are larger.

Our work is in the spirit of Perry & Porter and Basili & Elbaum (Basili and Elbaum, 2006; Perry et al., 2000) who point out that replications and integrating multiple studies are critical to gain confidence in empirical results. In addition, our findings suggest that some widely held intuition about the change-proneness of classes playing various roles in patterns can be explained by the simpler relationships of the underlying metapatterns.

We begin with a quick overview of metapatterns (§ 2), leading to a formulation of the main research questions. We then present a detailed review of metapatterns (§ 3) and discuss related work (§ 4). We describe (§ 5) our data extraction approach. We then present our results (§ 6), discuss threats, and conclude. Throughout this paper, we refer to the patterns introduced in the classic GOF (*Gang of Four*) Book (Gamma et al., 1995) as *design patterns* and the purely structural patterns presented by Pree as *metapatterns*.

2 Overview

Metapatterns capture the *pure structure* of design patterns. As an illustration, we describe the structural similarity between STATE and STRATEGY patterns. First, consider the STATE

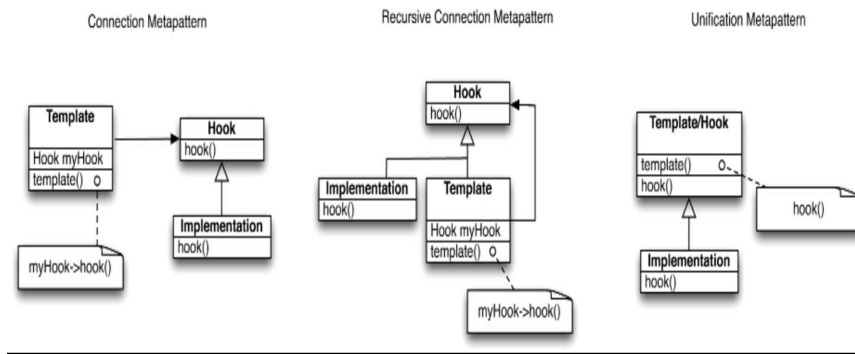


Fig. 2: The basic metapattern structures

design pattern (Fig. 1) with the `CONTEXT`, `STATE`, and `CONCRETESSTATE` roles.¹ Calls from clients (not shown) to `Context.Request()` are forwarded to the base-class method `State.Handle()` and then (e.g.) to the implementation `ConcreteStateA.Handle()` via the `STATE` instance variable. The `STRATEGY` design pattern has an analogous structure, consisting of `CONTEXT`, `STRATEGY` and `CONCRETESSTRATEGY`. Both `STATE` and `STRATEGY` have a *stable* part, and a *changeable* part. The `CONTEXT` role in these patterns is used by clients of the pattern, and could change in response to changing client needs. The `CONTEXT` role makes use of the `STATE/STRATEGY` role, which defines an interface, and thus is relatively fixed. This interface is implemented by `CONCRETESSTATE/CONCRETESSTRATEGY` roles, which provide varying implementations of the `STATE/STRATEGY` roles. The class playing the `STATE/STRATEGY` role, by remaining stable, effectively decouples the classes playing the other two roles. Pree noticed the structural similarity of these two patterns, and named the shared structure the 1-1-CONNECTION metapattern.

The `CONTEXT` role in the `STATE/STRATEGY` pattern is named the `TEMPLATE` meta-role in the 1-1-CONNECTION metapattern; the interface (`STATE` or `STRATEGY` role) is named the `HOOK` meta-role; and the changeable roles (`CONCRETESSTATE` or `CONCRETESSTRATEGY`) are called the `IMPLEMENTATION` meta-roles.

Pree defines 7 structural metapatterns and shows that most design patterns have, at their core, an instance of one or more of these metapatterns (more details in section 3).

2.1 Research Questions

A close reading of earlier results by Di Penta et al. (2008) on the relative change-proneness of classes playing different roles in a design pattern, suggests that the findings could be grouped by the underlying metapattern roles. Other studies of change-proneness in patterns suggest that *size* has a strong relationship to change-proneness (Bieman et al., 2001, 2003).

Size is an appealing, baseline phenomenon: the larger a component, the more likely some part of it changes. Moreover, we agree with Briand and Wust (2002) who assert that “the size of an artifact is a necessary part of any model predicting a property ... of this artifact.” If we are relating change-proneness to some property, we want to know that this property is telling us more than just that the larger classes are more change-prone. To this end it is often prudent to include size as a control in any models that relate properties potentially influenced by size to a particular outcome (El Emam et al., 2001). If our goal is

¹ In this example, classes have been given the same name as the roles for clarity.

to understand a phenomenon such as how design pattern roles affect change proneness, then it is necessary to determine if the role attributes that we are measuring are an artifact of the design pattern role, or simply a proxy for size. And so we ask: Do the observed differences in change-proneness hold up, even when accounting for the sizes of the respective classes playing the pattern or metapattern roles? We begin by studying the influence of size on the change-proneness of classes playing traditional design pattern roles.

Research Question 1: To what extent does design pattern *role* explain the variation in change-proneness of classes, when controlling for class size?

Our second research question considers the same issue, with respect to metapatterns:

Research Question 2: To what extent does metapattern *role* explain the variation in change-proneness of classes, when controlling for class size?

Next, we consider the relative explanatory power of pattern roles and metapattern roles:

Research Question 3: When controlling for size, do metapattern roles explain as much of the variation in change-proneness as design pattern roles?

Finally, we consider the impact of design pattern and metapattern roles on the sizes of the classes playing those roles. If the roles explain a significant level of the variance in size, then the differences in change-proneness of roles might simply be an indirect, mediated effect of the relative size differences in the classes playing those roles.

Research Question 4: Do (the purely) structural metapattern roles and pattern roles explain the variation in the sizes of classes playing those roles?

3 Phylogeny: Metapatterns and Patterns

Metapatterns are rooted in two structural roles, `TEMPLATE` and `HOOK`. A `TEMPLATE` is a class with a method t that calls a method h in the `HOOK` class (or interface). The `TEMPLATE` provides a “template” to accomplish a goal and the `HOOK` provides a way in, or a “hook,” into a flexible class hierarchy including the hook and its descendants. We use the same terms, without ambiguity, to refer to the `TEMPLATE` and `HOOK` methods, i.e. `HOOK` may refer to either a method or a class, depending on context. To make use of the `HOOK` class hierarchy, the `TEMPLATE` method must invoke the `HOOK` method through some variable or parameter f in the `TEMPLATE` class. This variable provides the link between the algorithm and the collection of classes available in the `HOOK` class hierarchy. The cardinality of f defines the cardinality of the instance relationship between the `TEMPLATE` and `HOOK` classes. When f is a container the `TEMPLATE` may invoke any number of `HOOK` instances and the relationship is $1 : N$. Alternatively, if f is a simple scalar instance of `HOOK` then `TEMPLATE` may invoke methods in only one `HOOK` instance and the relationship is $1 : 1$. A subject may have to update many observers so this would indicate a $1 : N$ relationship. On the other hand, a document formatting strategy may hold only a single document formatter reference even though it might choose one of several concrete formatters; this single object reference defines a $1 : 1$ relationship.

In every case, the `HOOK` method in the `HOOK` class can be overridden by methods in one or more `HOOK IMPLEMENTATION` (hereafter referred to as `IMPLEMENTATION`) classes derived from the `HOOK` class. In the example above, the concrete strategies that implemented different document formatters would play the `IMPLEMENTATION` roles.

The patterns are shown in Figure 1 and are defined as follows:

1. If the `HOOK` to `TEMPLATE` relation is purely associative or aggregative, it is a 1:1 or 1:N `CONNECTION` metapattern.
2. If `TEMPLATE` inherits from `hook`, it is a 1:1 or 1:N `RECURSIVE CONNECTION` metapattern.
3. If `TEMPLATE` and `HOOK` are the same class, this is a `UNIFICATION` metapattern.
4. If `TEMPLATE` and `HOOK` are the same class type, but `TEMPLATE` references or aggregates one or more instances of its own type, it is a 1:1 or 1:N `RECURSIVE UNIFICATION` metapattern.

3.1 Connection Metapatterns

In the `CONNECTION` metapattern, the `TEMPLATE` method delegates a specific task to the `HOOK` method. The `HOOK` method is an “articulation point” allowing the `IMPLEMENTATION` and `TEMPLATE` classes to change independently.

The first row of Table 1 lists the traditional design patterns that instantiate the 1-1-`CONNECTION` metapattern. Because the `hook` serves as the base class for one or more implementations, we can expect that the `HOOK` role is relatively less change prone than the `IMPLEMENTATION` or the `TEMPLATE` roles. This mirrors the intuition reported by Di Penta *et al.* across many design pattern roles such as those of the `ADAPTER`, `COMMAND`, `STATE`, `STRATEGY`, and `OBSERVER` patterns.

In general, given a pattern role, the corresponding meta-role is evident. For example, the `BUILDER` pattern, a 1-1-`CONNECTION` metapattern: the `BUILDER` pattern role maps to a `HOOK` role, the `DIRECTOR` to the `TEMPLATE` role, and the `CONCRETEBUILDER` to the `IMPLEMENTATION` role. We list the typical meta-roles for the canonical GOF structures in table 5.

The 1-N-`CONNECTION` metapattern is similar, to the 1-1-`CONNECTION` metapattern except that the `TEMPLATE` role may aggregate or reference multiple instances of the `HOOK` role objects. With respect to class structure and change-proneness we expect that 1-N-`CONNECTION` and 1-1-`CONNECTION` are similar: `HOOK` should change less than `IMPLEMENTATION` or `TEMPLATE`. Instances of 1-N-`CONNECTION` are shown in row 2 of Table 1.

3.2 Recursive Metapatterns

In the recursive metapatterns, the `TEMPLATE` class inherits from the `hook` class. This means that it calls into its own hierarchy, and typically must provide an implementation, as well as a “template”, for the `hook` class. Playing both `IMPLEMENTATION` and `TEMPLATE` roles complicates the `TEMPLATE` class: it acts as both a client of an algorithm, and often as an implementation of parts of the algorithm. The `TEMPLATE` is more strongly linked to the `HOOK` than in the non-recursive variant.

In the `DECORATOR` pattern, the `COMPONENT` class plays the `HOOK` role providing the interface for both `CONCRETE COMPONENT` and `DECORATOR`. The `DECORATOR` plays the `TEMPLATE` role and often a default `IMPLEMENTATION` role. the `CONCRETE COMPONENT` class also plays the `IMPLEMENTATION`. Because the `DECORATOR` can be subclassed by a `CONCRETE DECORATOR` that must correctly invoke methods on the `COMPONENT` classes that they decorate, the classes are more tightly bound than in many other patterns. The `DECORATOR` class holds a reference to a `CONCRETE COMPONENT` that is also a descendant of the `COMPONENT (HOOK)` and typically the `DECORATOR` methods forward requests from one `CONCRETEDECORATOR` to the next. Changes

Meta Pattern	Design Pattern
1 : 1 Connection	BRIDGE, BUILDER, MEDIATOR STATE, STRATEGY, VISITOR
1 : N Connection	ABSTRACT FACTORY, COMMAND, FLYWEIGHT, ITERATOR, OBSERVER, PROTOTYPE, ADAPTER PROXY, VISITOR
1 : 1 Recursive Connection	DECORATOR
1 : N Recursive Connection	COMPOSITE, INTERPRETER, VISITOR
Unification	FACTORY METHOD, TEMPLATE METHOD
1 : 1 Recursive Unification	CHAIN OF RESPONSIBILITY
1 : N Recursive Unification	None
No Metapattern	FACADE, MEMENTO, SINGLETON

Table 1: *Classifying design patterns into metapatterns. All the metapatterns are presented for completeness.*

in the base COMPONENT role, and changes in forwarding logic will induce changes across the DECORATOR subtree. Thus, we expect the DECORATOR, playing the TEMPLATE metapattern role, to be more change-prone than either its HOOK, or even the HOOK subclasses.

We expect the 1-N-RECURSIVE-CONNECTION metapattern to show a similar pattern of change-proneness to the 1-1-RECURSIVE-CONNECTION metapattern.

3.3 Unification Metapatterns

The UNIFICATION metapatterns combine TEMPLATE and HOOK both roles in a single class. Most patterns of this form do not contain an explicit HOOK reference and the HOOK call is made implicitly through the `this` reference. The recursive form of this pattern combines TEMPLATE and HOOK methods into a single recursive method. Since no design patterns employ the RECURSIVE UNIFICATION metapattern structure (see Table 1), we ignore it. The (non-recursive) UNIFICATION metapattern may be found in the FACTORY METHOD and TEMPLATE METHOD design patterns. There is no strict mapping between these patterns and the UNIFICATION metapattern, however, and both can also be implemented with the CONNECTION metapattern.

4 Related Work

There has been significant research effort focused on the validation of design pattern claims. The earliest results focused on comparing development efforts both with and without design patterns. In a controlled paper and pencil experiment on the effectiveness of design patterns, Prechelt and Unger found that while DECORATOR had a positive effect on program maintenance as compared to a simpler non pattern based design solution, OBSERVER had a negative effect; the benefits of ABSTRACT FACTORY were small, and, contrary to expectation, the VISITOR was neither beneficial nor detrimental (Prechelt et al., 2001). Vokáč replicated

this experiment in a real programming environment and found similar effects for DECORATOR and ABSTRACT FACTORY, a very strong negative effect for VISITOR and a positive effect for OBSERVER (Vokáč et al., 2004).

Ng *et al.* found that the difficulty of performing maintenance tasks is dependent on whether the subjects are changing concrete participants, abstract participants, or client code (Ng et al., 2007) and that even inexperienced coders were more able to affect changes on a system re-factored to design patterns than on the original non-design pattern based system (Ng et al., 2006).

Bieman *et al.* studied change-proneness in design pattern instances within five systems comparing classes participating in design patterns to those that played no role within any design pattern (Bieman et al., 2001, 2003). They theorized that, since design patterns support adaptability, classes instantiating design patterns should be more stable, and adaptations will occur via specialization of existing classes in preference to modification of existing classes. Contrary to *their* expectation, they found that classes participating in design patterns actually change more than classes not participating in design patterns. One interpretation of this result might be that design patterns do not lead to greater adaptability and lower change proneness. This interpretation does not, however, account for systematic differences between pattern and non-pattern classes. We do not find the Bieman *et al.* result surprising; in fact, we expect that design patterns would be deliberately used to make certain classes (perhaps playing critical roles) easier to change. By design, as the system evolves, these critical classes might well become the focal point for change. Consequently, a finding that design pattern classes change more than non-pattern classes does *not* imply that using patterns is bad: it is simply a reflection of the fact classes playing certain roles in certain patterns, by design and intention, change more often in response to normal software evolution. This leads naturally to the question of how pattern roles influence change-proneness, as we discuss below.

Aversano *et al.* (Aversano et al., 2007) studied patterns in several open source programs comparing change-proneness and co-change of classes participating in design patterns to an overall rate of change. They found that patterns change more frequently when they play a crucial role in the application. They also found that patterns make clients resistant to changes but that changes to the pattern interface reduce this resiliency. They concluded that overall class change frequency and quantity does not depend on pattern type but on the (design) role a pattern plays for the application. Their findings support the work of Bieman *et al.* in that they not only find pattern classes more change-prone, but that, in particular, classes in patterns at the core of an application's design are more change-prone, which, as with the work by Bieman *et al.*, runs counter to expected intuition that design patterns promote stability. This apparent contradiction melts away when pattern roles are considered.

One often ignored aspect of class role membership is that classes frequently play multiple roles. Khomh *et al.* studied the impact on classes playing multiple roles within a system. Their work focuses on cross motif roles and showed that while multiple roles cannot be ignored, a non-significant number of classes play a single role (Khomh et al., 2009). We can draw two key ideas from this work. First, since a statistically significant number of classes play single roles within a system it is reasonable to study the change proneness of patterns playing single roles; second, since we cannot ignore multiple roles, we must include this aspect in our models in some manner. A class playing multiple roles does not necessarily take on the characteristics of just one role and it would be erroneous to treat it as such.

One approach to control for between pattern variation is to consider expected intuition within patterns. Recently Di Penta *et al.* (Di Penta et al., 2008) studied change-proneness of specific roles within design patterns. They compared the change-proneness between groups of classes playing different design pattern roles. For each design pattern, the authors described their expected intuition regarding change-proneness of roles. In the STATE pattern, for example, the STATE role is an interface, and should be more stable. Likewise, for the

ADAPTER, the TARGET role both provides the interface that all adapters must implement as well as the external interface to the pattern via the client and hence should be relatively stable. We observe that their arguments could be abstracted into metapattern roles, in terms of the CONNECTION metapatterns. The TEMPLATE role expects a stable HOOK interface through which it can call on the IMPLEMENTATION classes that extend the HOOK interface. We see additional similarity in the intuition reported for the RECURSIVE CONNECTION based patterns. For both DECORATOR and COMPOSITE, the COMPONENT pattern role is expected to remain stable as it provides the interface through which clients make use of the decorator pattern as well as the interface that must be implemented by all CONCRETE COMPONENTS and DECORATORS. A change to this class may have ripple effects both in the clients that use the pattern and the classes that implement the pattern. Di Penta *et al.* reported, contrary to their intuition, that the DECORATOR and COMPOSITE roles often change more than expected. The COMPOSITE role implements a mechanism to build collections of CONCRETE COMPONENTS and the DECORATOR similarly contains the mechanism to add any number of CONCRETE DECORATOR implementations around each COMPONENT. Hence, these roles are more than simple interfaces and we expect that their TEMPLATE role nature explains previously reported observations. Since these classes implement the elements that comprise the core algorithm of the pattern it is expected that the COMPOSITE would be more change-prone and the CONCRETE COMPONENTS less so.

In summary, while Di Penta *et al.* report distinct results for how the roles in different design patterns are change-prone, their results in our analysis can be more simply and consistently viewed in the light of metapattern roles. In other words, we expect to see in the Di Penta *et al.* data that *Classes playing HOOK roles change less than those playing IMPLEMENTATION or TEMPLATE roles*. Moreover, the reported intuition, although expressed in terms of design pattern roles, is specifically related to the structure modeled by metapatterns.

Our research is partly animated by this finding; we investigate whether design pattern roles that can be grouped into the same *meta-pattern* roles show similar change-proneness. As DiPenta *et al.*, and unlike previous work, we compare roles within patterns. In addition, we control for size, multiple roles, and for between-release effects. This approach allows us to focus more clearly on the affect role membership has on class change proneness.

5 Data Gathering and Methodology

We gathered data from the same 3 projects used by Di Penta *et al.* (Di Penta *et al.*, 2008): JHotDraw (5.1 – 5.4b2), Xerces (1.0.0 – 1.4.4), and Eclipse JDT (1.0 – 2.1.2). We used the same approach to identify the pattern and metapattern instances that survived across multiple changes.

To gauge the effect of design pattern/metapattern roles on change-proneness, we identified pattern instances that remain stable, despite other changes, using the approach of Di Penta *et al.* A pattern instance is *stable* if its major roles are bound to the same class names in two consecutive releases. For metapatterns we require the TEMPLATE and HOOK and their associated template and hook methods to remain stable across two releases.

Given a class playing a stable pattern role between two releases R and $R + 1$, we count the number of changes to that class in a series of snapshots between the releases. We identify transactions: multiple commits within a narrow time window made by the same developer are counted as an atomic commit, and thus a single change.

Previous results counted each unique commit as a single change. We identified change counts in a similar way, but used Fluri's *Change Distiller* (Fluri *et al.*, 2007), a more fine-grained change analysis tool. Commits that contain no actual source code change, e.g. adding blank lines or rearranging methods in a class, are not counted.

Pattern	Key Roles
Abstract Factory	Abstract Factory, Abstract Product
Adapter	Target, Adapter, Adaptee
Command	Invoker, Command, Client
Composite	Composite, Component
Decorator	Component, Decorator
Observer	Subject, Observer
Prototype	Prototype
State/Strategy	State/Strategy, Context
Template Method	Abstract Class
Visitor	Visitor, Element

Table 2: Key roles for identification of duplicate pattern motifs across releases.

5.1 Design Pattern Detection

We use the DeMIMA approach of Di Penta *et al.* to identify occurrences of design patterns (Guéhéneuc and Antoniol, 2007). DeMIMA is built on the `Ptidej` toolset which constructs static and dynamic models of Java source code.

Each pattern is defined as a set of logical constraints; an explanation based constraint solver (Jussien and Barichard, 2000) is used to identify pattern occurrences. We used pattern participation data generated by the `Ptidej` solver provided by Guéhéneuc *et al.* to identify design patterns in JHotDraw, Xerces, and Eclipse JDT.

5.2 Metapattern Detection

We identify metapatterns in two ways. First, we link design patterns with their associated metapatterns, thus extracting metapattern instances from the the DeMIMA pattern data. We call these metapatterns *embedded* metapatterns. Thus, each design pattern role is associated with its metapattern roles. The number of metapattern roles in each design pattern varies and metapatterns may also span multiple design patterns. We consider only intra pattern metapatterns and do not classify any role as a meta-role unless we can identify all metapattern roles within the design pattern. These canonical metapatterns are derived from the example pattern motif structures presented in the GOF book (Gamma *et al.*, 1995) and are presented in recent work on design pattern detection (Hayashi *et al.*, 2008). As an example, the `OBSERVER` pattern has a canonical metapattern role where the (`SUBJECT`, `OBSERVER`, and `CONCRETE OBSERVER`) roles correspond to (`TEMPLATE`, `HOOK`, and `IMPLEMENTATION`) meta-roles respectively. The (`SUBJECT`, `CONCRETE SUBJECT`) often correspond to (`HOOK`, `IMPLEMENTATION`) meta-roles with respect to some `TEMPLATE` (often referred to as the client) that lies outside of the `OBSERVER` pattern. From this perspective the `CONCRETE SUBJECT` role plays no meta-role within the `OBSERVER` design pattern even though it often plays some meta-role within some metapattern that is partially external to the `OBSERVER` design pattern. Since we cannot identify all meta-roles of this *partial metapattern* within the `OBSERVER` pattern, we cannot label it as a stable metapattern, and hence, we do not include it in our analysis. Thus we focus on the metapatterns that model design pattern behavior within the design patterns detected by DeMIMA. This limitation only applies to our analysis of embedded metapatterns inferred from the DeMIMA data. It does not apply to the analysis of DeMIMA design patterns which include all roles detected by DeMIMA, nor does it apply to metapatterns extracted directly from code, as discussed below.

Our second approach to identifying metapatterns relies on our metapattern detection tool `THEX`, an abstract interpretation technique using the `ASM` library (Bruneton et al., 2002) that directly identifies `TEMPLATE/HOOK` relations in Java class files. This approach does not make assumptions about which metapatterns are associated with design patterns; nor does it distinguish between intra and inter pattern metapatterns. If a set of classes meets the requirements for a stable metapattern relationship, they are included in the results. This analysis is not possible with the `dimema` metapatterns because `DeMIMA` does not identify all of the classes that would comprise the inter-pattern metapatterns.

5.3 `THEX` Description

Here we present some details of how we define metapatterns in Java such that they can be detected by `THEX`. We begin with Tourwé’s (Tourwé and Mens, 2003) formal definition of `TEMPLATE/HOOK` relationships, and metapatterns. `THEX` identifies methods as a `HOOK` via inheritance and over-riding.

Given a class M and a class or interface H such that either M is a subclass of H or M implements H , a method $h \in H$ is a potential hook if there exists a method $m \in M$ such that m and h share a common signature and h is `protected` or `public`. In other words there must exist at least one implementation m of a hook h in order to classify h as a hook method.

Then, we find `TEMPLATE` methods t in class or interface T such that t invokes h through some variable v of type H . We take the definition of v from Tourwé and allow it to be either a field, a parameter, or a local variable. `THEX` then performs an intra-procedural data flow analysis using abstract interpretation to trace the type of sources used for method invocations. If the type of the source trace is a subtype of H then we consider the triplet (T, H, M) a basic metapattern with no specific metapattern type. `THEX` uses some minimal heuristics to identify certain types of composition references in arrays and Java Collections. Heuristics are also used to identify hook variable getters. If the call from t to h is made through some method $f \in T$ such that f simply returns the value of a local field y of type H , then we identify y as the hook variable. Once `HOOK` and `TEMPLATE` instances are identified, we use inheritance relationships as described above to identify metapatterns.

5.4 Pattern Counts

The number of pattern and metapattern motifs of each type are identified in Table 3. For each of these patterns we count only motifs that are stable over at least two releases. Whenever the same classes (identified by the same name) play the same key roles, as identified in Table 2 by Di Penta *et al.*, in multiple releases, we count the motif only once.

For embedded metapattern instances, we consider two instances identical if the design pattern instances that they are derived from are identical across two releases as identified above. Thus the embedded metapattern instances are really a grouping of design pattern instances based on the expected underlying metapattern. The `THEX` metapatterns are not associated with any design patterns so we use the `TEMPLATE` and `HOOK` roles as a basis for identification. Two `THEX` metapattern instances that have the same classes in `TEMPLATE` and `HOOK` roles across two releases are counted only once in the table.

The `VISITOR`, `PROTOTYPE`, `DECORATOR`, and `COMPOSITE` patterns are found quite infrequently in several projects; they are excluded from later analysis when there are too few for analysis. The large number of `ABSTRACT FACTORY` and `FACTORY METHOD` instances are

Pattern	JHotDraw				Xerces				ExclipseJDT			
	#	1	2	N	#	1	2	N	#	1	2	N
Design Patterns												
Adapter	118	215	187	36	135	230	224	35	419	1516	1893	316
Abstract Factory	1259	128	171	148	2657	163	323	317	0	NA	NA	NA
Command	50	183	106	49	88	234	167	95	285	1414	865	943
Composite	72	317	57	2	13	92	5	0	92	825	96	0
Decorator	22	220	22	3	0	NA	NA	NA	253	1101	395	45
Factory Method	444	204	205	12	596	56	26	2	3285	1570	1232	121
Observer	69	86	143	33	64	292	49	36	208	332	299	100
Prototype	2	44	0	0	0	NA	NA	NA	5	111	0	3
State/Strategy	46	276	37	7	95	634	114	21	317	2279	986	83
Template Method	63	184	241	50	90	270	357	55	470	1354	2433	396
Visitor	1	21	8	0	6	59	46	14	43	503	327	298
Ptidej Metapatterns												
Connection	236	272	519	75	307	848	553	103	1250	2004	3472	649
Recursive Connection	104	510	152	2	13	123	6	0	373	2103	686	44
Unification	64	500	87	0	100	733	132	0	500	4220	804	0
THEX Metapatterns												
Connection	339	380	319	6	289	486	203	13	3324	3484	2286	153
Recursive Connection	34	316	51	0	47	291	49	1	140	1025	340	14
Unification	30	270	21	0	25	230	15	0	188	2578	226	0

Table 3: Role and pattern cardinality counts *Counts for classes playing 1,2,N (> 2) roles within the specified design pattern. NA indicates that there were no patterns of that type in the project.*

most likely due to the simplistic nature of the patterns themselves. Although we derive the embedded metapatterns from the DeMIMA design pattern results, we do not see the same high numbers mirrored in the associated metapattern motif count. This is because we filter metapattern instances based on the `TEMPLATE` and `HOOK` roles as described above.

5.5 Methodology

We use these data in multiple regression models to examine the relationships between pattern and metapattern role, size, and change-proneness. This common modeling technique has been used, (*e.g.*) to study the effect of coordination on bug resolution times (Cataldo et al., 2006) and the effect of distributed development on defects (Bird et al., 2009).

In our context, we use pattern roles played by classes, project release, and the size of classes in lines of code (LOC) as *explanatory*, or *independent* variables and the number of distinct commits to a class as the *response*, or *dependent* variable. We use Understand for Java 1.4 by SciTools to compute LOC, which is taken to be the number of source lines that contain actual source code (Sci-Tools). For every design pattern, we build several linear models using roles as explanatory variables. There is one tuple per revision per class for each project within a system. Models are built on a subset of these tuples based on pattern participation. For the majority of models our response variable is the number of commits made to the class prior to the release. For the models where size *is* the response variable, it is measured in LOC and log transformed as described in the next section where we discuss explanatory and response variables in greater detail.

Our interpretation is based on two key attributes of these models. First, the estimate of the *coefficient* for a particular role indicates the difference in change-proneness relative to

the weighted mean change proneness of all classes in the same pattern. Both the sign and the magnitude indicate the relative effect. This effect is not meaningful if not statistically significant. We consider these role coefficients both with and without size control. We always include release as a control and its interpretation is not significant in this study, we discuss it briefly in the next section and again in threats to validity. Second, the R^2 value represents the explanatory power of the model, indicating the proportion of variance in the response variable that can be explained by variance in the explanatory variables.

Since we are interested in the difference in change-proneness between roles in the same pattern, we include only classes that play at least one role in one instance of the specified pattern. As Khomh et al. (2009) *et al.*, we observed many cases where a class played multiple roles. For example, a class may be a `COMPONENT` in one instance of the `COMPOSITE` pattern and a `LEAF` in another. To compare the change-proneness of a class playing a particular role to the mean change of classes within the pattern, we have to code role membership as a mutually exclusive dummy variable. Although a non-mutually exclusive coding is possible, it complicates interpretation of the coefficients. One choice would be to encode all possible combinations of roles. Unfortunately, this encoding leads to many highly correlated and insignificant predictor variables, even, when we build models for only a single pattern. The explosion of insignificant and correlated role combinations over multiple patterns precludes building a useful model over the entire system. Since a significant number of classes play only one role, we elect to use a binary predictor for these roles; we also include a binary predictor for classes playing two distinct roles, and a predictor for classes with n distinct roles (for $n > 2$). We are not concerned with a class that plays multiple instances of the same role. We consider a class that plays only the `HOOK` role in 10 different instances, for example, as playing a single role.

Class	TEMPLATE	HOOK	IMPLEMENTATION	HOOK <i>code</i>	IMPLEMENTATION <i>code</i>
C_1	1	0	0	-2/4	-1/4
C_2	0	1	0	1	0
C_3	0	0	1	0	1
C_4	1	0	0	-2/4	-1/4
C_5	0	1	0	1	0
C_6	1	0	0	-2/4	-1/4
C_7	1	0	0	-2/4	-1/4

Table 4: *Weighted effects coding example.* TEMPLATE is base, encoded as $-n_k/n_{base}$, and fractions are left unreduced for clarity.

Since the role memberships are unbalanced, i.e., there are typically more `IMPLEMENTATION` classes than `HOOK` classes, we use a weighted effects coding that facilitates a straightforward interpretation of the coefficients (Cohen et al., 1983). Each role membership coefficient indicates the relative effect of role membership on the dependent variable, compared to the weighted mean of the dependent variable across all classes participating in a pattern. For example, in the case where change-proneness is the dependent variable, a positive coefficient indicates that, on average, the class playing the role in question changes more often and a negative coefficient indicates that the it changes, on average, less often.

The role predictors (`COMPONENT`, etc.) take on a value of 1 if the class plays that role and 0 otherwise except as follows. For each model, one role is selected as the *base* role for the coding scheme and is coded as $-n_k/n_{base}$ where n_{base} is the count of classes playing the base role and n_k is the count of classes playing role k . For example, to code the roles for the `CONNECTION` metapattern suppose we use a simple coding where 1 indicates that the

class plays a role and 0 indicates that it does not. This coding is shown in the second through fourth columns of the Table 4. We might choose `TEMPLATE` to be the base role and recode as shown in the two rightmost columns. It is necessary that the base coded role is left out of the model in order to avoid perfect collinearity. Since the model now includes only predictor variables for the `HOOK` and `IMPLEMENTATION` classes, the model does not yield a coefficient for the `TEMPLATE` classes. To obtain this coefficient, each model is then repeated with a different base variable. This method produces the same coefficients for the non-base predictors and is the preferred method for obtaining coefficients for all coded predictors (Cohen et al., 1983).

Table 3 shows for each project and each pattern how many classes play a single role, two distinct roles, or more than two distinct roles within the same pattern. A class may play, for example, a `CONCRETE STRATEGY` role in one instance while also playing the `CLIENT` role for a second state/strategy pattern. As DiPenta *et al.*, we do not include cross pattern participation in this study as we are looking at relative change-proneness of classes playing roles within patterns. That is, we do not exclude classes playing roles across multiple patterns, however, we do not control for their multiple pattern membership either. We do consider within pattern role multiplicity, i.e. classes which play multiple roles within the same pattern, as it is not reasonable to arbitrarily assign role membership to a class when it plays multiple roles within the same pattern. For example, if a class plays a `HOOK` role in 3 patterns and a `IMPLEMENTATION` role in 8 more, we cannot choose either role arbitrarily to represent the role participation for the class.

It has been shown that size often follows a log-normal distribution (Zhang and Tan, 2007). We observe this distribution in all three projects and so we log transform `LOC` to increase central tendency, reduce heteroskedasticity, and improve the model fit. In addition, due to skewness, we also log transform the number of changes (Cohen et al., 1983).

There are potentially many reasons that a class may change including defect corrections, foreseen design changes, unforeseen design changes, etc. In addition, there is no reason to expect that the relative frequency of these changes is consistent across all releases. We might, for example, expect significant between release variation as a project moves from a development phase into a maintenance phase. In order to include all releases in the models while controlling for between release variation we treat release as a time fixed effect so that we are only looking at within-release, within-pattern variation of change-proneness (Brooks, 2008). The between release variance is captured by the coefficient of the release control variable. We do not include any other within-release, within-pattern control variables, such as change type, in this study (See 8).

To check that multicollinearity isn't a problem, we compute the variance inflation factor (VIF) of each dependent variable in all of the models. Although there is no particular value of VIF that is considered excessive, we use the commonly used conservative value of 5 (Cohen et al., 1983). We did not find any VIF values that exceeded 5 in any of the models and conclude that multicollinearity is not an issue.

We check for outliers through visual examination of the residuals vs. leverage plot for each model looking for both separation and large values of Cook's distance. Because of the large number of models, we automatically eliminate points with Cook's distance greater than the critical value of the F distribution with $\alpha = 0.5$, and $k + 1$, $n - k - 1$ degrees of freedom, where k is the number of independent variables in the model and n is the number of data points (Cohen et al., 1983).

Because we want to compare nested models (i.e., *a model which uses the same response variable, a subset of predictor variables, and the same set of data points*) to demonstrate the explanatory power of additional variables, we must ensure that automatic variable elimination results in the same data points for each model. Since Cook's distance is dependent on the choice of predictors, it varies across models. We find that by using the model $M_{r,v}$,

Pattern	TEMPLATE	HOOK	IMPLEMENTATION
Abstract Factory	CLIENT	FACTORY	CONCRETE FACTORY
(<i>element tree</i>)	CLIENT	PRODUCT	CONCRETE PRODUCT
Adapter	CLIENT	TARGET	ADAPTER
Command	INVOKER	COMMAND	CONCRETE COMMAND
Composite	COMPOSITE	COMPONENT	LEAF
Decorator	DECORATOR	COMPONENT	CONCRETE COMPONENT
Factory Method	CREATOR	CREATOR	CONCRETE CREATOR
Observer	SUBJECT	OBSERVER	CONCRETE OBSERVER
Prototype	CLIENT	PROTOTYPE	CONCRETE PROTOTYPE
State/Strategy	CONTEXT	STATE	CONCRETE STATE
Template Method	ABSTRACT CLASS	ABSTRACT CLASS	CONCRETE CLASS
Visitor	CLIENT	VISITOR	CONCRETE VISITOR
(<i>product tree</i>)	OBJECT STRUCTURE	ELEMENT	CONCRETE ELEMENT

Table 5: Design Pattern to Metapattern role mapping. (Remaining roles in order: Adapter:adaptee, Command:client, receiver Decorator:concrete decorator, Factory Method:product, Observer:concrete subject)

which uses roles and release as predictors, as a basis, models with additional predictors do not eliminate any additional data points.

We compute models for three categories of data in each of the three projects JHotDraw, Xerces, and Eclipse JDT. We first examine design patterns as identified by DeMIMA. We also examine metapatterns as extracted from the DeMIMA design patterns following the description presented in section 3. This set of metapatterns serves to group the design patterns by their expected underlying metapatterns. Finally, we analyze metapatterns identified by THEX presented earlier. The THEX metapatterns look at all found metapatterns independent of design patterns. By looking at the first set we can see if our expectations about metapatterns behavior hold when applied to known design patterns while the second set allows us to see if these same behaviors hold when we relax the detection requirements to only those required by metapatterns.

There are over 100 combinations of projects, categories of patterns, and particular patterns, making a detailed account of each model tedious and impractical (see summary in Tables 6,7, and 8). In lieu of this we provide a detailed summary of role behavior and an overall summary of model behavior across each project.

6 Results

Here we present the results ordered from RQ1 to RQ4. First, we discuss the degree to which pattern roles explain variance in change-proneness, when considering the impact of size. Second, we discuss the influence of metapattern roles on change-proneness, when controlling for size. Third, we discuss whether pattern roles provide additional power, as compared to metapattern roles, in explaining change-proneness (when controlling for size). Finally, we present data indicating that design pattern and metapattern roles are associated with size to a significant extent.

For the following discussion we will be referencing Tables 6,7, and 8. There is one table for each project. The rows in the table present the change-proneness results for different design patterns. The columns present different models. There are 4 groups of columns, starting with M_v . The major columns correspond to models that control for different effects (version

Pattern	M_v	M_{vr}						M_{vrs}						M_{vs}	$M_{vr \rightarrow s}$								
		T	H	M	rm	2	N	R^2	T	H	M	rm	2		N	R^2	T	H	M	rm	2	N	R^2
Design Patterns																							
Adapter	0.46	.	↓	.	.	.	↑	0.48	↓	↑	0.57	0.57	↑	↓	↓	.	↑	↑	0.36
Abstract Factory	0.54	.	↓	↑	.	.	↑	0.56	.	.	↑	.	.	.	0.62	0.62	↑	↓	↓	.	↑	↑	0.33
(product tree)		.	↓	↓	↓	↓	↓	.	.	
Command	0.50	.	.	↓	↓	.	↑	0.55	.	.	↓	.	.	↑	0.60	0.58	.	↓	↓	↓	↑	↑	0.25
Composite	0.34	.	↓	↓	.	.	↑	0.43	.	.	↓	↓	.	↑	0.50	0.47	.	↓	↓	↓	↑	↑	0.40
Decorator	0.48	↑	0.48	↑	0.57	0.57	↑	↓	↓	↓	↑	↑	0.23
Factory Method	0.48	.	.	↓	↓	.	↑	0.52	.	.	↓	.	.	↑	0.59	0.57	↓	↑	↑	.	↓	↑	0.13
Observer	0.48	.	↓	.	↑	.	↑	0.53	↑	↓	0.58	0.56	.	↓	↓	.	↑	↑	0.39
Prototype	NS	NS	NS	NS	NS
State/Strategy	0.40	.	↓	.	.	.	↑	0.44	0.51	0.51	↑	↓	.	.	↑	↑	0.41
Template Method	0.45	↓	↓	↓	.	.	↑	0.48	↓	↑	0.57	0.56	.	↓	↓	.	↑	↑	0.29
Visitor	NS	NS	NS	NS	NS
Embedded Metapatterns																							
Connection	0.37	.	↓	↓	.	.	↑	0.38	↑	↑	.	.	↓	.	0.46	0.45	.	↓	↓	.	↑	↑	0.36
Rec Connection	0.35	.	↓	↓	.	.	↑	0.37	.	↑	↓	.	.	.	0.47	0.45	.	↓	↓	↑	.	↑	0.34
Unification	0.37	↓	↑	0.38	↑	↑	.	.	↑	.	0.46	0.45	↓	.	.	.	↓	↑	0.34
THEX Metapatterns																							
Connection	0.37	.	↓	.	.	.	↑	0.38	.	↑	.	.	↓	.	0.49	0.47	↑	↓	↑	.	↑	↑	0.42
Rec Connection	0.29	.	.	↓	.	.	↑	0.33	.	↑	↓	.	.	.	0.46	0.44	↑	↓	.	.	↑	.	0.29
Unification	0.37	↑	↑	0.43	↑	0.50	0.50	↑	.	.	.	↑	.	0.17

Table 6: JHotDraw In model labels: v =release, r =role, s =size. Change proneness is response for all except $M_{vr \rightarrow s}$ where size is response. Read patterns across the rows and roles and models down the columns. (see individual model descriptions in text) Role labels (T/H/M/2/N/rm) indicate the metapattern role represented by the column. For design patterns see Table 5 for the specific design pattern role which maps to the metapattern label. 2 = 2 roles, N = more than 2 roles, rm = remaining roles. Role indicators ($\uparrow, \uparrow, \uparrow$) indicate positive significance at (0.01, 0.05, 0.10) levels respectively, ($\downarrow, \downarrow, \downarrow$) indicate negative significance at (0.01, 0.05, 0.10) levels respectively, and (·) indicates not significant. (Remaining roles (rm) in order: Adapter:adaptee, Command:client, receiver Decorator:concrete decorator, Factory Method:product, Observer:concrete subject) Unification patterns indicate template/hook role in template column. Patterns with two rows have more than one set of metapattern roles and numeric results are the same for both rows.

M_v , version+roles M_{rv} , version+roles+size M_{vrs} etc.) and are described in detail below. Under the major columns, the first three sub-columns show the direction and significance for the TEMPLATE, HOOK, and IMPLEMENTATION roles respectively. The next column shows the significance and direction of the remaining pattern roles that are not related to metapattern roles. The column labeled 2 represents the effects code for a class that is playing 2 distinct roles and the column labeled N indicates the effect when a class plays more than 2 distinct roles. Finally, the column labeled R^2 gives the variance explained by that particular model.

Each row and major column represents a specific model and within each major column (e.g., M_{vrs}), each sub-column represents a variable. The cell corresponding to row and column gives the significance of the variable. Due to the large number of models and the need to present the results somewhat succinctly we use symbols to indicate both the strength and direction of each coded predictor. We visually code the direction and significance of the p-values representing the role coefficient t-tests as explained in the caption of the figures. For each model we adjusted model p-values for multiple hypothesis testing using the Benjamini-Hochberg method (Benjamini and Hochberg, 1995). Models that were not statistically significant ($p > 0.05$) are indicated with NS. Entries that contain NA had an

Pattern	M_v	M_{vr}						M_{vrs}						M_{vs}	$M_{vr \rightarrow s}$									
		T	H	M	rm	2	N	R^2	T	H	M	rm	2		N	R^2	R^2	T	H	M	rm	2	N	R^2
Design Patterns																								
Adapter	0.11	↑	↓	·	·	·	0.15	·	·	·	·	·	·	·	0.32	0.32	↑	↓	↑	·	·	·	0.30	
Abstract Factory	0.11	↓	↓	·	·	↓	0.25	·	·	·	·	↓	↑	0.31	0.30	↓	↓	·	·	·	·	↓	↑	0.43
(product tree)		↓	↓	·	·	·																		
Command	0.11	·	↓	↓	↓	·	0.25	·	↓	↓	↓	·	↑	0.32	0.28	·	↓	↓	↓	↓	↑	↑	0.35	
Composite	0.20	↑	·	·	·	·	0.29	↑	·	↓	·	·	·	0.50	0.47	·	·	·	·	·	·	·	·	
Decorator	NA						NA							NA	NA								NA	
Factory Method	NA						NA							NA	NA								NA	
Observer	0.40	·	↓	↓	·	·	0.47	·	·	·	↓	·	↑	0.53	0.51	↓	↓	·	·	·	↑	↑	0.43	
Prototype	NA						NA							NA	NA								NA	
State/Strategy	0.17	↑	↓	↓	·	↑	0.34	·	·	↓	·	↑	↑	0.43	0.41	↑	↓	↓	·	↑	↑	↑	0.30	
Template Method	0.15	↑	↓	↓	·	↑	0.23	·	·	↓	·	↑	·	0.37	0.36	↑	↓	↓	·	↑	↑	·	0.19	
Visitor	NS	·	·	·	·	·	NS	·	·	·	·	·	·	NS	NS	·	·	·	·	·	·	·	NS	
Embedded Metapatterns																								
Connection	0.22	·	↓	↓	·	↑	0.28	·	·	↓	·	·	↑	0.35	0.35	↓	↓	↓	·	↑	↑	0.23		
Rec Connection																								
Unification	0.24	↓	·	·	·	·	0.26	·	·	·	·	↑	·	0.29	0.29	↑	↓	·	·	·	·	·	0.20	
THEX Metapatterns																								
Connection	0.28	·	↓	·	·	↑	0.35	↓	·	·	·	·	↑	0.39	0.37	↑	↓	·	·	↑	↑	0.49		
Rec Connection	0.31	↓	↓	·	·	↑	0.41	↓	·	·	·	·	·	0.46	0.42	↑	↓	↑	·	·	·	·	0.38	
Unification	0.38	·	↓	·	·	↑	0.46	↓	·	·	·	↑	·	0.56	0.54	↑	↓	·	·	↑	↑	·	0.20	

Table 7: Xerces-J In model labels: v =release, r =role, s =size. Change proneness is response for all except $M_{vr \rightarrow s}$ where size is response. Read patterns across the rows and roles and models down the columns. (see individual model descriptions in text) Model indicators (NS,NA) indicate that the model was either not significant, or not applicable if there were no patterns of the indicated type. Role labels (T/H/M/2/N/rm) indicate the metapattern role represented by the column. For design patterns see Table 5 for the specific design pattern role which maps to the metapattern label. 2 = 2 roles, N = more than 2 roles, rm = remaining roles. Role indicators (↑, ↑, ↑) indicate positive significance at (0.01, 0.05, 0.10) levels respectively, (↓, ↓, ↓) indicate negative significance at (0.01, 0.05, 0.10) levels respectively, and (·) indicates not significant. (Remaining roles (rm) in order: Adapter:adaptee, Command:client, receiver Decorator:concrete decorator, Factory Method:product, Observer:concrete subject) Unification patterns indicate template/hook role in template column. Patterns with two rows have more than one set of metapattern roles and numeric results are the same for both rows.

insufficient number of classes in one or more roles, or an insufficient number of pattern occurrences, such that the model could not be fit. For each model we consider only the subset of classes that play at least one role in the indicated pattern. We perform multiple hypothesis correction on the coefficient t-tests for role predictors but also report the p-values to three levels as indicated in the legend.

6.1 Models M_v

We first briefly discuss the baseline model for across-release change-proneness, which is shown in the leftmost column headed M_v . This model, which uses only the release identifier as a factor, captures the between-release change-proneness variation. We later use the release predictor variable to control for the between-release variation in other models. For JHotDraw, release alone accounts for significant change-proneness across all patterns. For Xerces, the effect of release is more moderate and for Eclipse JDT, significantly lower. In essence, this simple model captures the degree to which releases are different with respect

Pattern	M_v	M_{vr}					M_{vrs}					M_{vs}	$M_{vr \rightarrow s}$									
		T	H	M	rm	2	N	R^2	T	H	M		rm	2	N	R^2	T	H	M	rm	2	N
Design Patterns																						
Adapter	0.04	.	↓	.	.	↑	.	.	0.08	↓	↑	↓	.	.	↑	0.33	0.33	↑	↓	↑	↓	0.32
Abstract Factory	NA	.	↓	NA	↓	.	↓	.	.	.	↑	NA	NA	↑	↓	.	↓	NA
Command	0.03	.	↓	↓	↓	↑	.	0.15	↓	.	↓	.	.	↑	0.34	0.33	↑	↓	↓	↓	↑	0.31
Composite	0.07	↑	↑	↓	↓	↑	↑	0.10	.	↑	0.44	0.44	↑	↓	↓	↓	↑	0.14
Decorator		↑	↓	↓	↓	↑	↑	0.05	0.33	0.33	↑	↓	↓	↓	↑	0.09
Factory Method	0.02	↓	↑	↓	↓	↑	↑	0.10	.	↑	↓	.	.	↑	0.36	0.35	↓	↑	↑	.	↓	0.19
Observer	0.04	↓	↓	↑	↓	.	.	0.16	↓	↓	0.41	0.40	↓	↓	↓	↑	.	0.44
Prototype	NS	NS	NS	NS	NS
State/Strategy	0.03	↑	↓	↓	↓	↑	↑	0.14	↓	↑	↓	.	.	↑	0.35	0.34	↑	↑	↓	↓	↑	0.27
Template Method	0.04	↑	↓	↓	↓	↑	↑	0.07	.	↑	↓	.	.	↑	0.33	0.33	↑	↑	↓	↓	↑	0.21
Visitor	0.03	↓	↓	↓	.	↑	↑	0.10	↑	0.34	0.33	↑	↓	↓	.	↑	0.20
(element tree)		↓	↓	↓	.	↑	↑		↓	↑	↑	.	.	↑			↑	↓	↓	.	↑	
Embedded Metapatterns																						
Connection	0.05	↑	↓	↓	.	↑	↑	0.08	↓	↑	.	↓	↑	0.32	0.32	↑	↓	↓	.	↑	↓	0.21
Rec Connection	0.05	↑	↓	↓	.	↑	↑	0.09	.	.	↓	.	↑	0.37	0.37	↑	↓	↓	.	↑	↑	0.11
Unification	0.05	↓	.	.	.	↓	↓	0.09	↑	0.32	0.32	↓	.	.	.	↓	↑	0.23
THEX Metapatterns																						
Connection	0.05	↑	↓	↓	.	↑	↑	0.16	↓	↑	↓	.	↑	0.32	0.30	↑	↓	↓	.	↑	↑	0.45
Rec Connection	0.05	.	↓	↓	.	↑	↑	0.12	↓	.	↓	.	↑	0.28	0.27	↑	↓	↓	.	↑	↑	0.18
Unification	0.05	↑	.	.	.	↑	↑	0.07	↓	↓	.	.	↑	0.30	0.30	↑	.	.	.	↑	↑	0.08

Table 8: EclipseJDT In model labels: v =release, r =role, s =size. Change proneness is response for all except $M_{vr \rightarrow s}$ where size is response. Read patterns across the rows and roles and models down the columns. (see individual model descriptions in text) Model indicators (NS,NA) indicate that the model was either not significant, or not applicable if there were no patterns of the indicated type. Role labels (T/H/M/2/N/rm) indicate the metapattern role represented by the column. For design patterns see Table 5 for the specific design pattern role which maps to the metapattern label. 2 = 2 roles, N = more than 2 roles, rm = remaining roles. Role indicators ($\uparrow, \uparrow, \uparrow$) indicate positive significance at (0.01, 0.05, 0.10) levels respectively, ($\downarrow, \downarrow, \downarrow$) indicate negative significance at (0.01, 0.05, 0.10) levels respectively, and (·) indicates not significant. (Remaining roles (rm) in order: Adapter:adaptee, Command:client, receiver Decorator:concrete decorator, Factory Method:product, Observer:concrete subject) Unification patterns indicate template/hook role in template column. Patterns with two rows have more than one set of metapattern roles and numeric results are the same for both rows.

to change-proneness. For the purposes of this work, however, release is a control and we are not interpreting the coefficients other than to observe that its effect on change proneness is project dependent.

6.2 Models M_{vr}

The models in the second major column labeled M_{vr} are most similar to results by DiPenta *et al.*. These models show the relationship between roles and change-proneness when not controlling for size. From this we can see that almost without exception, whenever classes playing the hook role are significant, they are, as expected by reported intuition, less change-prone. For JHotDraw some roles are significant, for Xerces about one half are significant and for Eclipse JDT most of the roles are significant.

Looking at the results for Eclipse JDT, the roles reflect previously reported intuition for many of the patterns. For example, the the STATE/STRATEGY pattern results indicate that the STATE changes less often while the CONTEXT changes more often than the mean change

proneness for all classes participating in any occurrence of the STATE/STRATEGY pattern. For the ADAPTER pattern we see that the TARGET role is less change-prone, however, our results do not indicate that the CLIENT and ADAPTER roles change less often.

The difference from earlier results might be due to some difference in modeling. DiPenta *et al.* use a 2 sample test, that didn't consider differences between releases (Di Penta *et al.*, 2008). Another possibility is our choice of role coding. We are not comparing the roles directly with each other, rather, we are comparing each role with the mean change-proneness of all classes participating in the pattern. Nonetheless, the results in this table largely agree with earlier work, and with intuition. They indicate that, when we do not control for size, classes playing the HOOK role tend to change less often. For classes playing 2 distinct roles the results vary, but classes playing 3 or more distinct roles (in the same design pattern) are more change-prone across almost all patterns in all projects.

6.3 RQ1 and RQ2

In their study of the effect of design pattern role on change-proneness, Di Penta *et al.* (Di Penta *et al.*, 2008) did not examine the effect of size. We examine the effect of design pattern role on change-proneness when controlling for size by adding lines of code (LOC) as an additional predictor to the model. The results for these models are reported in the column labeled M_{vrs} . We do not report detailed coefficients for size as a predictor, but its effect can be easily summarized. Size is statistically significant and positively related to change proneness in every pattern in every project.

The models in the next column M_{vs} are also important to the following discussion. We note first that the R^2 values for the models including size are larger in every case than the models using only roles as predictors. Adding even a random variable to a model can increase the R^2 value, so we use adjusted R^2 values that show an increase only if the additional terms improve the model more than would be expected by chance (Toutenburg, 2002). Comparing M_{vr} and M_{vrs} directly, however, may be misleading as many predictors lose significance when including size as a predictor. Since size remains significant in every model whether we include roles or not, we can reasonably compare the R^2 values for M_{vs} , and M_{vrs} . Turning our attention to M_{vs} we can see that the reported R^2 values are only slightly smaller than the R^2 values for the M_{vrs} models. Adding roles to M_{vs} only increases the explanatory power of the model by a few percent. ANOVA tables of each model and model pair shows that either size or release is the dominant predictor in every model and every pattern, and, that adding roles improves each model only minimally. We note that release is only dominant for some models in JHotDraw and that when release is dominant, size is always the second most significant predictor. This suggests that the simpler models, which do not include roles, are the more parsimonious models. We must also consider, however, the effect of the statistically significant roles.

When size is included as a control, many roles lose their significance. This effect is especially true with Xerces where very few roles remain significant. Focusing on Eclipse JDT, which contains the largest number of significant roles after controlling for size, we can observe an interesting phenomenon: Although some roles are still significant, their direction is often reversed. This reversal is particularly true with IMPLEMENTATION roles. Most classes playing IMPLEMENTATION roles are less change-prone when we omit size control but are more change-prone after controlling for size. Interfaces that should be stable change more often relative to their size than the classes that implement them.

The effect of participating in two distinct roles varies with project and pattern. When classes participate in three or more distinct roles the effect of that participation is fairly consistently positive. We might expect that classes playing many roles are going to be larger, but,

even after controlling for size, classes that play more than two distinct roles in a particular pattern, *e.g.* a `CONCRETE STRATEGY` role in one instance may also be the `CONTEXT` for another instance, are more change-prone. As Khomh points out, classes playing two distinct roles in the same pattern instance must be participating in a degenerate pattern (Khomh et al., 2009). It is not the case, however, that a class playing two distinct roles in the same pattern but different instances, as in the previous example, must be degenerate. A class playing more than two roles in the same pattern receives change pressure from multiple pattern instances so is likely to change more often. We note, however, that the multiple role predictors are included in the tiny improvement that roles contribute to the overall explanatory power of the model.

With respect to **RQ1**, which asks if roles are significant predictors of change-proneness when controlling for size, the answer is twofold. First, the explanatory power of roles is so small when controlling for size that too few roles remain statistically significant to support the expected intuition for the effect of roles on change-proneness. Second, when roles are significant, their effect often runs counter to reported intuition. That is, `HOOK` classes are often more change prone after controlling for size. When controlling for class size, the role that a class plays in a pattern has little effect on change-proneness unless the class plays three or more distinct roles. Thus, with respect to RQ1 and RQ2:

Conclusion for RQ1 and RQ2: *Size appears to be the strongest factor in explaining change-proneness. Beyond this, pattern and metapattern roles add very little additional explanatory power.*

6.4 RQ3

Research question 3 is similar to RQ2 except that it pertains to the value of metapatterns in evaluating change-proneness. When controlling for size, do metapattern roles explain as much of the differences in change-proneness as well as pattern roles? In light of the answer to the first two research questions, the answer to the third becomes almost moot. However, we did contrast the effect of design pattern roles on change-proneness and metapattern roles when controlling for class size.

In this case, we quantify the effect of pattern role as the the difference in R^2 between the model including only class size, M_{vs} , and the model with class size *and* pattern role, M_{vrs} . In general, design pattern roles do not consistently have a greater effect on the percentage variance in change proneness to classes than metapattern roles. Although the percentage varies slightly it is consistently less than 2% across all patterns in all projects. So although we cannot conclude that metapattern roles offer any quantitative advantage over design pattern roles in evaluating pattern properties, we can observe some qualitative properties.

First, whenever roles are significant after controlling for size we observe a similar trend in their associated design patterns. However, we observe that this trend is not particularly interesting, in short, `HOOK` classes change more often whereas `TEMPLATE` and `IMPLEMENTATION` classes change less often. Second, metapatterns derived from detected design patterns do not explain appreciably more or less than those detected by `THEX`.

We also observe that there exist fewer opportunities to play more than 2 roles with metapatterns as we have at most 3 roles. As a consequence, greater than 2 roles are only significant across metapatterns in Eclipse JDT.

Conclusion for RQ3 *When explaining change-proneness, while controlling for size, there is very limited additional explanatory power provided by pattern or metapattern roles. Furthermore, there is very little difference in the additional power provided by metapattern roles, as compared to the explanatory power of pattern roles. Metapattern roles do, however, follow a similar pattern of change-proneness and size as their design pattern cousins.*

6.5 RQ4

We now turn to RQ4, *viz.*, the relationship of pattern and metapattern roles with size. Note that we use just the roles and release factor as predictors in the model: size is now the response variable. As before, weighted effects codes for the roles indicate the relative effect on size when a class plays the corresponding role and the chosen symbol indicates the direction and significance of the effect. The overall R^2 indicates how much of the variance is explained by the roles.

The most striking observation regarding size is the strong agreement (both direction and significance) of the role coefficients with the corresponding role coefficients in M_{vr} . With only a few exceptions, whenever a role is significant in both models the direction of significance coincides. In most models, for example, `HOOK` classes are predicted to be less change-prone, and similarly, in most models, `HOOK` classes are predicted to be smaller.

The previous models have shown that size is the dominant feature in change-proneness; these models complete the picture by demonstrating that the change-proneness relationships between roles are largely a function of the size of the roles. This model also explains, in part, why roles lose significance in the M_{vrs} models. Roles are, in fact, strongly correlated with size. That is, they are in part explaining the same phenomena; thus when we include size in the model, many roles become insignificant. From these results we might conclude that roles affect size in at least two ways: 1) directly, albeit minimally, and 2) indirectly through size.

Therefore, roles, in fact, do partially influence change-proneness. However, the amount of indirect change is related to the product of the variances through the mediation variable *size*. A significant portion of the size change-proneness relationship will not be accounted for by roles through this indirect path.

We summarize many of the intuitions about design patterns previously reported by observing that they are common across metapattern roles. Hence, based on the intuitions reported by Di Penta *et al.* and our summary from § 3 we would expect that classes that play the `HOOK` role will be smaller, and consequently less change-prone, than classes that play the `IMPLEMENTATION`, and (sometimes) `TEMPLATE` roles.

Conclusion for RQ4 *Pattern and metapattern roles have a significant relationship with size*
Summary: Our summary from the results presented above is that design pattern roles (and meta-pattern roles) explain very little about change-proneness of classes, beyond what is explained by size.

7 Threats to Validity

Perry *et al.* (Perry et al., 2000) identify three forms of validity that must be addressed in research studies. We now examine threats to each form of validity in our study and the methods used to mitigate these threats where possible.

The main threats to *construct validity* come from identification of patterns and our method of quantifying change-proneness and size. Perfect, automatic identification of design pattern instances in software remains unsolved. However, vast strides have been made, and we use data produced by a state-of-the-art detector (Guéhéneuc and Antoniol, 2007).

Guéhéneuc and Antoniol report recall and precision: however recall is reported as 100% and precision ranging from 34% to 80%. This wide range of precision limits the accuracy of even state of the art approaches to pattern detection. Although the reported perfect recall implies that all patterns are measured, the low precision for some patterns insures that any measurement will include many non-pattern classes. This tool has been used in prior studies

examining properties of design patterns (Di Penta et al., 2008). Metapatterns are defined in purely structural terms and are much easier to detect using static analysis techniques.

We evaluated our tool, `THEX`, on several code-bases manually examining many identified instances of design patterns and metapatterns to verify their accuracy ?. We measure *size* using lines of code (LOC), which is an accepted standard of code size (Rosenberg, 1997). We measure *change-proneness* of a class by counting the number of distinct commits to that class between releases. However, we also performed this analysis by using number of semantic changes as described in Fluri *et al.* (Fluri et al., 2007) with similar results.

Internal validity is the ability of a study to establish a causal link between independent and dependent variables regardless of what the variables are believed to represent. Our study is focused on examining a link between pattern role and change proneness. In our analysis, we limit our examination to changes made to stable pattern instances (instances that exist in at least two consecutive releases) and only count changes that occur *after* the instance comes into existence. The use of this criteria mitigates threats to internal validity, but does not completely remove them as we are not counting pattern roles that exist for only a single release. We control for release dependent effects on change-proneness by including release as a factor. This has an aggregation effect over each project and assumes an essentially static relationship between the change proneness of classes playing pattern roles.

In terms of model validity we assume that there exist release specific effects that affect all classes within a release in the same manner. This assumption limits our analysis to a change in intercept for role membership effects. The treatment of release as a time fixed effect captures the time dependent variance in a single control variable and is comparable to previous approaches that seek to capture differences in role or pattern behavior as a scalar measured over the life of the pattern or project.

Lastly, *external validity* refers to how these results generalize. Our study includes multiple projects and we find similar results among all projects studied. However, the sample size is only three projects, all of which, are written in the Java programming language. While this gives evidence of the ability of our results to generalize, further study on more projects and different languages will increase our confidence in these findings as answers to the research questions on a broad level.

8 Conclusion

Our analysis of earlier papers on change-proneness of design pattern roles suggested that the differences in change-proneness of pattern roles might be explained by the simpler, purely structural metapattern roles. In fact, when we controlled for sizes, both pattern and metapattern roles showed only minor differences. Next, we also noted a significant association between both pattern and metapattern roles and size. We argue therefore, that the previously reported association of pattern roles with change-proneness, might in fact, be due to the size variations in the classes playing the roles.

We believe that metapatterns can be useful in studying design pattern behaviors in software systems. We have shown that some beliefs about the intuition of design patterns can be captured by metapatterns. Metapatterns can yield interpretable results due to their abstract nature and smaller set of role behaviors.

E.B. Swanson identified three categories of maintenance: corrective, adaptive, and perfective (Swanson, 1976). In this study, we do not distinguish between these different maintenance activities, hence, interpretation of our results may be dependent on the underlying distribution of change type. While the strength of this dependence is worthy of consideration, this is left as future work.

Measuring the size of a class is simple, fast, and accurate; finding its pattern roles, (or even its meta-pattern roles) is substantially more difficult and error-prone. Therefore, pragmatically, if one is seeking to get an indication of which classes might be change-prone, our study suggests that it might best to ignore pattern roles altogether, and just use size as an indicator.

However, pattern roles might be known early in the design process, and thus would be an early and useful indicator of the eventual future sizes of classes playing those roles. This expected future size, in turn, is a useful indicator of change-proneness.

9 Acknowledgements

Our thanks to Yann-Gaël Guéhéneuc for allowing the use of the Ptidej toolset and pattern detection data, Beat Fluri for his change distiller tool, and to Sci-Tech corporation for providing academic use of Understand for Java.

References

- L. Aversano, G. Canfora, L. Cerulo, C. D. Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 385–394, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: <http://doi.acm.org/10.1145/1287624.1287680>.
- V. Basili and S. Elbaum. Empirically driven SE research: state of the art and required maturity. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 32–32, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: <http://doi.acm.org/10.1145/1134285.1134291>.
- Y. Benjamini and Y. Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.
- J. Bieman, D. Jain, and H. Yang. OO design patterns, design structure, and program changes: an industrial case study. *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 580–589, 2001.
- J. Bieman, G. Straw, H. Wang, P. Munger, and R. Alexander. Design patterns and change proneness: an examination of five evolving systems. *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 40–49, 2003.
- C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In *Proc. of the International Conference on Software Engineering*, 2009.
- L. Briand and J. Wust. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 56:98–167, 2002.
- C. Brooks. *Introductory econometrics for finance*. Cambridge Univ Pr, 2008.
- E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002.
- M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 353–362, New York, NY, USA, 2006. ACM. ISBN 1-59593-249-6. doi: <http://doi.acm.org/10.1145/1180875.1180929>.

- J. Cohen, P. Cohen, S. West, and L. Aiken. Applied multiple regression/correlation analysis for the behavioral sciences. 1983.
- M. Di Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*, pages 217–226. IEEE, 2008.
- K. El Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *Software Engineering, IEEE Transactions on*, 27(7): 630–650, Jul 2001. ISSN 0098-5589. doi: 10.1109/32.935855.
- B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 725–743, 2007. URL <http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70731>.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.
- Y. Guéhéneuc and G. Antoniol. Demima: A multi-layered framework for design pattern identification. *Transactions on Software Engineering*, 2007.
- A. Güneş Koru and H. Liu. Identifying and characterizing change-prone classes in two large-scale open-source products. *J. Syst. Softw.*, 80(1):63–73, 2007. ISSN 0164-1212. doi: <http://dx.doi.org/10.1016/j.jss.2006.05.017>.
- S. Hayashi, J. Katada, R. Sakamoto, T. Kobayashi, and M. Saeki. Design Pattern Detection by Using Meta Patterns. *IEICE TRANSACTIONS on Information and Systems*, 91(4): 933–944, 2008.
- N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, 2000.
- F. Khomh, Y. Guéhéneuc, and G. Antoniol. Playing roles in design patterns: An empirical descriptive and analytic study. 2009.
- R. Lindeman, P. Merenda, and R. Gold. Introduction to bivariate and multivariate analysis. *New York*.
- T. Ng, S. Cheung, W. Chan, and Y. Yu. Work experience versus refactoring to design patterns: a controlled experiment. *Proceedings of the 13th ACM SIGSOFT 14th international symposium on Foundations of software engineering*, pages 12–22, 2006.
- T. Ng, S. Cheung, W. Chan, and Y. Yu. Do maintainers utilize deployed design patterns effectively? In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 168–177, May 2007. doi: 10.1109/ICSE.2007.33.
- D. Perry, A. Porter, and L. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 345–355. ACM New York, NY, USA, 2000.
- D. Posnett, C. Bird, and P. Devanbu. Thex: Mining Metapatterns in Java. In *Proceedings of the Seventh Working Conference on Mining Software Repositories*, Cape Town, South Africa, 2010. IEEE Computer Society.
- L. Prechelt, B. Unger, W. Tichy, P. Brossler, and L. Votta. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *Software Engineering, IEEE Transactions on*, 27(12):1134–1144, 2001.
- W. Pree. Meta patterns—a means for capturing the essentials of reusable object-oriented design. *Lecture Notes in Computer Science*, 821:150–162, 1994.
- J. Rosenberg. Some misconceptions about lines of code. In *Proceeding of the Fourth International Software Metrics Symposium*, pages 137–142, Nov 1997. doi: 10.1109/METRIC.1997.637174.
- Sci-Tools. Understand for java 1.4.

-
- E. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497. IEEE Computer Society Press, 1976.
- T. Tourwé and T. Mens. Automated support for framework-based software. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 148–157, 2003.
- H. Toutenburg. *Statistical Analysis of Designed Experiments*. Springer, second edition, 2002.
- M. Vokáč, W. Tichy, D. Sjøberg, E. Arisholm, and M. Aldrin. A controlled experiment comparing the maintainability of programs designed with and without Design Patterns—a replication in a real programming environment. *Empirical Software Engineering*, 9(3): 149–195, 2004.
- H. Zhang and H. B. K. Tan. An empirical study of class sizes for large java systems. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 230–237, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3057-5. doi: <http://dx.doi.org/10.1109/APSEC.2007.20>.