An Empirical Study of the Relationships between Design Pattern Roles and Class Change Proneness

Massimiliano Di Penta[1], Luigi Cerulo[1], Yann-Gaël Guéhéneuc[2], and Giuliano Antoniol[3]
[1] RCOST – Department of Engineering, University of Sannio, Italy
[2] Ptidej Team – GEODES Lab. – DIRO, Université de Montréal, Québec, Canada
[3] SOCCER Lab. – DGIGL, École Polytechnique de Montréal, Québec, Canada

Abstract

Analyzing the change-proneness of design patterns and the kinds of changes occurring to classes playing role(s) in some design pattern(s) during software evolution poses the basis for guidelines to help developers who have to choose, apply or maintain design patterns. Building on previous work, this paper shifts the focus from design patterns as wholes to the finer-grain level of design pattern roles.

The paper presents an empirical study to understand whether there are roles that are more change-prone than others and whether there are changes that are more likely to occur to certain roles. The study relies on data extracted from the source code repositories of three different systems (JHotDraw, Xerces, and Eclipse-JDT) and from 12 design patterns. Results obtained confirm the intuitive behavior about changeability of many roles in design motifs, but also warns about properly designing parts of the motif subject to frequent changes.

1 Introduction

The design pattern idea is likely to be one of the most relevant contributions to the field of software engineering in the past 20 years; the book of Gamma *et al.* [9] is one of the most influential software engineering book; it has deeply changed object orientation with a profound influence on the way of designing and developing object oriented software.

Design patterns [9] are solutions to recurring problem in software system design that help in improving reusability, maintainability, comprehensibility, and robustness. Design patterns suggest *design motifs*, which are prototypical solutions to design problems. The application of design patterns in a system results in *occurrences* of their motifs being disseminated in its source code. Despite 13 years passed from the Gamma *et al.* book [9] publication, many fundamental questions still remain unanswered. Design patterns are expected to promote software evolution; they easy maintenance tasks by explicitly identifying class roles and by localizing where extension and changes should occur.

This paper presents a study of the evolution of classes playing different roles in design motifs to understand whether (i) there are roles that are more change-prone than others and (ii) there are some kinds of changes that occur more to certain roles than to others. Kinds of changes include: change to method implementation, method addition/removal, attribute addition/removal, and extension by subclassing. Gamma *et al.*'s book [9] provides hints on the changeability of classes playing roles in different motifs but, to the best of our knowledge, no empirical study has been performed to verify if the current practice confirm the intuition of these authors or to assess if different roles for a given pattern behave differently in term of kinds and frequency of changes. Deviation from the expected theoretical or logical change behavior may indicate a poor understanding or an improper patterns usage, as well as an extension from intended initial pattern design to new applications. Empirical evidence on kinds and frequency of changes in different pattern roles is crucial to conceive and design a new generation of recommender systems [19] aiming at supporting object-oriented design pattern evolution and maintenance.

Recently, Aversano *et al.* presented a study on the evolution of design motifs in three different systems [2], with the objective of investigating (i) how frequently design motifs change, (ii) what kinds of changes different motifs are subject to, and (iii) what is the ability of motifs to make a system more robust to change. This previous study considered the occurrences of the motifs as "wholes" and inferred the characteristics of the changes from all the classes playing roles in the motifs. It did not analyze whether classes playing different roles exhibited different change frequency, and different kinds and amount of changes. This study builds upon and extends the previous contribution.

The empirical study is performed using three different systems from three different domains: JHotDraw, Xerces, and Eclipse-JDT, and the 12 design patterns presented in Ta-

ble 2. The design motif identification was performed using the DeMIMA approach and tool [12]; DeMIMA not only identifies the classes, but also the different roles played by classes in occurrences of motifs. We report new evidence on kind and frequency of changes; evidence often confirms the intuitive behavior—e.g., concrete classes tend to be more subject to changes than abstract classes—but sometimes, in a few more complex cases, it contradicts the expected intuitive behavior. Results suggest that developers must carefully design classes playing within a motif roles that will likely be subject to frequent changes and kinds of changes (e.g., in the method interfaces) that will likely impact elsewhere in the system.

This paper is organized as follows. Section 2 presents related work. Section 3 summarizes the design motif identification approach. Section 4 describes the extraction of the data required to perform the empirical study. Section 5 describes the context and research questions of the study. Section 6 reports the results of the study. Section 7 provides a discussion of the results and of threats to their validity. Finally, Section 8 concludes and outlines future work.

2 Related Work

Historical data on the changes of open-source systems allowed researchers to empirically investigate on the evolution of various software entities, such as clones [10, 14] or design motifs [2]. Some studies considered changes between releases, i.e., stable snapshots released after some amount of changes, while others exploited the historical data available in versioning systems, such as CVS or SVN. This data describes changes close in time and represents the developers' activities, such as bug fixing, refactoring, feature enhancement [8]. Such data has been used to predict change propagation [23, 24], identify cross-cutting concerns [4, 5], detect logical coupling among modules [8], identify common error patterns [17], or identify fix-inducing changes [15].

Bieman *et al.* [3] analyzed four small systems and one large system to study the change proneness of classes playing roles in some design motifs. In this paper, we perform a finer-grain study that distinguishes among different kinds of changes. Also, we distinguish each role played by a class. Vokáč [21] analyzed the corrective maintenance of a large commercial system over three years, comparing the defect rates of classes participating in design motifs against those that did not. He found that participating classes are less bug prone than others. We cast the study in this paper into both corrective maintenance and evolution tasks, and consider the roles of the classes.

Prechelt *et al.* [18] performed a series of controlled experiments to compare design motifs with alternative, simpler solutions to assess their impact on maintenance tasks. They concluded that design motifs could be beneficial to developers if, when applying design patterns, they would document their choice and evaluate alternatives solutions. Our study has a different perspective, by analyzing changes performed to classes playing roles in design motifs during the evolution of a software system.

Several design motif identification approaches have been proposed in the past. Kramer *et al.* [16] and Antoniol *et al.* [1] applied graph matching techniques to class diagrams and were able to identify occurrences of some structural design motifs. Tsantalis *et al.* [20] used graph similarity measures to identify occurrences of structural design motifs but a careful study of the identified occurrences revealed some imprecision. Costagliola *et al.* [6] used a visual language parsing technique to identify occurrences of design motifs with a high precision and recall but medium scalability. Heuzeroth *et al.* [13] combined static and dynamic analyses of class diagrams and execution traces. The accuracy of their approach is low, especially for occurrences that are not executed in the dynamic analysis.

Overall, no previous study addressed the same or similar questions to those reported in Section 5 providing evidence of the kind and frequency of changes in different design pattern roles.

3 Detection Approach

We use our Design Motif Identification Multi-layered Approach (DeMIMA) to identify occurrences of design motifs [12]. A design motif is typically a class diagram describing the solution to the recurring problem addressed by a design pattern. By design, DeMIMA ensures 100% recall and has been found to have up to 80% of precision, with an average of 34% for the 12 design motifs in Table 2.

In DeMIMA each design motif to be identified is modelled using a meta-model, PADL. The instances of PADL describe each motif in terms of the roles, methods, associations and inheritance relationships among roles.

Wee apply DeMIMA to identify and reify occurrences of the motifs in the source code of a program. DeMIMA uses explanation-based constraint programming to provide automatically explanations on the identified occurrences and in particular the roles and relationships in micro-architecture implementing an occurrence of a motif. More details can be found in [11, 12].

4 Data Extraction Process

The process followed to extract data is similar to the process used in our previous work [2], although here we rely on the DeMIMA [12] design motif discovery tool instead of using tool provided by Tsantalis *et al.* [20], since the former is able to identify all the roles of each motif.

In the first step of the approach we used DeMIMA to identify occurrences of design motifs on each release of the systems under study. DeMIMA identifies each role for each design motif and the class(es) playing these roles.

Once design motifs have been identified, motif occurrences are traced across releases. We infer the evolution across releases of each identified occurrences making the following assumption. An occurrence identified in release R_{j+1} is an evolution of an occurrence in R_j if and only if they share the sets of classes participating to their "main" roles (see Table 2). We define main roles based on Gamma *et al.*'s book [9] and our own experience. Table 2 presents the main roles of each motif. Thus, we can follow the evolution of occurrences across releases as "wholes", for example to identify when an occurrence was introduced and when it was modified and/or removed.

In parallel to the previous activities, snapshots are extracted and change sets identified. We group file revisions extracted from CVS/SVN repositories in sets of logically coupled changes performed by developers using Gall *et al.*'s technique [8]. Gall *et al.* consider the evolution of a system as a sequence of *snapshots* ($S_0, S_1, \ldots, S_m$) generated by a sequence of *change sets* (Δ_1, Δ_2, ..., Δ_m). Change sets represent the logical changes performed by developers in terms of added, deleted, and changed source code lines.

Change sets can be extracted from a CVS/SVN history log using different approaches. We adopt a time-windowing approach that considers a Δ_k as a sequence of file revisions that share the same author, branch, and commit notes, and with a difference between timestamps of less than 200 seconds [24]. In such a way we are able to compute the difference between two subsequent snapshots and identify which files have been changed by a developer.

Finally, we identify the changes to occurrences by using the change sets. We determine in which snapshot a class c_i participating to an occurrence has been changed by comparing the class revision in snapshots S_{j-1} and S_j. This comparison aims at identifying: (1) addition/removal/change of/to attributes and associations; (2) addition/removal of methods or changes to their signatures; (3) changes in method implementation; and (4) addition/removal of subclasses. The presence of at least one difference between $c_{i,j-1}$ and $c_{i,j}$ indicates that the class c_i, and thus the occurrence in which it plays a role, has been changed with respect to S_j.

5 Empirical Study Design

The *goal* of our study is to perform a fine-grain analysis of the evolution of classes playing roles in occurrences of design motifs, with the *purpose* of determining whether some roles are (i) more change prone, (ii) more subject to a particular kind of changes. The *quality focus* is the change-ability of classes playing some roles. The *perspective* is that of both researchers and practitioners to understand class change proneness with respect to design motifs and the rest of the system. The *context* is the analysis of design motif evolution in three object-oriented open-source systems: JHotDraw, Eclipse-JDT, and Xerces.

Table 1. Characteristics of the Systems

SYSTEMS	SNAPS	RELEASES	KNLOC	CLASSES
JHotDraw	177	5.2–5.4B2	13.5–36.3	164–489
Xerces-j	4952	1.3.1–1.4.3	35.4–294.1	162–1395
Eclipse-JDT	23424	1.0–3.1	205.5–535.5	2089–6961

Table 1 highlights the main characteristics of the the systems we analyzed. Table 2 briefly describes all the motifs detected by DeMIMA that were object of our study and, for each motif, the main roles used to trace the motif across releases are listed.

JHotDraw[1] is a Java framework for drawing 2D graphics. The framework came to existence in October 2000 with the purpose of illustrating the use of design patterns. We extracted a total of 177 snapshots from release 5.2 in March 2001 to release $5.4b2$ in February 2004. The size of the framework grew almost linearly from 13.5 KNLOC (*i.e.,* 1000 non commented lines of code) at release 5.2 to 36.5 KNLOC at release $5.4b2$. 72 Observers, 51 Adapters, 3 Singletons, 50 Template Methods, 2687 Abstract Factories, 25 Decorators, 30 States, 43 Commands, 465 Factory Methods, 64 Composites, and 1 Visitor were extracted from releases 5.1, 5.2, 5.3, $5.4b1$, and $5.4b2$. Here and in the following, the number of instances (e.g., 25 Decorators) refers to the number of different instances of different design motifs detected in all releases; if the same design motif with the same roles is detected in multiple releases it is counted just once.

Xerces-j[2] is a Java XML parser. The project started in 1999 from an initial code base of IBM's XML4J. We extracted a total of $4,952$ snapshots from release $1.3.1$ in July 2000 to release $1.4.3$ in August 2001. The system grew almost linearly from 35.4 to 294.1 KNLOC. 65 Observers, 57 Adapters, 63 Template Methods, 3958 Abstract Factories, 46 States, 76 Commands, 239 Factory Methods, 13 Composites, and 12 Visitors were extracted from releases $1.3.1$, $1.4.0$, $1.4.1$, $1.4.2$, and $1.4.3$.

Eclipse-JDT is a set of plug-ins that provides the capabilities of a full-featured Java IDE to the Eclipse platform[3]. This set of plug-ins is entirely in Java and its CVS repository is divided into three types of releases: stable, integration, and nightly builds. We extracted $19,750$ snapshots in the time interval between November 2001 and June 2004, when Eclipse 3.0 was released. The size of the set grew al-

[1] `http://www.jhotdraw.org`
[2] `http://xerces.apache.org/xerces-j`
[3] `http://www.eclipse.org`

Table 2. Design patterns and the main roles of their motifs

PATTERNS	DESCRIPTIONS	MAIN ROLES
Abstract Factory	Provides an interface for creating families of related or dependent objects without specifying their concrete classes	Abstract Factory, Abstract Product
Adapter	Converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces	Target, Adapter, Adaptee
Command	Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations	Invoker, Command, Client
Composite	Composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly	Composite, Component
Decorator	Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality	Component, Decorator
Factory Method	Defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses	Creator, Product
Observer	Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically	Subject, Observer
Prototype	Specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype	Prototype
Singleton	Ensures a class only has one instance, and provide a global point of access to it	Singleton
State/Strategy	Allows an object to alter its behavior when its internal state changes / Define a family of algorithms, encapsulate each one, and make them interchangeable	State/Strategy, Context
Template Method	Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure	Abstract Class
Visitor	Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates	Visitor, Element

most linearly from 205.5 KNLOC to 449.4 KNLOC, while the number of classes doubled in the first year and then grew almost linearly in the following two years. 390 Observers, 6 Singletons, 478 Adapters, 541 Template Methods, 297 Decorators, 317 States, 544 Commands, 4872 Factory Methods, 149 Composites, and 66 Visitors were extracted from releases 1.0, 2.0, 2.1, $2.1.2$, and $2.1.3$.

The high number of Factories (both Abstract and Method) detected in each system is mainly due to the fact that we consider as participants also classes coming from *java.** packages.

5.1 Research Questions

This empirical study aims at answering the following research questions:

- **RQ1 – Change frequency:** Are classes playing a particular role in a given design motif more change-prone than classes playing other roles?
- **RQ2 – Kind of change:** Are classes playing a particular role in a given design motif more subject to particular kinds of changes?

5.2 Analysis Method

We use different kinds of statistical analyses—performed using the R statistical environment[4]—to answer the research questions:

- **RQ1:** We compare the differences in terms of number of changes among different roles of all occurrences of a motif using Kruskal-Wallis test, which is a non-parametric test for testing the difference among multiple medians. We use the non-parametric Mann-Whitney test to perform pair-wise comparisons among motifs. We correct the significance level (95% in all the tests) using Bonferroni correction, *i.e.*, dividing the significance by the number of tests performed (the number of combinations across roles), because the presence of significant differences implied a comparison among all roles for a particular motif. For sake of simplicity, we report corrected p-values, *i.e.*, p-values multiplied by the number of tests performed.
- **RQ2:** We use proportion test to investigate whether the proportion of changes of a given kind significantly varies across roles of all occurrences of a given motif.

[4]http://www.r-project.org

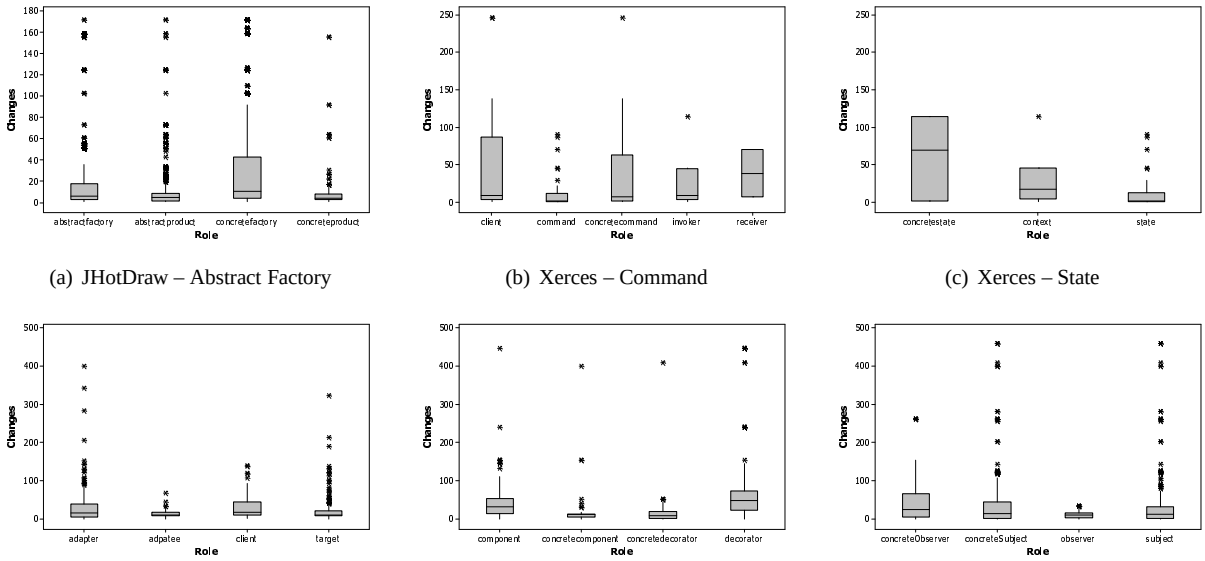
6 Empirical Study Results

Due to the lack of space, we now report the most interesting results and figures of our empirical study. The interested reader can access the complete study in [7].

6.1 RQ1 – Change frequency

Figure 1, for some selected design motifs of the three systems, shows change frequency per roles in the set of analyzed snapshots.

In JHotDraw, the Kruskal-Wallis test indicates that only Abstract Factory (Figure 1-a, p-value $< 2.2 \cdot 10^{-16}$) and Factory Method (p-value $= 6.5 \cdot 10^{-7}$) exhibit significant differences among roles. For the Abstract Factory, the Concrete Factory role exhibits a significantly higher number of changes than the Abstract Factory role (p-value $= 1.3 \cdot 10^{-15}$). For the Factory Method, the Concrete



(a) JHotDraw – Abstract Factory (b) Xerces – Command (c) Xerces – State

(d) Eclipse-JDT – Adapter (e) Eclipse-JDT – Decorator (f) Eclipse-JDT – Observer

Figure 1. Examples of significant change frequency among roles.

Creator role changes more than the Creator role (p-value $= 0.04$).

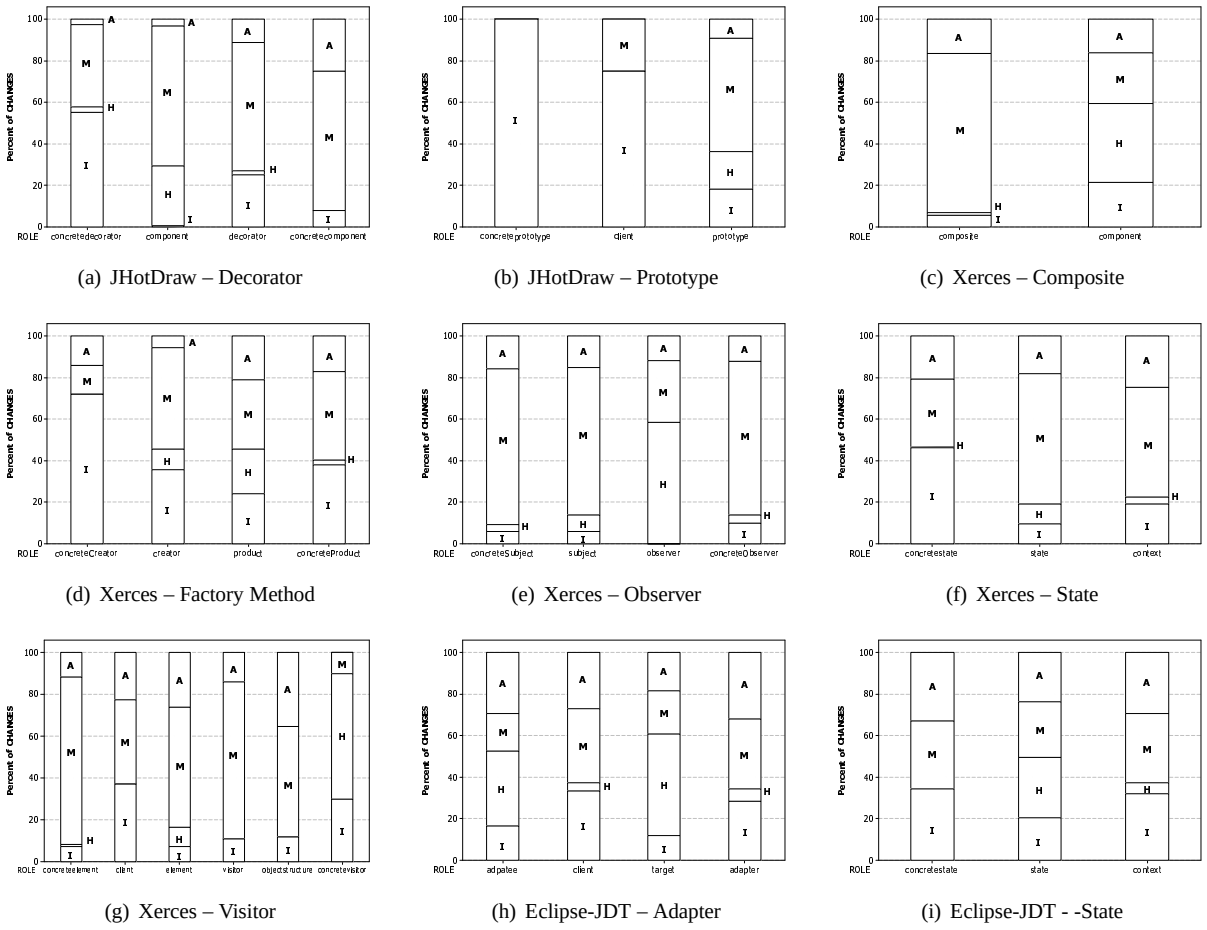
In Xerces, significant differences exist for Abstract Factory (p-value $< 2.2 \cdot 10^{-16}$), Factory Method (p-value $= 4.1 \cdot 10^{-8}$), Command (Figure 1-b, p-value $= 0.01$), State (Figure 1-c, p-value $= 0.004$), and Composite (p-value $= 0.0008$). For Abstract Factory, similarly to JHotDraw, the Concrete Factory role changes more than the Abstract Factory role (p-value $= 1.3 \cdot 10^{-15}$). For Factory Method, similarly to JHotDraw, the Concrete Creator role changes more than the Creator role (p-value $= 2.4 \cdot 10^{-7}$). For Command, the Client role changes more than the Command role (p-value $= 0.02$). In State/Strategy, the Concrete State changes more than the State role (p-value $= 0.04$). Finally, for Composite, the Composite role changes more than the Component role (p-value $= 9.1 \cdot 10^{-4}$).

In Eclipse-JDT, we identify significant differences among roles for several motifs, also considering that the number of motifs instances and of snapshots was higher than for other systems:

- Adapter (Figure 1-d, p-value $= 3.7 \cdot 10^{-6}$): the Client (p-value $= 1.7 \cdot 10^{-5}$) and Adapter roles (p-value $= 0.007$) change more than the Target role;
- Composite (p-value $= 0.0002$): the Composite role changes more than the Component role (p-value $= 9.2 \cdot 10^{-5}$);
- Decorator (Figure 1-e, p-value $< 2.2 \cdot 10^{-16}$): the Component (p-value $= 5.9 \cdot 10^{-7}$), Concrete Decorator (p-value $= 1.3 \cdot 10^{-15}$) and Concrete Component (p-value $< 2.2 \cdot 10^{-7}$) roles change more than the Decorator role; the Concrete Component change more than the Component role (p-value $= 3.9 \cdot 10^{-4}$);
- Factory Method (p-value $< 2.2 \cdot 10^{-16}$): the Concrete Creator role changes more than the Creator role (p-value $= 1.3 \cdot 10^{-15}$); the Concrete Product role changes more than the Product role (p-value $= 6.5 \cdot 10^{-6}$);
- Template Method (p-value $= 2.9 \cdot 10^{-6}$): the Client role changes more than the Abstract Class role (p-value $= 4.0 \cdot 10^{-6}$);
- Command (p-value $= 1.7 \cdot 10^{-6}$): the Client role changes more than the Command role (p-value $6.2 \cdot 10^{-6}$);
- Observer (Figure 1-f, p-value $= 3.1 \cdot 10^{-11}$): the Concrete Observer role changes more than the Observer role (p-value $= 5.5 \cdot 10^{-13}$);
- State (p-value $= 0.02$): the Context role changes more than the State role (p-value $= 0.004$).

6.2 RQ2: Kind of Change

In JHotDraw, we find that for all motifs, but Decorator and Prototype, classes playing any role mostly change because of (M)ethod addition/removal. Proportions are between 52% and 81%. For Abstract Factory (p-value $<$



(a) JHotDraw – Decorator (b) JHotDraw – Prototype (c) Xerces – Composite

(d) Xerces – Factory Method (e) Xerces – Observer (f) Xerces – State

(g) Xerces – Visitor (h) Eclipse-JDT – Adapter (i) Eclipse-JDT - -State

Figure 2. Examples of Kinds of Changes in Roles (A: Attribute change; M: Method; I: Implementation; H: inHeritance).

$2.2 \cdot 10^{-16}$), the Concrete Factory role changes more, while the Concrete Product role changes less. For Command (p-value $= 0.001$), Factory Method (p-value $< 2.2 \cdot 10^{-16}$), and Observer (p-value $= 9.7 \cdot 10^{-14}$), proportions significantly vary among roles.

For Decorator (see Figure 1-a), all classes have a large proportion of changes due to (M)ethod addition/removal, however classes belonging to the Concrete Decorator role mostly change because of (I)mplementation changes with a proportion of 39%. For both kinds of changes (p-values $= 0.0004$ and $< 2.2 \cdot 10^{-16}$), proportions of changes significantly vary among roles.

For Prototype (see Figure 2-b), (I)mplementation changes significantly differed among roles (p-value $= 0.005$) and are the largest proportion for all roles (75% for Client, 100% for Concrete Prototype), but for the Prototype role, where the largest proportion of 55% is due to (M)ethod addition/removal.

For Visitor, no statistical conclusion is possible due to the limited number of changes and occurrences.

In Xerces, (M)ethod addition/removal is the most frequent change for Abstract Factory (with a higher proportion of 75% for the Concrete Product role), Adapter (with a higher proportion of 67% for the Client role), Command (with a higher proportion of 78% for the Receiver role), and Template Method (with a higher proportion of 66% for the Client role).

For Composite (see Figure 2-c), Composite classes mainly have (M)ethod addition/removal changes (proportion of 77%) while the Component role has more addition/removal of subclasses (H) with 38%.

For Factory Method (see Figure 2-d), the Concrete Product role has mostly (M)ethod addition/removal changes with a proportion of 43%, and (I)mplementation changes

with 38%. The Concrete Creator role has mostly (I)mplementation changes with a proportion of 72%. (M)ethod addition/removal changes dominate in the Creator and Product roles with 49% and 33% respectively.

For Observer (see Figure 2-e), (M)ethod addition/removal changes predominate for Concrete Observer with 74%, Concrete Subject with 75%, and Subject with 71%, and Observer has mostly changes in hierarchies (H) with 58%.

For State/Strategy (see Figure 2-f), the Context and State/Strategy roles have a higher proportion of (M)ethod addition/removal changes with 53% and 63% respectively, while the Concrete State/Strategy have a higher proportion of (I)mplementation changes.

Finally, for Visitor (see Figure 2-g, (M)ethod addition/removal changes predominate for all roles but for the Concrete Visitor role, which has mostly changing to its subclasses with a proportion of 60%.

Results of proportion tests indicate a significant difference among roles for all kinds of changes but for (A)ttribute changes for Composite (p-value $= 1$) and for Observer (p-value $= 0.37$).

In Eclipse-JDT, proportions of changes vary among motifs and roles. In particular:

- For Adapter (see Figure 2-h), Adaptee and Target roles mainly have changes to their subclasses with a proportion of 36% and 49% respectively. The changes to the Adapter and Client roles are almost equally distributed among kinds of changes, however for changes to their subclasses (H) below 10%.
- For Composite, the Component role has mostly changes to its subclasses with a proportion of 56%, while the Leaf role has all kinds of changes almost equally distributed, however with no changes to its subclasses at all.
- For Decorator, all roles but the Component role are subject to all kinds of changes, but the subclassing changes. The Component role undergoes changes to its subclasses, with a higher proportion for (M)ethod addition/removal (32%).
- For Command, the Client and Invoker roles have changes equally distributed across all kinds with subclassing changes below 10%.
- For Factory Method, we notice that the Creator and Product roles have mostly changes to their subclasses (43% and 47%); such kind of changes is below 5% for the other roles.
- For Template Method, subclassing changes (H) are the most frequent for Abstract Classes (49%) and (M)ethod addition/removal for Clients (39%).
- For Observer, (M)ethod addition/removal are the most frequent for Concrete Observers (48%), Concrete Subjects (40%), and Subjects (38%). Subclassing changes (H) are most frequent for the Observer role (73%).
- For Prototype, the number of changes and of occurrences are small, and no significant difference is found among roles for (A)ttribute and (M)ethod addition/removal.
- For State/Strategy (see Figure 2-i), changes for the Concrete State/Strategy and Context roles are equally distributed across all kinds of changes but for subclassing changes.
- For Visitor, Client role has mostly (I)mplementation changes (48%), while Concrete Element role has 32% and 31% and (M)ethod addition/removal and (I)mplementation changes.

7 Discussion

We discuss the results by comparing intuition and practice for each motif with observations gathered from the empirical data. We can therefore attempt to confirm/refute intuition and put into perspective the data by inspecting the source code. Due to space limitation, we only report discussion when evidence or interesting findings have been discovered, thus for example we omit discussion of both Prototype and the Singleton. The first because no evidence on role change frequencies or kind or differences were found; the latter since not only no evidence was found, but also its intent is to provide an unique access point rather than making the system robust to changes; in other words *RQ1* and *RQ2* have no meaning for the Singleton.

Abstract Factory. *Intuition:* Any change to the set of Abstract Products will impact the interface of the Abstract Factory and all the Concrete Factories. Depending on the system, new Products may be added, removed, or modified but we could expect that little change would happen to the Products in mature systems as they usually form the core of their functioning, for example a set of widgets in a graphic library. Changes to the Abstract Factory interface could happen, but mostly to the non-Product related methods, for example helper methods or constructors. Such changes could happen in any release. Therefore, we could expect the set of Factories and Products to change rarely, the Factories being more change-prone than the products.

Observation: The intuition is confirmed both in JHotDraw and Xerces, where Concrete Factories change more frequently than Abstract Factories. However, (M)ethod addition/removal changes, affecting their interfaces, are the

most common changes even for Abstract Factories. The importance of such changes could be explained by the fact that both systems underwent major refactorings in which the common interface to the Concrete Factories were adapted to the new designs.

Adapter. *Intuition:* The intent of the Adapter motif is to separate the Target from the Adaptee through the Adapter to allow them to change without impacting one another. Therefore, we could expect the Target and Adaptee roles to change separately, while any change would impact the Adapter. Given the important role of the Target with respect to Clients, we could expect Targets to be less change-prone than Adaptees.

Observation: The intuition is confirmed in Eclipse-JDT as Adapters exhibit more changes that Targets. The most common change for Adaptees is change to their subclasses, *i.e.*, new Adapters are modified/added to adapt to changing/new Targets.

Command. *Intuition:* The Command interface defines usually only one method, typically *execute()*, and therefore can be expected to be little subject to changes. On the contrary, the Receiver defines the methods performing the concrete actions and therefore would probably evolve as the system evolves and be, consequently, more change prone.

Observation: The intuition is confirmed in Xerces, where Receivers change more than Commands, although the difference is not significant. In all the three systems Commands appear to be less prone than other roles, especially less prone than Clients.

Composite. *Intuition:* The Component interface defines the methods that must be implemented by both the Composite and the Leaves, in a mature system, we could expect this interface to change rarely. The Composite implements an iterator over its set of Leaves. Therefore, we could expect also the Composite to change rarely. The Leaves implement the algorithms that must be triggered recursively by the Composite. These algorithms are system and context dependent. Consequently, the Leaf role is more change prone than both the Component and the Composite roles. We could also expect the Composite and Leaf roles to simultaneously adapt to changes occurring in the Component.

Observation: The intuition is not confirmed. In JHotDraw, no significant difference is found among roles; in Xerces changes are concentrated in Composites, while Leaves do not change at all; finally, in Eclipse-JDT, Composites change more than Components, while no significant difference is found with Leaves. As expected, Component has mainly subclassing changes. In many cases, classes playing the role of Composite have to handle non trivial tasks, thus they are subject to a high number of changes. For example, in JHotDraw a composite figure has to properly handle the layouting of all the figures part of the composition.

Decorator. *Intuition:* Similarly to the Composite motif, the Concrete Components are probably more change prone than the Component and Decorator roles. The Concrete Decorators might change independently from Components and Decorators, since they are system and context dependent. Therefore, we could expect that the Concrete Components be change prone, as would the Concrete Decorators.

Obervation: The intuition is confirmed in Eclipse-JDT, where Concrete Components and Decorators are more change prone than other roles. Nothing can be said for other systems. On the other hand, it is worth to be noted that, although a high change frequency is expected for Concrete Decorators, Decorators often have a higher change frequency. As for the Composite, this is due to the fact that Decorators have to handle complex tasks and in these case they might be involved in tasks not properly part of the motif, as discussed in Section 7.1.

Factory Method. *Intuition:* Similarly to the Abstract Factory motif, we could expect that any change to the Concrete Products would impact the Concrete Creators, while the Creator would change depending on the system and on the context.

Observation: In all the systems, the intuition is confirmed: Concrete Creators change more than Creators. For Eclipse-JDT, Concrete Products also change more than Products. Differently from the Abstract Factory motif, Abstract Classes (Creators and Products) mainly have subclassing changes rather than (M)ethod addition/removals.

Observer. *Intuition:* The Subject and Observer interfaces define the set of methods that their respective implementations—Concrete Subjects and Concrete Observers—should implement. Therefore, we could expect these two roles to change rarely. On the contrary, the roles Concrete Subject and Concrete Observer are system and context dependent and would change according to the developers' needs.

Observation: Results from Eclipse-JDT confirm the intuition. Also, concrete classes tend to have (M)ethod addition/removal changes, while abstract classes tend to have subclassing changes, as expected.

State/Strategy. *Intuition:* The State/Strategy role defines an interface, typically embodied in a method *handle()* (or *execute()*), to be used by Clients. Therefore, we could expect this role to be less change prone than the Concrete States/Strategies, which implement the concrete logic of the

States/Strategies and are thus dependent on the system and its context. We also expect that Contexts change more than States/Strategies.

Observation: The intuition is confirmed in Xerces, where Concrete States change more than States. In Eclipse-JDT, we observe that Contexts classes change more than State classes.

Template Method. *Intuition:* The Template Method motif is typically used in class-based programming languages to offer some flexibility in the implementations of a set of methods, typically called "primitive operations". Therefore, we could expect the Concrete Classes implementing these operations to be more change prone than the Abstract Classes.

Observation: The intuition is confirmed in Eclipse-JDT only, where Abstact Classes change less than Concrete Classes. As expected, Abstract Classes mainly underwent subclassing changes, while Concrete Classes underwent more (M)ethod addition/removal and (I)mplementation changes.

Visitor. *Intuition:* Similarly to the Abstract Factory motif, the Visitor separates between a set of Elements, representing usually a data structure, and a set of Visitors performing operations on this structure. Therefore, we could expect the Elements to change less in mature systems than the Concrete Visitors.

Observation: Nothing can be said regarding the frequencies of changes. We notice a high proportion of subclassing changes in Concrete Visitors for both Xerces and Eclipse-JDT. We would have expected high proportion for Visitors rather than for Concrete Visitors. This proportion suggests that developers of Xerces and Eclipse-JDT tend to reuse Concrete Visitors to create new ones.

7.1 Threats to Validity

Construct Validity threats concern the relationship between theory and observation. They are mainly due to the precision and recall of DeMIMA, although reported recall is 100% and precision between 34% and 80% on average. These figures are consistent with the precision and recall of other approaches considering that all roles are identified. We encountered no scalability issue. Another threats to construct validity depends on the fact that we analyze changes on classes that play a role in a motif, although these changes might be related to pieces of functionality a class implements outside the motif. We limit this threat by performing a manual inspection of changes reporting and discussing (see Section 7) unexpected or contradictory cases.

Threats to *internal validity* did not affect this study, being an explorative study [22].

Threats to *external validity* concern the generalization of our conclusions. We considered three systems with different domains and sizes. Future work should include reproducing this study with other systems and programming languages. We considered a convenient sampling of 12 out of 23 motifs from Gamma *et al.*'s book: those currently detected by DeMIMA.

Threats to *reliability validity* concern the possibility of replicating our study and obtaining consistent results. The source code of the three systems is publicly available; DeMIMA is described in [12], and this paper describes our analyses. Statistical conclusions were supported by Kruskal-Wallis tests for analyzing multiple medians, Mann-Whitney tests for un-paired comparisons of two medians, corrected using Bonferroni correction, and proportion test for comparing proportions.

8 Conclusion and Future Work

This paper reported an empirical study aimed at investigating on the frequency and on the kind of changes of classes playing different roles in a set of design motifs, corresponding to well-known and widely used object oriented design patterns.

Our findings on three open source Java systems of different size and from different domains—namely JHotDraw, Xerces-J, and Eclipse-JDT—often confirm the expected and intuitive behavior. For example, in the Adapter motif, classes playing the role of Adapter is more change prone, while for the Adaptees we can mainly observe changes in its number of subclasses. In a similar way, intuition is confirmed for the Abstract Factory and Command. For the former, Abstract Factories and Products change less frequently than concrete ones; for the latter, Receivers change more frequently than Commands. Overall, the obtained results suggest to carefully design—e.g., in terms of method interfaces—roles more subjects to changes, since their change proneness can make other parts of the system less robust to changes. This confirms and makes even stronger the suggestions made in paper [2] about carefully designing patterns playing a crucial role in the application.

There are, however, many cases where the actual, observed changes deviates from the intuition and from the common wisdom. For example, in the Composite motifs classes playing the role of Composite can be more complex than expected and because of that undergoing a high number of changes.

Future works will be devoted to extend the empirical investigation to a larger set of systems, to investigate on different roles co-changes, and to find evidence of correlations between kind of changes, frequency of changes and defects in classes playing particular roles in the motifs.

9 Acknowledgments

Massimiliano Di Penta and Luigi Cerulo are partially supported by the project METAMORPHOS (MEthods and Tools for migrAting software systeMs towards web and service Oriented aRchitectures: exPerimental evaluation, usability, and tecHnOlogy tranSfer), funded by MiUR (Ministero dell'Università e della Ricerca) under grant PRIN2006-2006098097.

References

[1] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.

[2] L. Aversano, G. Canfora, L. Cerulo, C. Del 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 Press.

[3] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *9th International Software Metrics Symposium (METRICS'03)*, pages 40–49. IEEE Computer Society, 2003.

[4] S. Breu and T. Zimmermann. Mining aspects from version history. In S. Uchitel and S. Easterbrook, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 221–230. ACM Press, September 2006.

[5] G. Canfora, L. Cerulo, and M. Di Penta. On the use of line co-change for identifying crosscutting concern code. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, PA, USA*, pages 213–222, 2006.

[6] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. Design pattern recovery by visual language parsing. In *9th European Conference on Software Maintenance and Reengineering (CSMR 2005), 21-23 March 2005, Manchester, UK*, pages 102–111, 2005.

[7] 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. Technical report, RCOST - Univ. of Sannio, Italy, 2008. *http://rcost.unisannio.it/mdipenta/rolesTR.pdf*.

[8] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13. IEEE Computer Society, 2003.

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

[10] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering (FASE)*, number 3922 in LNCS, pages 411–425, Vienna, Austria, March 2006. Springer.

[11] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In D. C. Schmidt, editor, *Proceedings of the 19^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM Press, October 2004.

[12] Y.-G. Guéhéneuc and G. Antoniol. DeMIMA: a multi-layered framework for design pattern identification. *Transactions on Software Engineering*, December 2007. *Under revision.*

[13] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *11th International Workshop on Program Comprehension (IWPC 2003), May 10-11, 2003, Portland, Oregon, USA*, pages 94–103, 2003.

[14] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, pages 187–196, Lisbon, Portogal, September 2005.

[15] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 81–90. IEEE Computer Society, 2006.

[16] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of IEEE Working Conference on Reverse Engineering*, pages 208–215, 1996.

[17] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305. ACM Press, 2005.

[18] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Trans. Software Eng.*, 27(12):1134–1144, 2001.

[19] P. Resnick and H. R. Varian. Recommender systems. *Special Issue of the Commun. ACM*, 40(3):56–89, 1997.

[20] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Software Eng.*, 32(11):896–909, 2006.

[21] M. Vokáč. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Software Eng.*, 30:904–917, 2004.

[22] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.

[23] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Trans. Software Eng.*, 30:574–586, sep 2004.

[24] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.