An empirical study on the evolution of design patterns

Conference Paper · September 2007

DOI: 10.1145/1287624.1287680 · Source: DBLP

L. Aversano
University of Sannio
168 PUBLICATIONS **2,639** CITATIONS
SEE PROFILE

Gerardo Canfora
University of Sannio
353 PUBLICATIONS **13,858** CITATIONS
SEE PROFILE

Luigi Cerulo
University of Sannio
127 PUBLICATIONS **3,435** CITATIONS
SEE PROFILE

Massimiliano Di Penta
University of Sannio
452 PUBLICATIONS **22,975** CITATIONS
SEE PROFILE

An Empirical Study on the Evolution of Design Patterns

Lerina Aversano, Gerardo Canfora, Luigi Cerulo,
Concettina Del Grosso, Massimiliano Di Penta
RCOST – Research Centre on Software Technology, University of Sannio
Via Traiano, 82100 Benevento, Italy
aversano@unisannio,it, canfora@unisannio.it, lcerulo@unisannio.it,
tina.delgrosso@unisannio.it, dipenta@unisannio.it

ABSTRACT

Design patterns are solutions to recurring design problems, conceived to increase benefits in terms of reuse, code quality and, above all, maintainability and resilience to changes.

This paper presents results from an empirical study aimed at understanding the evolution of design patterns in three open source systems, namely JHotDraw, ArgoUML, and Eclipse-JDT. Specifically, the study analyzes how frequently patterns are modified, to what changes they undergo and what classes co-change with the patterns. Results show how patterns more suited to support the application purpose tend to change more frequently, and that different kind of changes have a different impact on co-changed classes and a different capability of making the system resilient to changes.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools And Techniques—*Object-oriented design methods*

General Terms

Design, Experimentation, Measurement

Keywords

Design patterns, Software Evolution, Mining Software Repositories, Empirical Software Engineering

1. INTRODUCTION

It has been claimed that the use of design patterns — i.e., of recurring design solutions for object-oriented systems — provides several advantages, such as increased reusability, and improved maintainability and comprehensibility of existing systems [11]. A relevant benefit of design patterns is the resilience to changes, avoiding that new requirements, and in general any kind of system evolution, causes major re-design. Gamma *et al.* [11] state *"Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change"*. Advantages of design patterns include decoupling a request from specific operations (Chain of Responsibility and Command), making a system independent from software and hardware platforms (Abstract Factory and Bridge), independent from algorithmic solutions (Iterator, Strategy, Visitor), or avoid modifying implementations (Adapter, Decorator, Visitor). Further discussion on design pattern advantages, and extensive pattern catalogues can be found in books such as [11] or [9].

While many benefits related to the use of design patterns have been stated, a little has been done to empirically investigate pattern change proneness [3] or whether there is a relationships between the presence of defects in the source code and the use of design patterns [24]. In particular, there is lack of empirical studies aimed at analyzing what kind of changes each type of pattern undergoes during software evolution, and whether such a change can be related to changes contextually made on other classes not belonging to the pattern. The availability of source repositories for many object-oriented open source systems realized making use of design patterns, of techniques for identifying change sets [10] — i.e., sets of artifacts changed together by the same author — from source code repositories, and of design pattern detection techniques and tools [1, 8, 15, 19, 23], triggers opportunities for this kind of studies.

This paper reports and discusses results from an empirical study aimed at analyzing how design patterns change during a software system lifetime, and to what extent such changes cause modifications to other classes not part of the design pattern. The study has been performed on three Java software systems, JHotDraw, ArgoUML and Eclipse-JDT. First, we detected design patterns on different subsequent releases of the three systems by using the approach and tool presented by Tsantalis *et al.* [23]. Then, we mined co-changes from Concurrent Versioning System (CVS) repositories to identify when a pattern changed, what kind of change was performed, which classes co-changed with the pattern, whether these classes had a dependency to or from the pattern, and what was the relationship between the type of change made and the resulting co-change.

The remainder of this paper is organized as follows. After a review of the literature in Section 2, Section 3 details the process to extract the information needed to perform the empirical study. Section 4 describes the empirical study context and research questions. Section 5 reports and discusses the case study results. Section 6 discusses the study threats to validity. Finally, Section 7 concludes the paper and outlines directions for future work.

2. RELATED WORK

In our knowledge there are a very few papers aiming at empirically analyzing the evolution of design pattern. Bieman *et al.* [3] conducted a study closely related to ours. They analyzed four small size systems and one large size system to identify the observable effects of the use of design patterns, such as pattern change proneness. Our work differs in the level of change granularity and on the aspects considered: they considered differences between releases and did not consider types of changes, while we consider differences between snapshots generated by CVS modification transactions. This leads to obtain a sample set that contains information about the type of changes performed by developers, thus allowing for the empirical analysis presented in this paper. Vokáč [24] analyzed the corrective maintenance of a large commercial product over three years, comparing defect rates for classes that participated in design patterns versus those that did not participate. In contrast to our work, he only focused on defects, while (i) we focus on both corrective and evolutive maintenance; (ii) our study also aims at relating change type to co-changes; and (iii) we perform a study on three open source systems having different characteristics with the aim of identifying commonalities and differences in pattern evolution activities. Finally, Prechelt *et al.* [22] performed a series of controlled experiments with the aim of compare design patterns with alternative, simpler solutions to perform maintenance tasks. They concluded that design patterns could be beneficial provided that developers, when introduce design patterns, properly document them and evaluate alternatives solutions. Rather than focusing on the effect of design patterns on a single maintenance task, our study focuses on analyzing pattern changes during software system evolution.

Historical co-change analysis has provided new opportunities for a number of issues: predict change propagation [26, 27], observe clones [12, 16] and crosscutting concerns [5] evolution, identify crosscutting concerns [4, 7], detect of logical coupling between modules [10], find common error patterns [20], or identify fix inducing changes [18].

The study presented in this paper relies on a design pattern recovery approach proposed by Tsantalis *et al.* [23]. However, several other approaches have been proposed in the past. Some of them relied on static analysis techniques, applying cliché matching on class diagrams, for example the approach proposed by Kramer *et al.* [19], by Antoniol *et al.* [1] or by Gueheneuc *et al.* [14]. The approach proposed by Costagliola *et al.* [8] identifies design patterns by visual language parsing, while the approach proposed by Heuzeroth *et al.* [15] combines static analysis with dynamic analysis on execution traces. We share with all the above authors the idea that design pattern identification into existing source code is a powerful mechanism to assess code quality, in particular indicating whether the use of design patterns makes the source code more resilient to changes.

While we focus on design patterns, other studies investigate on the evolution of micro patterns. Micro patterns are similar to design patterns, although their level of abstraction is closer to the implementation. Gil and Maman [13] presented a catalog of 27 micro-patterns defined starting from Java classes and interfaces. They performed an analysis on the prevalence of micro patterns and showed that a majority of Java classes follows one or more of the patterns in the catalog. Kim *et al.* [17] observed the evolution of micro patterns on three open source projects, analyzing micro pattern frequencies, common kinds of micro pattern evolution, and bug-prone micro patterns. They found that the micro pattern distributions and common change kinds are similar across the analyzed projects. Besides the different level of abstraction our study differs from the above mentioned ones since, as it will be shown in Section 5, distributions of pattern changes and change frequencies are not common among different patterns and among the analyzed projects.

3. PATTERN ANALYSIS PROCESS

This section describes the steps necessary to extract, from the CVS of the system under analysis, the data required to perform the empirical study presented in this paper.

3.1 Step 1: Detecting Design patterns

The first step concerns the identification of design patterns on each release of the system. This is performed using a graph-matching based approach proposed by Tsantalis *et al.* [23]. This approach is based on similarity scoring between graph vertices. It takes as inputs both the system and the pattern graph (cliché) and computes similarity scores between their vertices. Due to the nature of the underlying graph algorithm, this approach is able to recognize not only patterns in their basic form — the ones perfectly matching the cliché described in the Gamma *et al.* book [11] — but also modified versions (variants). The analysis has been performed using the tool[1] developed by Tsantalis *et al.*, which analyzes Java bytecode. The tool identifies the two main participants (i.e., superclasses) of each pattern, and is able to detect the following patterns: Object Adapter-Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State-Strategy, Template Method, and Visitor. The whole set of classes belonging to a pattern is made of main participants and their descendants.

3.2 Step 2: Reconstructing pattern evolution history across releases

Once having identified instances of design patterns in each release of the software system, it is necessary to trace them, i.e., to identify whether a pattern instance identified in release j represents an evolution of a design pattern instance identified in release $j-1$. This allows for reconstructing the history of each pattern instance, i.e., in which release it was created and when it was removed. Studying and discussing such an history — i.e., when and why patterns are created and removed during software evolution — is not part of this work and will be treated in other articles. To build the pattern history, we assume that a pattern instance in release j represents the evolution of a pattern instance in release $j-1$ if and only if (i) the type of pattern is the same; and (ii) at least one of the two main participant classes of the pattern is the same class in both releases $j-1$ and j.

[1]http://java.uom.gr/∼nikos/pattern-detection.html

3.3 Step 3: snapshots extraction and co-change identification

After having identified the history of each pattern instance, we analyze its changes by mining the CVS of the system under analysis. CVS can certainly provide us information on how and when a class belonging to the pattern was changed. However, we are also interested to analyze the set of classes that the same developer changed together with the pattern. To this aim we rely on a technique to extract from CVS/SubVersioN repositories logical coupled changes performed by developers working on a bug fix or an enhancement feature [10]. Such techniques consider the evolution of a software system as a sequence of *Snapshots* (S) generated by a sequence of *Modification Transactions* (MTs) (also known as Change Sets), representing the logical changes performed by a developer in terms of added, deleted, and changed source code lines. MTs can be extracted from a CVS history log using various approaches. We adopt a time-windowing approach that considers a MT as sequence of file revisions that share the same author, branch, and commit notes, and such that the difference between the timestamps of two subsequent commits is less or equal than 200 seconds [27].

3.4 Step 4: Locating pattern changes and determining the kind of change

To determine in which snapshot a class c_i participating to the pattern has been changed, we analyze the source code files changed in each snapshot, and perform a comparison between the class revision in snapshot $j-1$ and in snapshot j. Such a comparison aims at identifying: (i) addition/removal/change of attributes and associations; (ii) addition/removal of methods or changes in their signatures; (iii) changes in methods source code; and (iv) addition/removal of subclasses. The presence of at least a difference between $c_{i,j-1}$ and $c_{i,j}$ indicates that the class c_i, and thus the pattern, has been changed in correspondence of the snapshot j. The analysis is performed by using a fact extractor based on the JavaCC parser generator[2] and a Perl script that compares facts of $c_{i,j-1}$ and $c_{i,j}$ to identify the above mentioned differences.

3.5 Step 5: Analyzing pattern co-change

One of the objectives of this paper is to investigate on the code that changes contextually with the pattern. To this aim we use the same analyzer described in Section 3.4 to identify the set of classes that co-change with the pattern while not directly belonging to the pattern. Let $C_i \equiv \{c_{1,i}, \dots c_{n,i}\}$ be the set of classes that have been changed in the snapshot i. If we consider $C_i \equiv P_i \cup \bar{P}_i$, i.e., C_i is the union of the changed classes belonging to the pattern (P_i) and those not belonging to the pattern $\bar{P}_i$. Changes in $\bar{P}_i$ can be either changes requiring the pattern to be modified or changes subsequent to the pattern modification. At a finer-level of detail, we identify, within $\bar{P}_i$, the following two (not necessarily disjoint) subsets of classes:

- the set of potential pattern *clients*, i.e., classes not directly belonging to the pattern that depends on least one of the pattern classes. Examples are the Adapter client, that needs to access a piece of functionality provided by the Adaptee, or the Visitor client, using the Visitor to perform some operations on a data structure;
- the set of potential pattern *targets*, i.e., classes not directly belonging to the pattern on which the pattern depends on. For example, the set of classes used by a Façade to export a higher-level piece of functionality.

[2]https://javacc.dev.java.net/

Table 1: Case study history characteristics

System	Snaps	Releases	KNLOC	Classes
JHotDraw	177	5.2–5.4B2	13.5–36.3	164–489
ArgoUML	5525	0.9–0.20	99.5–159.5	801–2373
Eclipse-JDT	19750	1.0–3.0	205.5–534.4	2089–6949

Dependencies are computed in a conservative way considering, for each class: (i) attribute types for the class and for the parent classes, i.e., associations, (ii) method parameter types and local variable types, (iii) casting and constructor invocations. Finally, dependencies are propagated to subclasses. This quickly provides a superset of possible dependencies from/to the pattern, although more precise approaches can be used to refine the dependency set [21].

4. EMPIRICAL STUDY

This section describes the empirical study context and defines its research questions.

4.1 Context description

We selected three open-source systems, *JHotDraw*, *ArgoUML*, and *Eclipse-JDT*, which can be classified as small, medium, and large systems, respectively. We extracted from such systems only the HEAD development trunk (i.e., excluding branches) by using the time-window heuristic [10]. Table 1 reports for each system, the number of extracted snapshots, the range of analyzed releases, the number of non-commented lines of code (KNLOC), and the number of classes (excluding anonymous classes). Table 2 reports for each design pattern the minimum and maximum number of instances identified in all the analyzed software system releases. Abbreviations of Design Pattern column refers to the following: FM = Factory Method, P=Prototype, S=Singleton, AC=Adapter-Command, C=Composite, D= Decorator, O=Observer, SS=State-Strategy, TM= Template Method, V=Visitor.

JHotDraw[3] is a Java framework for drawing 2D graphics. The project started in October 2000 with the main purpose of showing the Design Pattern Programming in a real context. We extracted a total of 177 snapshots from release 5.2 to release 5.4 BETA2, in the time interval between March 2001 and February 2004. In that interval the size of the system grew almost linearly from 13.5 KNLOC at release 5.2 to 36.5 KNLOC at release 5.4 BETA2. With a similar trend the number of detected design patterns grew from 93 at release 5.2 to 158 at release 5.4 BETA2. Such design patterns were never deleted from the system.

ArgoUML[4] is an open source UML modeling tool with advanced software design features, such as reverse engineering and code generation. The project started in September 2000 and is still active. The number of active developers was initially 5 and has grown rapidly reaching a peak of 23 active

[3]*http://www.jhotdraw.org*
[4]*http://argouml.tigris.org*

Table 2: Detected design patterns

System	Design Patterns										Total
	Creational			Structural			Behavioral				
	FM	P	S	AC	C	D	O	SS	TM	V	
JHotDraw	2–9	4–14	2–7	18–24	1–2	3–11	6–10	43–78	4–6	1	93–162
ArgoUML	0–7	0–9	76–174	7–27	1	6–15	5–9	5–48	12–33	0	117–256
Eclipse-JDT	47–54	0–6	21–45	106–270	1–4	16–25	11–33	168–242	63–109	0–71	564–831

developers in September 2002. We considered an interval of observation ranging from September, 2000 (release 0.9.0) to December 2005 (release 0.20 ALPHA 4), where 58 releases have been produced including alpha, beta, and release candidates. In such interval we extracted 5525 snapshots. KNLOC has grown from 99.5 to 159.5 in the same interval. Although the total number of detected design patterns grew from 117 to 255, for some of them — Factory-Methods, Singletons, and Adapter-Commands — such a number oscillates over the time.

Eclipse-JDT is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse[5] platform. *Eclipse-JDT*, as any Eclipse project is organized into release, stable, integration, and nightly builds. The number of CVS accounts is 47, which is probably the number of different developers that contribute to the project. We extracted 19750 snapshots in the time interval between November 2001 and June 2004, when Eclipse 3.0 was released. In this time interval, KNLOC grew almost linearly from 205.5 to 449.4. The number of detected design patterns reaches the maximum of 831 at release 2.1. In such a release, the patterns Factory-Methods, Prototypes, Adapter-Commands, Composites, Decorators, Observers, and Visitors reached their maximum number of detected instances. Instead, the Singletons, State-Strategies, and Template-Methods, grew almost linearly.

4.2 Research Questions

This empirical study aims at answering the following research questions:

- *RQ1: how frequently do patterns change across releases?* This research question analyzes the change frequency of a pattern across snapshots, investigating whether some particular patterns tend to change more frequently than others we have studied.
- *RQ2: what kind of changes are different patterns subject to?* This research question analyzes whether some patterns are more prone to a particular kind of change (method addition or removal, attribute addition or removal, subclass addition or removal, or method implementation change) than others we have studied.
- *RQ3: how much source code co-changes with patterns?* This research question analyzes whether there is a relationship between the pattern co-change in terms of classes and source code lines, and the pattern type. It also considers the relationships between the kind of change in the pattern and the amount of co-change. Moreover, it restricts the analysis to co-changed classes having a dependency relationship with the pattern.
- *RQ4: what is the relationship between pattern targets changes and pattern client changes?* This question investigates whether some patterns make the pattern clients more resilient to changes performed in pattern targets.

[5] *http://www.eclipse.org*

5. RESULTS

This section reports results of analyses of data collected on the three systems according to the process described in Section 3, with the aim of answering the research questions formulated in Section 4.2. Further analyses are reported in a longer technical report[6].

5.1 RQ1: How frequently do patterns change across releases?

To answer *RQ1*, we counted for each pattern the number of snapshots where at least a class belonging to the pattern changed, and divided it by the total number of snapshots. Overall, snapshots involving pattern changes were 94 out of 177 for JHotDraw (53%), 1923 out of 5525 for ArgoUML (35%), and 5606 out of 19750 for Eclipse-JDT (28%). Change frequencies for each design pattern detected in the three systems are reported in Figure 1.

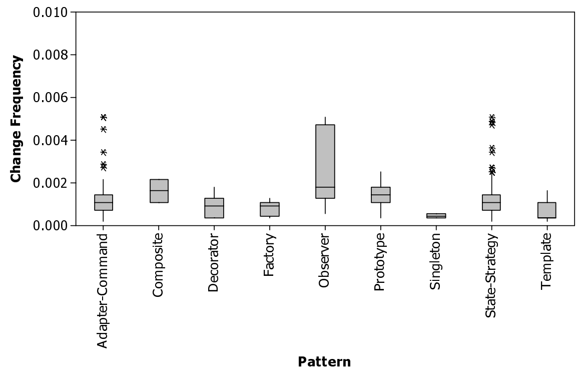
The Analysis of Variance (ANOVA) indicates significant differences among different patterns for JHotDraw (p-value =0.0003), for ArgoUML (p-value =0.0002), and for Eclipse-JDT (p-value $=2.3\cdot10^{-12}$), showing that some patterns change more frequently than others we have studied. To this aim, we compared highly changed patterns against other ones by using a Mann-Whitney two-tailed test, using the Bonferroni correction[7] to determine the statistical significance of a result where multiple tests were needed. It was found that, for example, for JHotDraw Observers changed more frequently than other patterns (p-value=0.001). Visitors in JHotDraw did not change, since they were used to navigate a picture, feature that remained unchanged across releases thanks to the use of a Composite pattern. For ArgoUML Adapters/Commands changed more frequently than other patterns (p-value$=2.9\cdot10^{-6}$). For Eclipse-JDT, Visitors changed more frequently than others (p-value$=7.9\cdot10^{-15}$).

5.1.1 Discussion

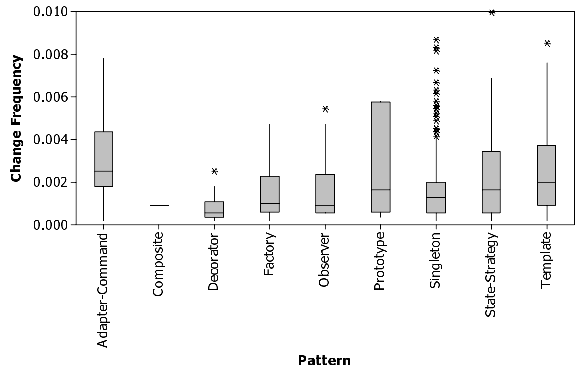
Overall, results indicate that the most frequently changed patterns are: Observers in JHotDraw, Commands in ArgoUML, and Visitors in Eclipse-JDT. As shown in Table 3, these patterns play a very important role for the particular application. In JHotDraw Observers are used to implement the notification-listening mechanism that manages the updating of figure visualization after changes. This, in general, applies for any application with a Model/View/Controller user interface. Gamma *et al.* [11] report among known uses of Observer:

[6] *http://www.rcost.unisannio.it/mdipenta/pattern-evol-tr-07.pdf*

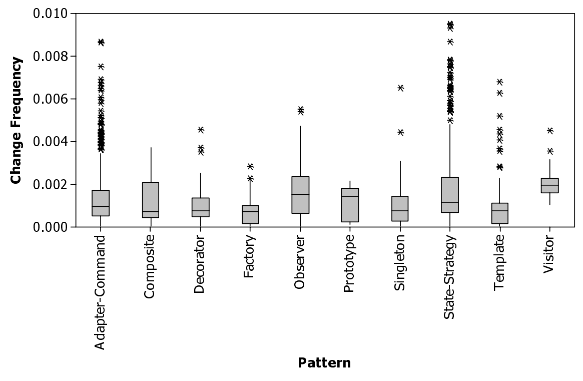
[7] Dividing the significance level α by the number of tests performed.



(a) JHotDraw

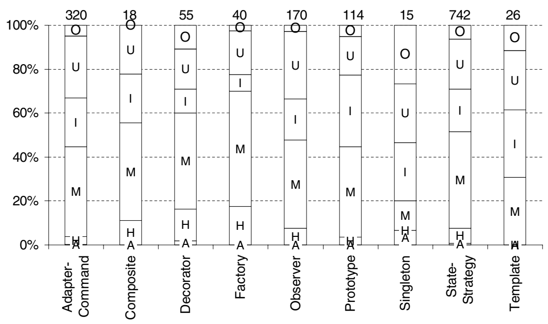


(b) ArgoUML

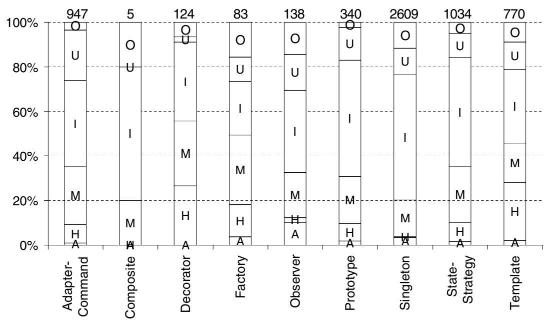


(c) Eclipse-JDT

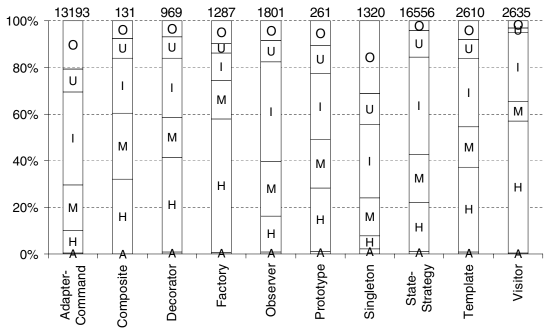
Figure 1: Change frequency of different design patterns.



(a) JHotDraw



(b) ArgoUML



(c) Eclipse-JDT

Figure 2: Kind of change for different patterns.

"The first and perhaps best-known example of Observer pattern appears in Smalltalk Model/View/ Controller (MVC), the user interface framework in the Smalltalk environment."

In ArgoUML, Commands change to support the execution — from user interface menus — of new modeling features, while new Decorator concrete classes are added to support new code generation features. Adapters are used to adapt interfaces of the UML metamodel to visualize them in Swing tables (a known usage according to Gamma *et al.*). Finally, in Eclipse-JDT Visitors are used to support navigation of Java Abstract Syntax Trees (AST), Concrete Visitors support new code artifact search or re-factoring activities. This can also be considered a general claim, since also other source code analysis and manipulation frameworks (e.g., JavaCC) make use of visitors for any AST navigation purpose. According to Gamma *et al.*:

"It is used primarly for algorithms that analyze source code. It is not used for code generation or pretty-printing, although it could be."

Overall, results related to research question *RQ1* brings to the conclusion that patterns change more frequently when they play a crucial role for the intent of the application.

Table 3: Most frequently changed patterns

	JHotDraw	ArgoUML	Eclipse-JDT
Patterns	Observer, Composite	Adapter-Command, Decorator, Factory	Visitor
Used for	Model View Controller of Draws, Handling composite figures	Adapting and decorating UML objects to different views - Execute menu actions	Visiting Java AST
Purpose of change	Adding new draw elements	Adding new menu actions and presentations	Adding new code analyses

5.2 RQ2: What kind of changes are different patterns subject to?

To answer *RQ2* we analyzed, for each pattern, the kind of changes that were performed on its classes across releases. Figure 2 shows, in percentage, the kind of change happened for different patterns, distinguishing between attribute changes (A), addition or removal of subclasses (H), method interface changes (M), method implementation changes (I), changes involving more than one of the aforementioned characteristics (U), and other kind of changes, e.g., comments or style issues (O). On top of each bar it is indicated the total number of changes for each pattern type.

For JHotDraw, it can be noted that, in most cases, method changes predominate over other changes. Proportion test did not indicate patterns having a significantly higher proportion in terms of method interface changes (p-value=0.28), while it was the case for method implementation (p-value =0.008), with higher proportions for Prototype (prop=0.32) and Template Method (prop=0.31). Attributes changed in proportion more on Singleton (prop=0.067) than on other patterns considered in this study (p-value=0.06), while subclassing changed more for Composite (prop=0.14) and Decorator (prop=0.17) (p-value = 0.001).

For ArgoUML changes in method implementations predominated, followed by changes in method interfaces. Both significantly varied in proportion across patterns (p-value$<2.2 \cdot 10^{-16}$ for implementations and $1.5 \cdot 10^{-12}$ for interfaces). Implementations changed in proportion more on Composite (prop=0.60) — although this pattern was only involved in 5 co-changes — and Singleton (prop=0.56), while interfaces changed more on Decorator (prop=0.29) and Factory (prop=0.31). Changes for attributes varied significantly (p-value=$5.2 \cdot 10^{-10}$); in particular Observer exhibited a relatively higher proportion (prop=0.10). Changes also varied significantly for sub-classing (p-value $<2.2 \cdot 10^{-16}$); in this case the proportion was higher for Decorator (prop=0.26) and Template Method (prop=0.26).

Finally, for Eclipse-JDT changes in sub-classing predominated, followed by changes in method implementations. Changes for method implementations and interfaces significantly varied in proportion across patterns (p-value$<2.2 \cdot 10^{-16}$ in both cases). Implementations changed with higher proportions for Observer (prop=0.43), State-Strategy (prop=0.42) and Adapter (prop=0.40), while interfaces changed more for Composite (prop=0.28). Attribute changes varied in proportion across patterns (p-value =$4.1 \cdot 10^{-13}$) with a relatively high proportion for Singleton (prop=0.02). Finally, changes for sub-classing also varied significantly (p-value $<2.2 \cdot 10^{-16}$) with higher proportions for Factory (prop=0.57) and Visitor (prop=0.57), and relatively high for Decorator (prop=0.40).

5.2.1 Discussion

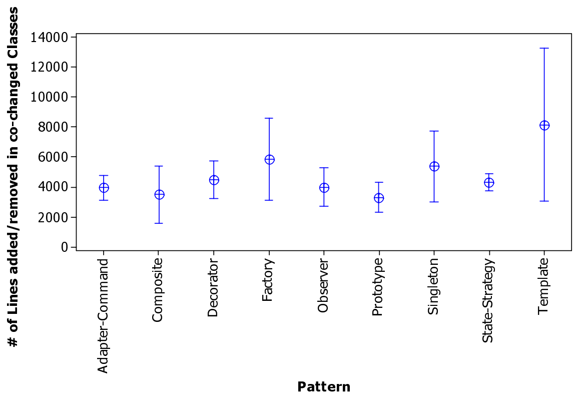
As for RQ1, also here the application size and domain influences the kind of change patterns undergo. Not only different patterns tend to undergo different kinds of changes, but, noticeably, the same pattern tend to change differently in the three systems. A small system originally conceived as a "design exercise" (JHotDraw) tends to exhibit more changes in pattern method interfaces because of the addition of new features. Being the system very small, developers did not put effort in avoiding changes in pattern interfaces, since the impact of such changes was fairly limited.

Larger systems (e.g., ArgoUML and Eclipse-JDT) tend to exhibit, in proportion, more changes in pattern implementations, either due to enhancement or corrective maintenance, and less changes in method interfaces. Overall, this highlights a correct usages of patterns, in a context where developers attempts whenever possible to design fairly stable pattern interfaces, so that evolution happens by means of (i) changes in method implementations or (ii) by means of subclassing, as it expecially happens in Eclipse-JDT. This is visible in particular for the Visitor pattern, widely used in Eclipse for source code analysis purpose. The data structure (i.e., the Java AST) was almost stable, while analysis features (i.e., new Visitors) were often added/changed. One possible cause of the different pattern evolution for ArgoUML and Eclipse can be found in their different architecture. Eclipse is a framework that contributors evolve by adding plugins, that is made by extending a Java interface. This largely affects design patterns such as Composite, Decorator, Factory, and Visitor. Instead, ArgoUML is a monolithic application, where the introduction of new features happens with the addition of new classes or methods, while enhancements or corrective maintenance activities very often are confined within method implementations.

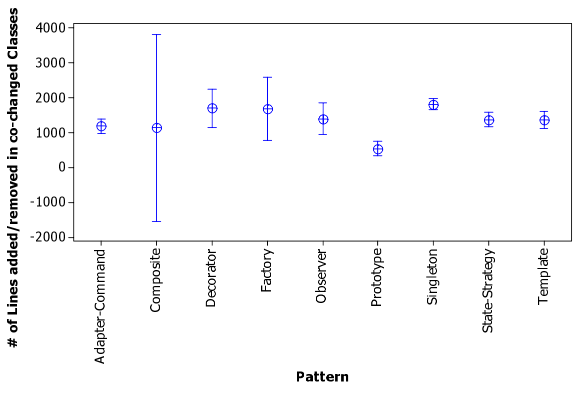
5.3 RQ3: How much source code co-changes with patterns?

This section analyzes the amount of code co-changed with patterns, analyzing differences for patterns of different type and different purpose. The analysis has been performed at two different level of details, i.e., considering the number of classes and the number of source code lines co-changed with the pattern. Results are only shown for the second case, explaining differences where present. Figure 3 shows, for the three systems and for each pattern type, boxplots of the number of source code lines co-changed with the pattern.

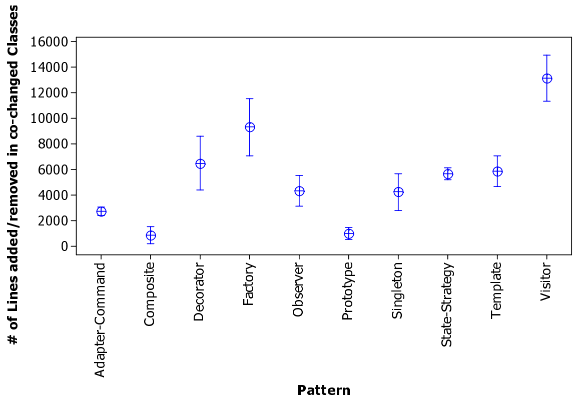
For JHotDraw, no significant difference was found among different patterns (p-value=0.17). For ArgoUML, the difference was significant (p-value=$3.2 \cdot 10^{-9}$): Singletons had more co-change than other patterns we considered (p-value =$5.8 \cdot 10^{-13}$), while Prototypes had less co-change than others (p-value= $3.0 \cdot 10^{-15}$). For Eclipse-JDT, a significant difference was found among different patterns (p-value $<2.2 \cdot 10^{-16}$). Visitors had, in particular, a higher number of co-changed lines than other patterns (p-value$< 2.2 \cdot 10^{-16}$). Besides the number of lines co-changed with patterns, we also analyzed the number of classes co-changed with patterns, obtaining results consistent with the above findings.



(a) JHotDraw



(b) ArgoUML



(c) Eclipse-JDT

Figure 3: Co-changed lines by pattern

While co-changes provide a rough indication of the impact of pattern changes, they do not indicate to what extent changes in patterns directly affect classes having a dependency on the pattern. For JHotDraw, it was found that the Composite had a significantly larger number of co-changed client classes (p-value=0.02), due to its role in handling features for composite shapes [11]. For ArgoUML, the number of client class lines co-changed significantly varied among different patterns (p-value $<2.2\cdot10^{-16}$). In particular, it was found that the Factory had more co-changed lines (and classes) than other patterns (p-value =0.0004), while Singleton had less (p-value$<2.2\cdot10^{-16}$). Together with Command, Factory was widely used in ArgoUML to create actions responsible of creating different kind of UML diagrams. For Eclipse-JDT, both the number of client classes and the number of lines added or removed significantly varied for different patterns (p-value$<2.2\cdot10^{-16}$). In particular, it was found that Visitor had a significantly higher number of classes and lines co-changed than other patterns (p-value$<2.2\cdot10^{-16}$). Noticeably, the mean number of co-changed client classes for Visitors (6) was higher than for other patterns (between 0 and 1), as well as the number of lines (mean=1500 while other patterns had a mean below 220).

5.3.1 *Discussion*

By analyzing the co-change by pattern, we found a result consistent to what found for *RQ1*: patterns crucial for the intent of the application are part of larger co-changing sets than for other patterns considered in this study. This indicates that, when a pattern is crucial for the application purpose, it tends to change together with a large number of other classes.

In JHotDraw, Observers and Composites have the highest number of co-changed classes, while in ArgoUML both Adapter-Commands and Decorators co-change with a large number of other classes. For Eclipse-JDT there is apparently a predominance of the Factory pattern, widely used in the Eclipse platform. The Visitors had less co-changes, although they exhibited a higher impact on co-changed classes, since the average number of lines changed per class (20) is higher than for other patterns (e.g., 9 for the Factory).

In some cases, since the co-change involves a client class of the pattern the co-change indicates how changes in the pattern impact on other classes. Also in this case, changes of patterns playing an important role for the application (e.g., Composite for JHotDraw, Factory for ArgoUML, and Visitor for Eclipse-JDT) occur together with a large number of changes in their client classes, meaning that such patterns are widely accessed throughout many software system classes.

5.4 RQ4: What is the relationship between pattern targets changes and pattern client changes?

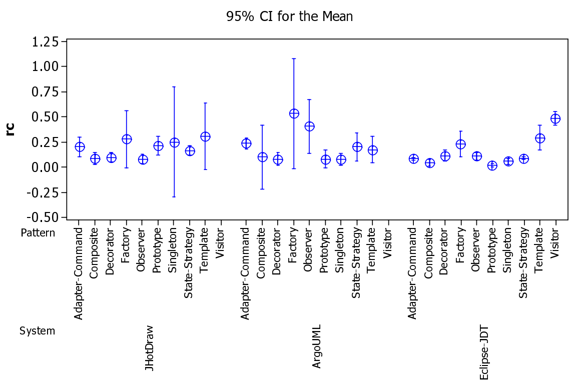
Finally we analyzed, for each pattern, the ratios:

$$r_c = \frac{\#\, of client\ classes\ co-changed}{\#\ of\ target\ classes\ co-changed}$$

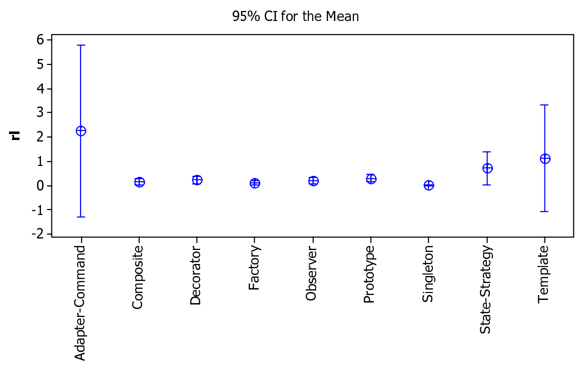
$$r_l = \frac{\#\, of client\ class\ lines\ co-changed}{\#\ of\ target\ class\ lines\ co-changed}$$

for all the cases where the target change set — either classes or lines — was not empty. Ratios smaller than one indicate that a pattern attenuates the impact on clients for changes happened in targets, while ratios higher than one indicate that a pattern does not make the system resilient to such a change. Interval plots are shown in Figure 4-a for classes and in Figure 4-b,c,d for lines.

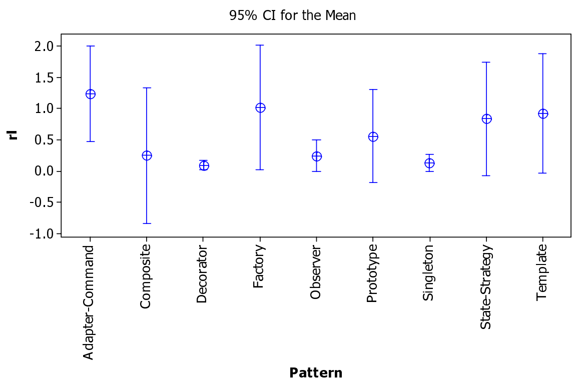
For JHotDraw, ANOVA did not indicate any significant difference among patterns for both r_c (p-value=0.39) and



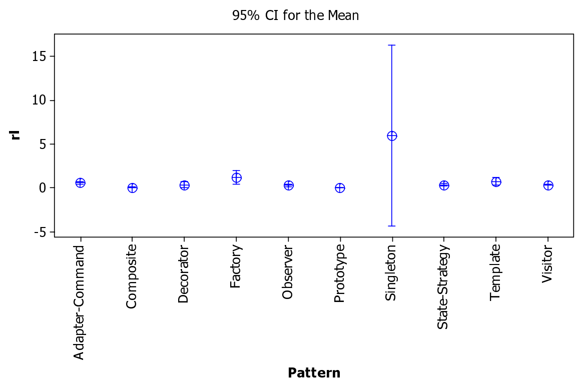
(a) r_c



(b) r_l (JHotDraw)



(c) r_l (ArgoUML)



(d) r_l (Eclipse-JDT)

Figure 4: Client / Target Ratios (r_c and r_l)

r_l (p-value=0.93). However, it was found that Adapter-Command had a significantly higher r_l than other patterns we have studied (p-value=0.02) and a mean $r_l = 2$ while others had $r_l < 1$. For ArgoUML, ANOVA did not indicate any significant difference for r_c (p-value=0.07) and r_l (p-value=0.7), although it was found that Adapter-Command, Factory, State-Strategy and Template Method exhibited a significantly higher r_l than other pattern ($r_l \sim 1$ while other had $r_l \leq 0.5$). For Eclipse-JDT, a significant difference was found on r_c (p-value$<2.2 \cdot 10^{-16}$). In particular, Visitor had a higher r_c than other patterns (p-value$<2.2 \cdot 10^{-16}$); such a difference was however not visible for r_l, indicating that, although changes involving Visitor targets impact many client classes, the impact is limited to a small number of source code lines — i.e., the code used to accept the Visitor on the data structure. Singleton had a higher mean r_l than other patterns, although its large variability made such a difference not statistically significant.

5.4.1 Discussion

In all the three systems we observed that, excepts for some specific patterns, the ratio between the amount of changes in client classes and target classes is very low ($r_c, r_l << 1$). Although no general conclusion can be drawn from this result, it can be noted that, when pattern targets changes, such a change often has a limited effect on pattern clients. One possible reason of this result is the capability many patterns have of making the system resilient to changes, as discussed by Gamma *et al.* [11]. In other words, some patterns avoid that changes happened in specific classes — the pattern targets — propagate on other classes, i.e., the pattern clients. One example of this capability is the Command pattern: when a new action has to be added, this causes the modification of a callback to the Receiver in the Concrete Command, without however affecting the client. For example, In ArgoUML 0.19.3, a new action *org.argouml.uml.ui.ActionRESequenceDiagram*, was introduced, affecting a set of State-Strategy and Command patterns belonging to *org.argouml.uml.reveng.java.Modeller*. Such a new feature caused the addition of just one instruction of a new action in the explorer popup menu.

By looking at results of *RQ2* and *RQ4*, it was also noted that, at least for ArgoUML and Eclipse-JDT r_c and r_l are signficantly correlated with the type of change made on the patterns (the Spearman correlation indicated a p-value of $< 2.2 \cdot 10^{-16}$ in all cases). As expected, when a pattern interface changes, this causes a high-level of co-change in pattern clients, and also high values for the r_c and r_l ratios. Changes in hierarchies are often propagated in client classes with a new or a different instantiation of pattern Concrete participants by using constructors or casting, i.e., a small and focused change. This is confirmed by low r_c

and, above all, very low r_l. For example, this happens for Visitors in Eclipse-JDT: this pattern often undergoes to hierarchy changes. This however, results in a low r_l, while r_c is still relatively high, if compared with other patterns detected for the same system. This because the use of new Visitors requires adding new Visitor *accept* crosscutting in some classes, however the amount of code to be added is very small.

6. THREATS TO VALIDITY

This section discusses threats to validity that can affect the results reported in Section 5, following a well-known template for case studies [25]. Regarding *construct validity*, threats can be due to the measurement performed, in particular related to design pattern identification into source code, analysis of dependencies, and the use of change sets to determine impacts. For which concerns design pattern identification, we rely on Tsantalis *et al.* work, and we are aware that our results can be influenced by its performances in terms of precision and recall. Nevertheless, precision showed in reference [23] is quite high — although authors assessed it only considering pattern conformance to the cliché and not the developers' intent (as Antoniol *et al.* [1] did, although their tool only discovers a limited set of structural patterns). Our results can still be affected by the presence of *false negatives*, i.e., by a low recall exhibited by the design pattern detection tool. At least for JHotDraw (where patterns are documented) the tool exhibits a very high recall as shown in [23]. Nevertheless, in case pattern implementations are variants of the simple cliché defined by Gamma *et al.*, some patterns may be missed during the detection phase. Although the sample of detected patterns can be considered large enough to claim our conclusions, further investigations aimed at assessing to what extent the detection tool performace assess our results are needed. Finally although Tsantalis *et al.* applied in their work the tool on small systems, we experienced no particular scalability issue when applying it to larger systems such as Eclipse-JDT, although the analysis of each release took about 8 hours.

Regarding dependency analysis, as discussed in Section 3, we consider a conservative set of classes for both pattern client and targets. More precise analyses could have restricted this set and made our findings less pessimistic. Finally, we analyzed change sets as a way to assess the impact of pattern change. Clearly, more sophisticated impact analysis techniques are available in literature [2], although change-sets and bug issues are used for this purpose [6, 27]. Moreover, we made this analysis more precise by considering the intersection of change sets with pattern client and targets.

Threats to *internal validity* did not affect this particular kind of study, being it an explorative study [25].

Threats to *external validity* are related to what extent we can generalize our findings. We considered three software systems, differing for their domain and size, and obtained some common findings and some results peculiar to each system. Nevertheless, it would be desirable to analyze further systems — also developed in different programming languages — to draw more general conclusions. Finally, we considered a subset of 12 out of the 23 patterns from the Gamma *et al.* catalogue, i.e., those detected by the adopted tool.

Regarding *reliability validity*, the source code of the three systems is publicly available, as well as the design pattern detection tool; the way our analyses were performed is described in detail in Section 3. Statistical conclusions were supported by proper tests, ANOVA for analyzing multiple means, Mann-Whitney for unpaired comparisons of two means and proportion test for comparing proportions.

7. CONCLUSIONS

This paper reported an empirical study on the evolution of design patterns in three Java software systems. Results indicate that the pattern change frequency and amount of co-change does not depend on the pattern type, but rather on the role played by the pattern to support the application features. Patterns are often changed either in their implementation or by adding subclasses or changing method interfaces: the latter however causes a higher co-change on client classes.

Results suggest that developers should carefully consider pattern usage when this support crucial features of the application. Such patterns will likely undergo frequent changes and involved in large maintenance activities, that would be highly affected by wrong pattern choices.

The analysis process proposed in this paper poses the basis for further studies aimed at increasing the external validity of results leading to more general conclusions. Future work also aims at improving the accuracy of results, especially considering the fact that we rely on design pattern detection techniques affected by both false positives and false negatives. Although large systems with documented design patterns scarcely available, we plan to rely on statistical sampling technique to measure design pattern detection precision and recall. Finally, we plan to use more accurate impact analysis or dependency analysis techniques.

8. ACKNOWLEDGMENTS

The work described in this paper is 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. We would like to thank the anonymous reviewers. Their comments greatly helped to improve the paper. Also, we would like to aknowledge the authors of the design pattern recovery tool used in this empirical study, N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis.

9. 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] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 1993), Montréal, Quebec, Canada*, pages 292–301. IEEE Computer Society, 1993.

[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 and L. Cerulo. How crosscutting concerns evolve in JHotDraw. In *STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 65–73. IEEE Computer Society, 2005.

[6] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *11th IEEE International Software Metrics Symposium (METRICS 2005)*, pages 29–38. IEEE Computer Society, 2005.

[7] 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.

[8] 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.

[9] B. Eckel. *Thinking in Patterns with Java* http://www.mindview.net/Books/TIPatterns/ *Last accessed March, 11 2007*. Mindview Inc., 2005.

[10] 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.

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

[12] 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.

[13] J. Gil and I. Maman. Micro patterns in java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 97–116. ACM, 2005.

[14] Y.-G. Guéhéneuc, H. A. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *11th Working Conference on Reverse Engineering (WCRE 2004), 8-12 November 2004, Delft, The Netherlands*, pages 172–181, 2004.

[15] 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.

[16] 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.

[17] S. Kim, K. Pan, and E. J. J. Whitehead. Micro pattern evolution. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, pages 40–46. ACM, 2006.

[18] 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.

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

[20] 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.

[21] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.