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 resilence 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 resilent to changes.

Categories and Subject Descriptors

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

Keywords

Design patterns, Software Evolution, Mining Software Repositories, Empirical Software Engineering, Object-Oriented Software Design

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 reference [11] or reference [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 [22]. 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 to identify 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, 14, 17, 21] 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 tool and approach presented by Tsantalis *et al.* [21]. 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. Section 2 details the process to extract the information needed to perform the empirical study. Section 3 describes the empirical study context and research questions. Section 4 reports the case study results. Section 5 summarizes the main findings of this study and discusses its threats to validity.

After a review of the literature in Section 6, Section 7 concludes the paper and outlines directions for future work.

2. 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.

2.1 Step 1: Detecting Design patterns on releases

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.* in [21]. 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.

2.2 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 a 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.

2.3 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* (*MT*s) (also known as Change Sets), representing the logical changes performed by a developer in terms of added, deleted, and changed source code lines. *MT*s 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 [25].

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

2.4 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.

2.5 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 2.4 to identify the set of classes that co-change with the pattern while not directly belonging to the pattern. Let $C_i \equiv \{c_{1,i}, \ldots c_{n,i}\}$ 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 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.

Dependencies are computed in a conservative way considering, for each class: (i) attribute types for the class and

[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

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 [19].

3. EMPIRICAL STUDY

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

3.1 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. the variation range across 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 to shown the Design Pattern Programming in a real context. There is one active developer that usually controls the commit of new changes coming from volunteers. 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 design patterns detected across releases 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 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 design patterns detected among releases grew from 117 to 255, for some of them — Factory-Methods, Singletons, and Adapter-Commands — the 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 such time interval, KNLOC grew almost linearly from 205.5 to 449.4. The number of design patterns detected reaches the maximum of 831 at release 2.1. In such 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.

3.2 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, or pattern having a specific purpose, tend to change more frequently than others.
- *RQ2: to what kind of change are different pattern 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.
- *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 and purpose. 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: which 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.

4. RESULTS

This section reports results of analyses of data collected on the three systems according to the process described in Section 2, with the aim of answering the research questions formulated in Section 3.2.

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

4.1 How 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 (p-value $=2.3 \cdot 10^{-12}$), showing that some patterns change more frequently than others. To this aim, we compared highly changed patterns against other ones by using a Mann-Whitney two-tailed test, using the Bonferroni correction[6] 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=$2.9 \cdot 10^{-6}$). For Eclipse-JDT, Visitors changed more frequently than others (p-value=$7.9 \cdot 10^{-15}$).

Comparing change frequency by purpose, we found significant differences for Eclipse-JDT (p-value=$1.9 \cdot 10^{-6}$), while no significant difference was found for JHotDraw (p-value =0.35) and ArgoUML (p-value=0.38). In the case of Eclipse-JDT, we found that Structural patterns changed significantly more than Behavioral patterns (p-value=0.001) and than Creational patterns (p-value=$1.46 \cdot 10^{-5}$).

4.2 To what kind of change are different pattern 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 (proportion=0.32) and Template (0.31). Attributes changed in proportion more on Singleton (proportion=0.067) than on other patterns (p-value=0.06), while sub-classing changed more for Composite (0.14) and Decorator (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 (0.60) — although this pattern was only involved in 5 co-changes — and Singleton (0.56), while interfaces changed more on Decorator (0.29) and Factory (0.31). Changes for attributes varied significantly (p-value=$5.2 \cdot 10^{-10}$) — Observer exhibited a relatively higher proportion (0.10) — as well as sub-classing changes (p-value <$2.2 \cdot 10^{-16}$) — in this case the proportion was higher for Decorator (0.26) and Template method (0.26).

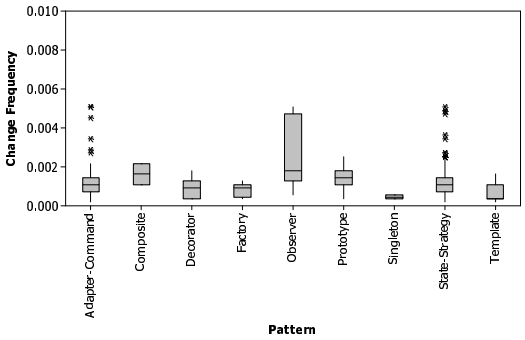
Finally, for Eclipse-JDT changes in sub-classing predominated, followed by changes in method implementations. Changes for method implementations and interfaces significantly varied in proportion across patterns (p-value<$2.2 \cdot 10^{-16}$ in both cases). Implementations changed with higher proportions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

4.3 How much source code co-changes with patterns?

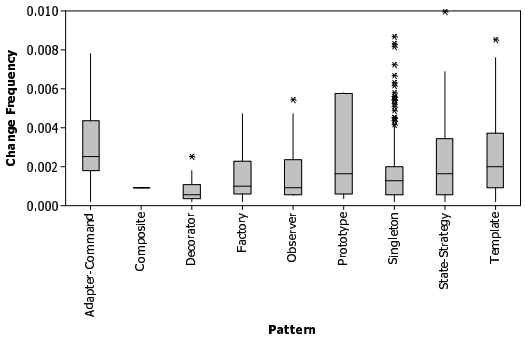
This section analyzes the amount of code co-changed with patterns, analyzing differences for patterns of different type and different purpose. The analysis has been performed at two different level of details, i.e., considering the number of classes and the number of source code lines co-changed with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

Table 3 shows descriptive statistics of the number of source code lines co-changed with the pattern, classified by pattern purpose. For JHotDraw, no significant difference was found among different purposes (p-value=0.69) and among different patterns (p-value=0.17). For ArgoUML, the difference was significant among different purposes (p-value=0.0005) and among different patterns (p-value=$3.2 \cdot 10^{-9}$). Regard-

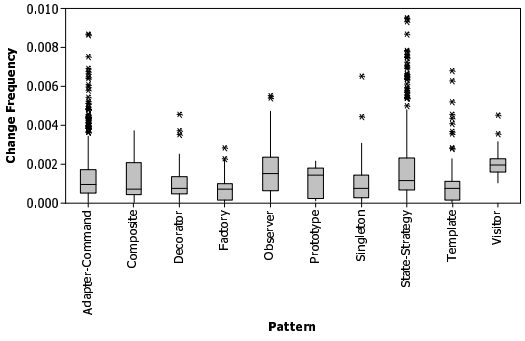
[6]Dividing the significance level α by the number of tests performed.



(a) JHotDraw

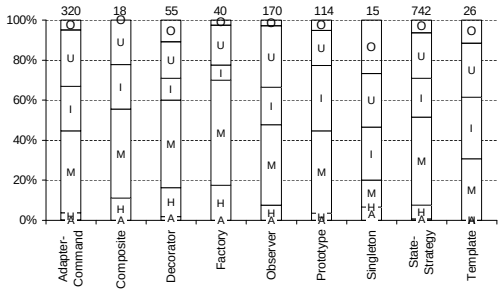


(b) ArgoUML

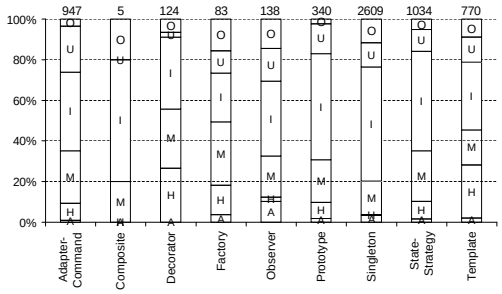


(c) Eclipse-JDT

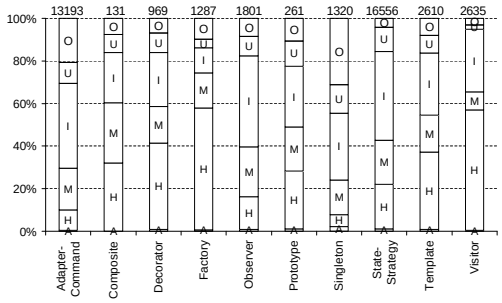
Figure 1: Change frequency of different design patterns.



(a) JHotDraw



(b) ArgoUML



(c) Eclipse-JDT

Figure 2: Kind of change for different patterns.

Table 3: Number of lines added/removed in cochange by category

JHotDraw			
	Structural	Behavioral	Creational
Range	0–48152	0–48153	0–48152
Avg.	4015	4386	4096
Median	837	947	1701
Std. dev.	6989	8156	6265
ArgoUML			
	Structural	Behavioral	Creational
Range	0–17902	0–30437	0–17915
Avg.	1241	1363	1666
Median	133	108	150
Std. dev.	3169	3336	3819
Eclipse-JDT			
	Structural	Behavioral	Creational
Range	0–233544	0–233493	0–233595
Avg.	2963	6432	6211
Median	42	49	42
Std. dev.	21564	32905	32577

Table 4: Number of lines added/removed in co-changed client classes for patterns having different purpose

JHotDraw			
	Structural	Behavioral	Creational
Range	0–1931	0–1931	0–812
Avg.	82	93	101
Median	0	0	0
Std. dev.	247	250	215
ArgoUML			
	Structural	Behavioral	Creational
Range	0–4664	0–1830	0–11377
Avg.	119	20	12
Median	0	0	0
Std. dev.	598	139	233
Eclipse-JDT			
	Structural	Behavioral	Creational
Range	0–61676	0–68997	0–19148
Avg.	78	334	134
Median	0	0	0
Std. dev.	1139	3498	1021

ing pattern purposes, it was found that Creational patterns had a significantly higher number of co-changed lines than Behavioral patterns (p-value=0.0008). A difference was found between Creational and Structural patterns, however it was not significant (p-value=0.19). Singletons had more co-change than other patterns (p-value $=5.8{\cdot}10^{-13}$), while Prototype had less co-change than other patterns (p-value=$3.0{\cdot}10^{-15}$). For Eclipse-JDT, a significant difference was found among purposes and among pattern types (p-value $< 2.2{\cdot}10^{-16}$ in both cases). Structural patterns exhibited less co-change than Behavioral (p-value $< 2.2{\cdot}10^{-16}$) and Creational patterns (p-value=$3.2{\cdot}10^{-7}$). Visitors had, in particular, a higher number of co-changed lines than other patterns (p-value$< 2.2{\cdot}10^{-16}$). All the results were confirmed analyzing the number of classes co-changed with the pattern where, however, the difference between Creational patterns and other patterns was more evident — and significant — for Eclipse-JDT (p-value$< 2.2{\cdot}10^{-16}$)

While co-changes provide a rough indication of the impact of pattern changes, it does not indicate to what extent changes in patterns directly affected classes having a dependency on the pattern. Table 4 shows descriptive statistics of the number of lines added or removed for client classes co-changed with the pattern. The table highlights how only a small portion of co-change is related to pattern client classes, indicating that patterns are contextually modified with other parts of the system. For JHotDraw, comparisons by purpose and patterns confirmed the absence of significant differences both in terms of classes and lines co-changed. It was however 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 pattern purposes (p-value$<2.2{\cdot}10^{-16}$), indicating in particular that Structural patterns had more co-changed client class lines than others (p-value$<2.2{\cdot}10^{-16}$). Finally, both the number of client classes and the number of lines added or removed significantly varied for different purposes (p-value$<2.2{\cdot}10^{-16}$), indicating that Behavioral patterns had more co-changed client classes and lines than other patterns (p-value$<2.2{\cdot}10^{-16}$). In particular, the Visitor had a significantly higher number of classes and lines co-changed than other patterns (p-value$<2.2{\cdot}10^{-16}$). 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).

4.4 What is the relationship between pattern targets changes and pattern client changes?

Finally we analyzed, for each pattern, the ratios:

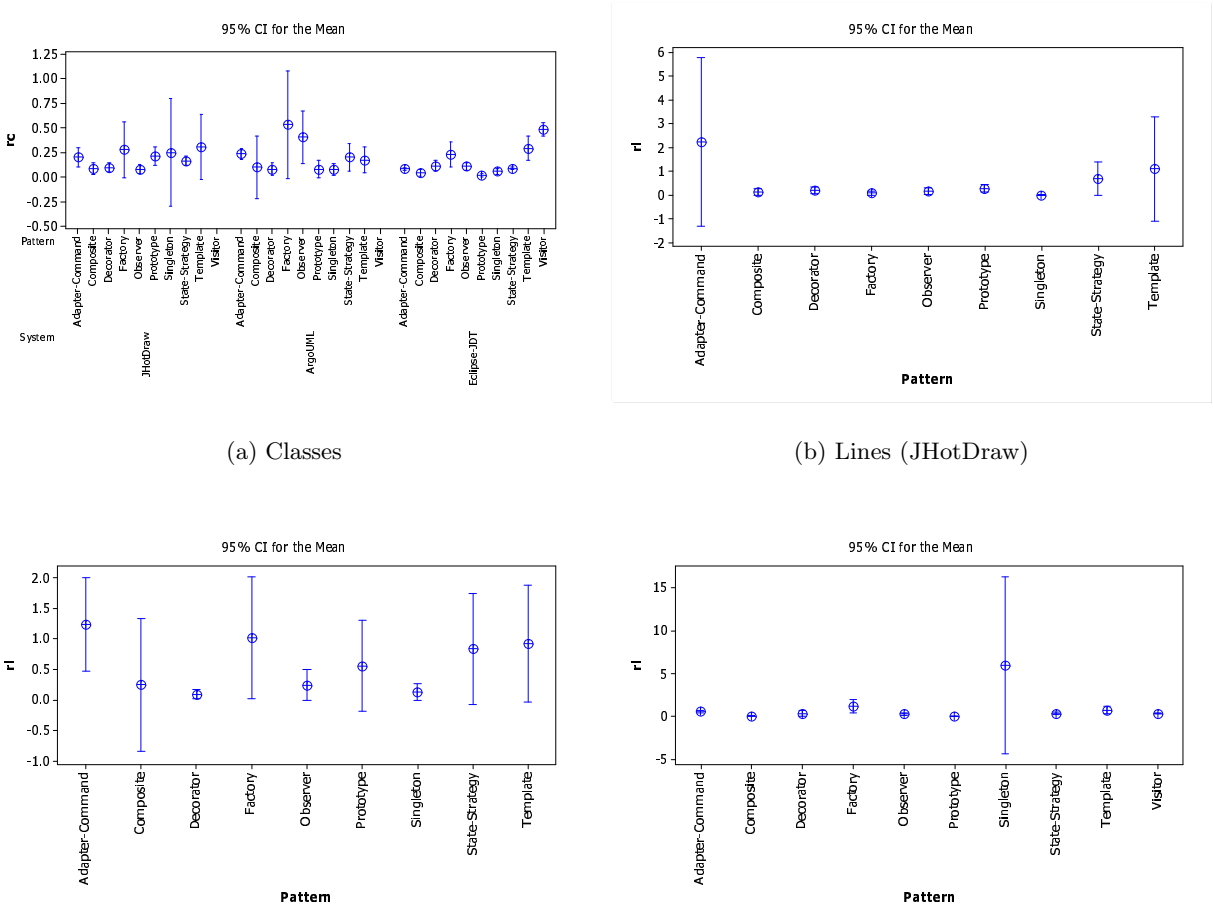
$$r_c = \frac{\#\ of client\ classes\ co-changed}{\#\ of\ target\ classes\ co-changed}$$

and

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

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 3-a for classes and in Figure 3-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 r_l (p-value=0.93). However, it was found that Adapter-Command had a significantly higher r_l than other patterns (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 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$<2.2{\cdot}10^{-16}$).



(a) Classes

(b) Lines (JHotDraw)

(c) Lines (ArgoUML)

(d) Lines (Eclipse-JDT)

Figure 3: Client / Target Ratios (r_c and r_l)

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. DISCUSSION

This section discusses quantitative results presented in Section 4, explaining them in terms of pattern *intent* and reasons of changes (summarized in Table 5), as discovered by inspecting the source code and analyzing change messages and system documentation.

R1: *Patterns change more frequently when they play a crucial role for the intent of the application.*

Results in Section 4.1 (*RQ1*) indicate that the most frequently changed patterns are: Observers in JHotDraw, Commands in ArgoUML, and Visitors in Eclipse-JDT. As shown in Table 5, 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. 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 artifact search or re-factoring activities. Results suggest to carefully choose patterns that support the application main features, since these patterns will likely be subject to frequent changes.

R2: *Creational patterns exhibit more co-change than other patterns.*

Table 5: Most frequently changed patterns

	JHotDraw	ArgoUML	Eclipse-JDT
Patterns	Observer, Composite	Adapter-Command, Decorator	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

Section 4.3 (*RQ3*) shows that, overall, Creational patterns have more co-change than other patterns. If we assume that co-changes are related to change impacts [24, 25], such finding states that creational patterns have the highest impact on other code. This can be explained by the different aim of the pattern purpose. Structural patterns are mainly used for decoupling purposes, thus their co-changes are supposed to be limited; changes in Behavioral pattern clients are often limited to a small number of source code lines related to pattern class instantiation or method invocations. Creational patterns help to make a system independent of how its object are created, composed, and represented. In this case, the co-change — containing classes using resources created by Creational patterns — is higher than for others. In general, co-change tends to be high when pattern clients are crosscutted across many classes [7].

R3: *Patterns crucial to the application role have more co-change than other patterns.*

By analyzing the co-change by pattern, we found that patterns more relevant for the intent of the application have more co-change than others. 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). Results also show that Singletons often have high co-changes, however this pattern has to be considered as a particular case, i.e., they just ensure that a class exists in only one instance. Although, as it will discussed below, patterns make the system resilient to changes, co-change is still high for patterns used to support crucial features. This because these patterns are changed in the context of wider maintenance activities, involving other parts of the system.

R4: *Patterns make clients resilient to changes.*

In all the three systems we observed (Section 4.4, *RQ4*) that the ratio between the amount of changes in client classes and target classes is very low ($r_c, r_l << 1$). Reasons for exceptions to that results are discussed in *R5*. When target classes change, clients changes proportionally lower. This confirms that, as indicated by Gamma *et al.* [11] patterns are generally capable of making clients resilient to changes. Just to mention a few concrete examples, in JHotDraw 5.3, a new storage format, *CH.ifa.draw.contrib.SVGStorageFormat*, was introduced affecting some patterns: Observers, Adapter-Commands, and State-Strategies. The only client affected was a sample application, *CH.ifa.draw.samples.javadraw.JavaDrawApp*, in which the new storage format has been added during the initialization of menus configuration. In ArgoUML 0.19.3, a new reverse engineering action, *org.argouml.uml.ui.ActionRESequenceDiagram*, has been introduced affecting a set of State-Strategy 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.

R5: *Changes to the pattern interfaces make clients less resilient to changes.*

By combining results from *RQ2* and *RQ4*, it is possible to analyze whether different types of change make the pattern more or less resilient to changes. The relationship between change type and both r_c and r_l was also highlighted by a significant Spearman correlation, at least for ArgoUML and Eclipse-JDT. We noted that changes causing a high impact are mainly method interface changes, or addition/deletion of methods, and addition/deletion of pattern subclasses (often referred as *concrete* participants to the pattern). For example, in JHotDraw 5.3 a new class, *CH.ifa.draw.framework.FigureEnumeration*, has been introduced to manage collection of figures, previously managed with *java.util.Vector*. This caused the modification of many pattern interfaces, impacted on client classes. In ArgoUML 0.16.*beta*1, the persistence mechanism has been re-factored by separating responsibilities. Such intervention caused the modification of some patterns interfaces, such as those related to Factory and State-Strategy patterns, affecting an high number of clients before managing persistency in a different way. Changes in hierarchies are often propagated in client classes with a new or a different instantiation of object by using constructor; thus 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 for the same system. This because adding new Visitors requires adding new Visitor *accept* crosscutting in some classes, however the amount of code to be added is very small. Modifications to method interfaces cause high r_c and r_l, indicating that the pattern is not very resilient to changes. This confirms what suggested by Gamma *et al.*, i.e., trying to make the pattern interface as stable as possible, and evolving the pattern by means of sub-classing.

5.1 Threats to Validity

This section discusses threats to validity that can affect the results reported in Section 4, following a well-known template for case studies [23]. 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 identifica-

tion, 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 [21] 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). Regarding dependency analysis, as discussed in Section 2, 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 recently change-sets and bug issues are extensively used for this purpose [6, 25]. 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 [23].

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 23 Gamma *et al.* patterns, although well distributed across purposes.

Regarding *reliability validity*, the source code of the three systems is publicly available, as well as the design pattern detection tool, and the way our analyses were performed is described in detail in Section 2. 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.

6. RELATED WORK

In our knowledge there are a very few papers aiming at analyzing empirically the evolution of design pattern. The research work most closely related to our was performed by Bieman *et al.* [3]. They analyze 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 consider differences between releases and do 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 [22] analyzes the corrective maintenance of a large commercial product over three years, comparing defect rates for classes that participated in design patterns versus those that do not participate. In contrast to our work, he focus only 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.* [20] 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 (case study instead of controlled experiment) 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 [24, 25], observe clone [12, 15] and crosscutting concern [5] evolution, identify crosscutting concerns [4, 7], detect of logical coupling between modules [10], find common error patterns [18], or identify fix inducing changes [16].

The study presented in this paper relies on a design pattern recovery approach proposed by Tsantalis *et al.* [21]. 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.* [17], by Antoniol *et al.* [1] or by Gueheneuc *et al.* [13]. The approach proposed by Costagliola *et al.* [8] identifies design patterns by visual language parsing, while the approach proposed by Heuzeroth *et al.* [14] 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.

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 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 reduces the pattern capability to improve the system resilience to changes. Results suggest that the pattern usage should be carefully considered, especially when the pattern should support crucial features of the application, and that pattern interfaces should be made as more stable as possible.

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, by considering and comparing alternative design pattern detection techniques, as well as more accurate impact analysis or dependency analysis techniques. Finally, we will further investigate on the relationship between design pattern evolution and other phenomena, such as the presence of crosscutting concerns in the source code.

8. 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 (METRICS03)*, 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] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

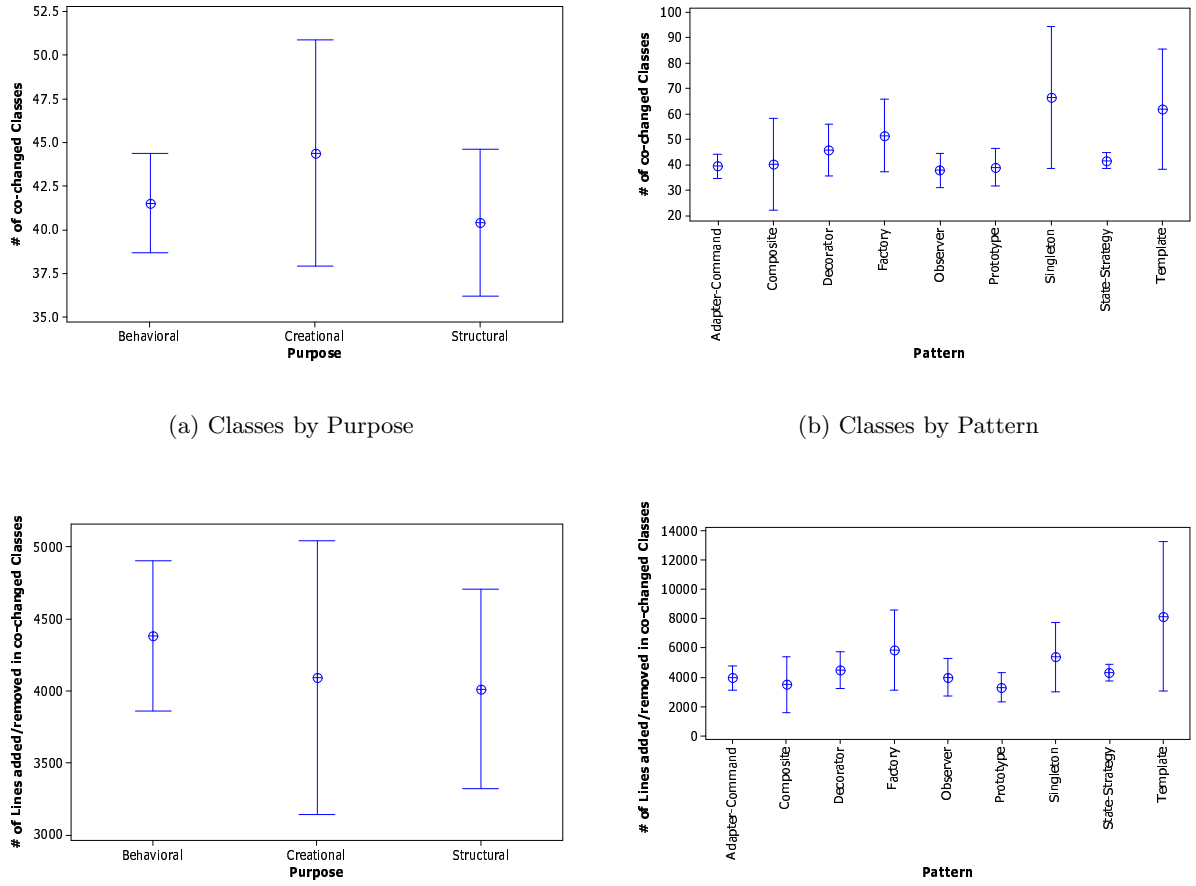
[23] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.

[24] 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.

[25] 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.

APPENDIX

A. ADDITIONAL ANALYSES



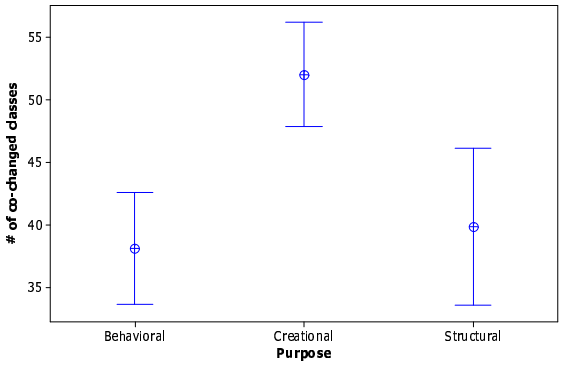
(a) Classes by Purpose

(b) Classes by Pattern

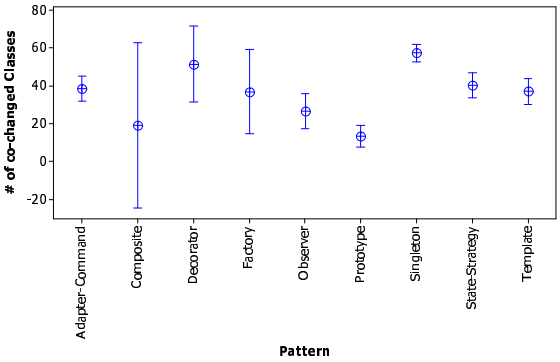
(c) Lines by Purpose

(d) Lines by Pattern

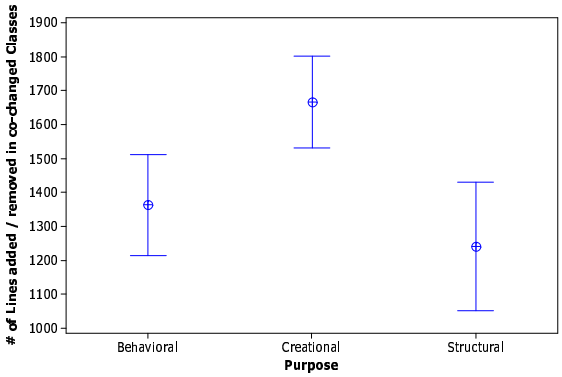
Figure 4: JHotDraw: co-changed classes and lines by purpose and pattern



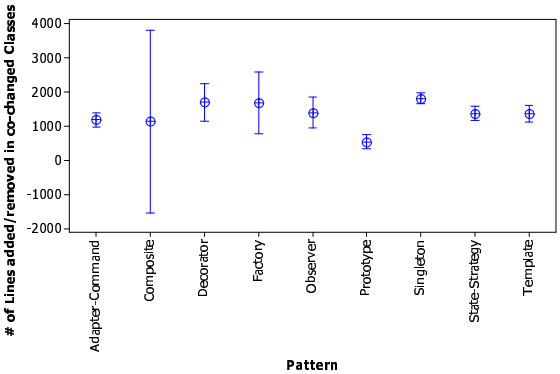
(a) Classes by Purpose



(b) Classes by Pattern

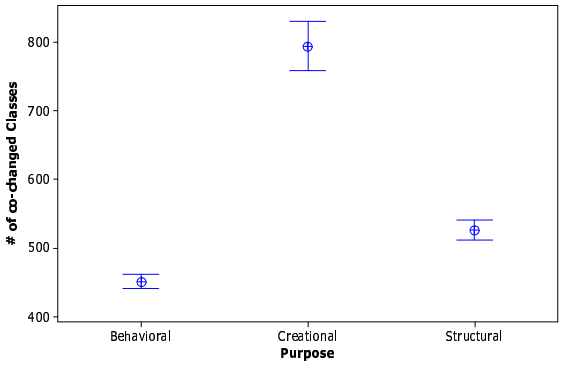


(c) Lines by Purpose

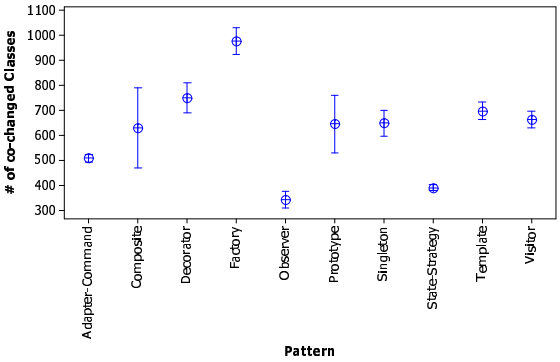


(d) Lines by Pattern

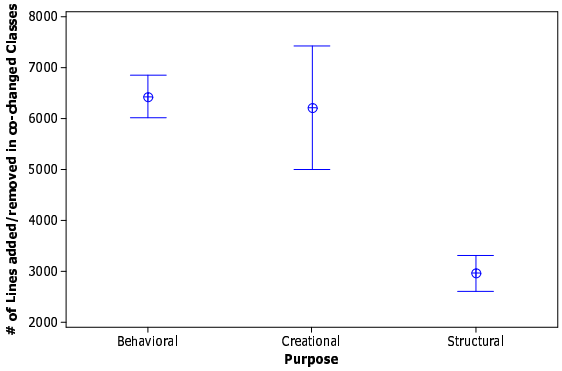
Figure 5: ArgoUML: co-changed classes and lines by purpose and pattern



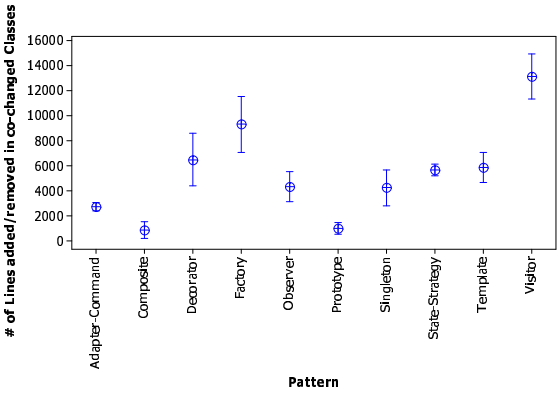
(a) Classes by Purpose



(b) Classes by Pattern

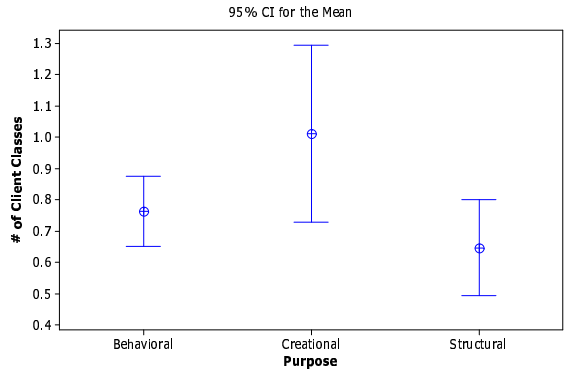


(c) Lines by Purpose

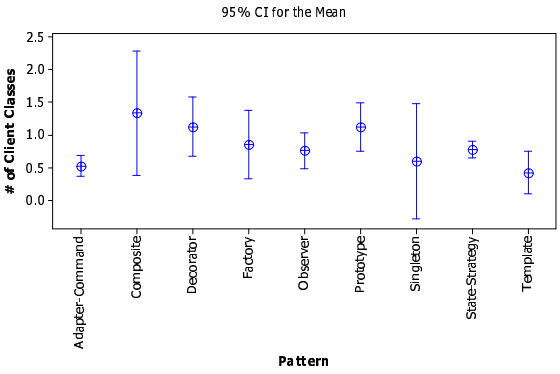


(d) Lines by Pattern

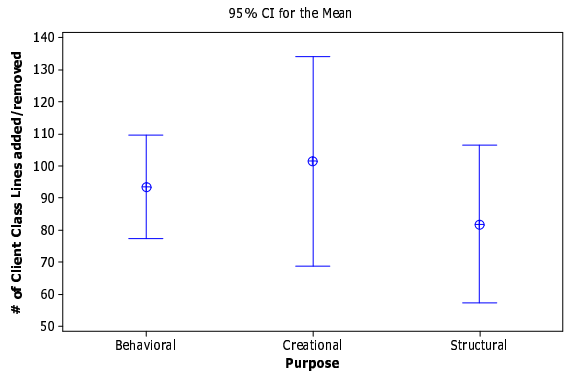
Figure 6: Eclipse-JDT: co-changed classes and lines by purpose and pattern



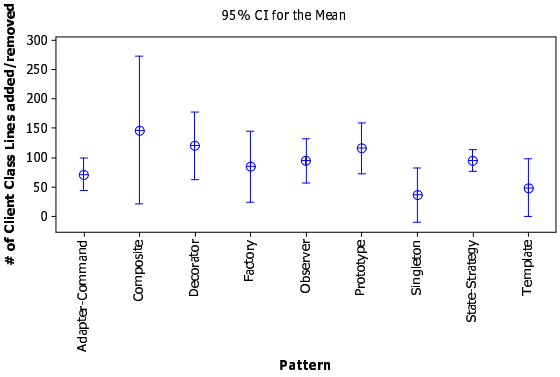
(a) Classes by Purpose



(b) Classes by Pattern

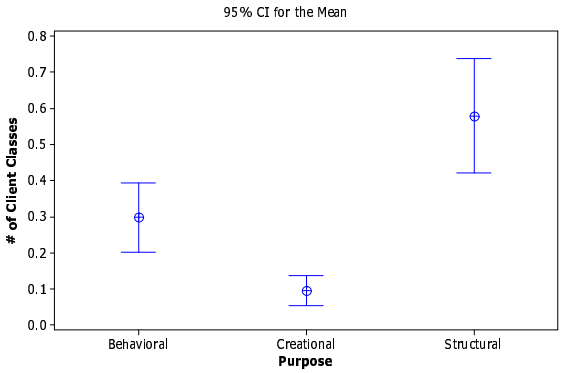


(c) Lines by Purpose

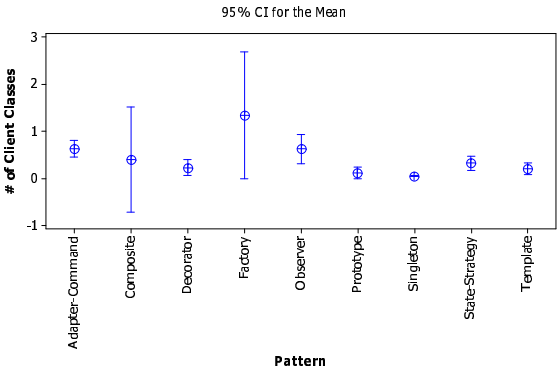


(d) Lines by Pattern

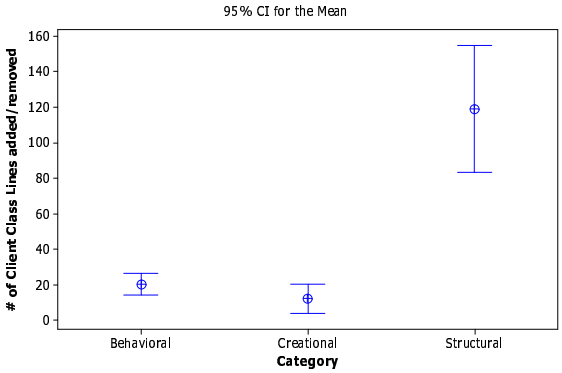
Figure 7: JHotDraw: changed client classes and lines by purpose and pattern



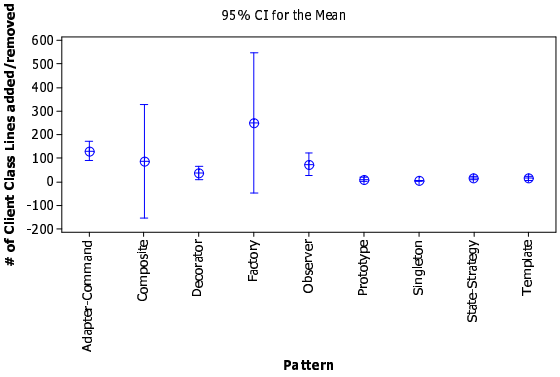
(a) Classes by Purpose



(b) Classes by Pattern

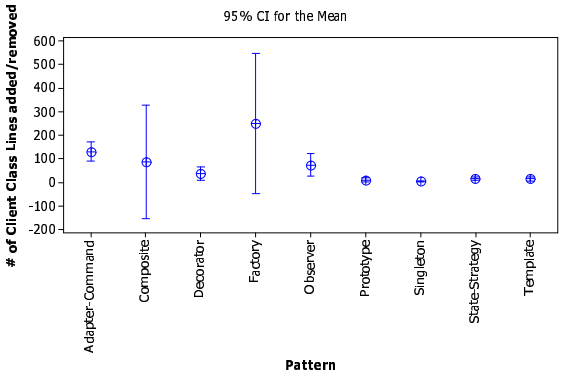


(c) Lines by Purpose

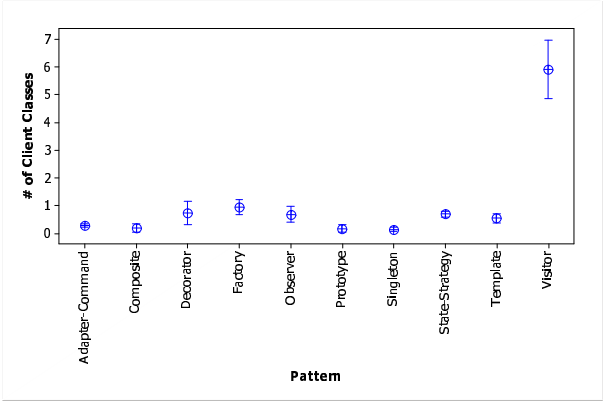


(d) Lines by Pattern

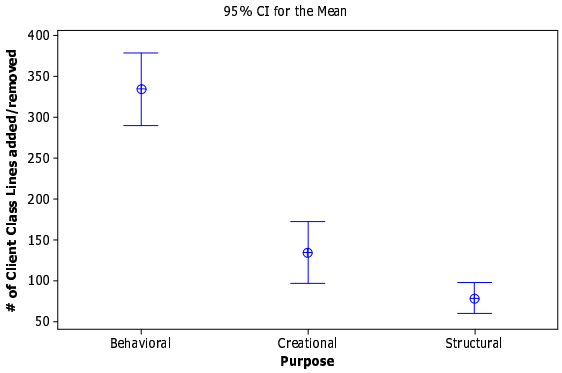
Figure 8: ArgoUML: changed client classes and lines by purpose and pattern



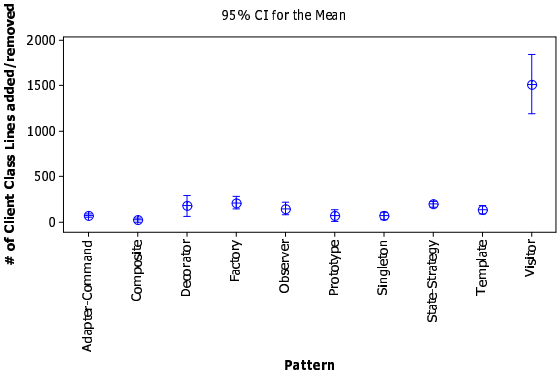
(a) Classes by Purpose



(b) Classes by Pattern

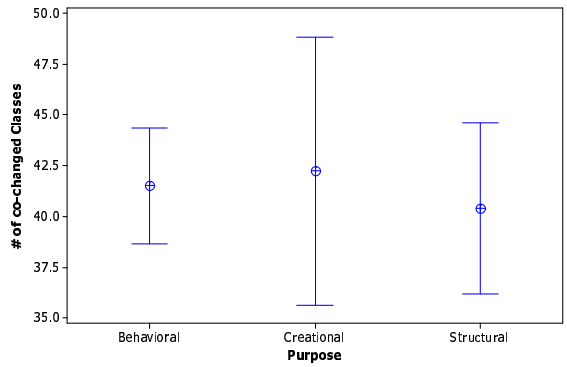


(c) Lines by Purpose

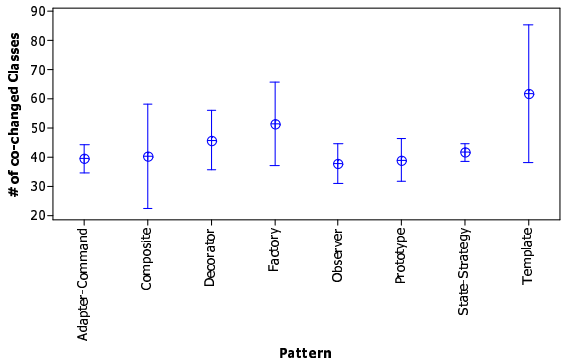


(d) Lines by Pattern

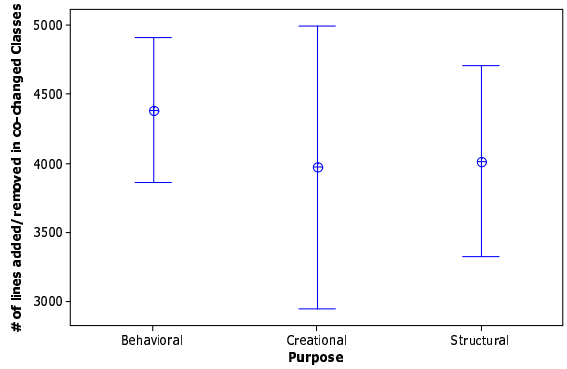
Figure 9: Eclipse-JDT: changed client classes and lines by purpose and pattern



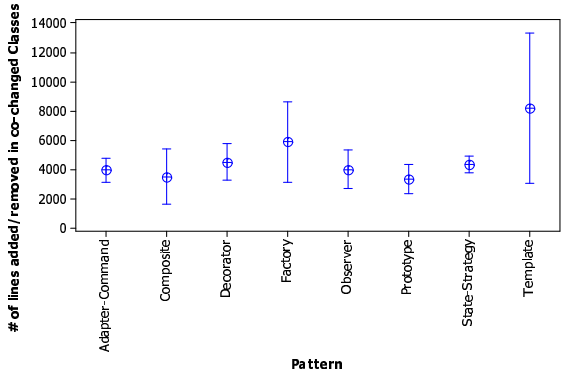
(a) Classes by Purpose



(b) Classes by Pattern

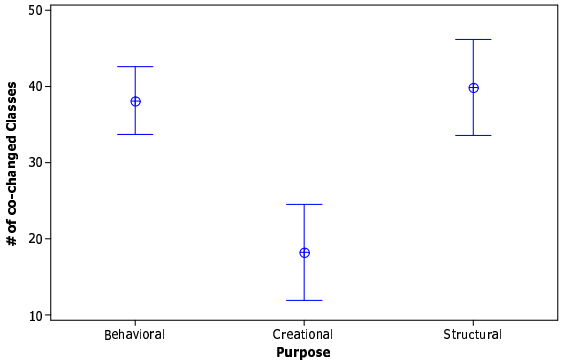


(c) Lines by Purpose

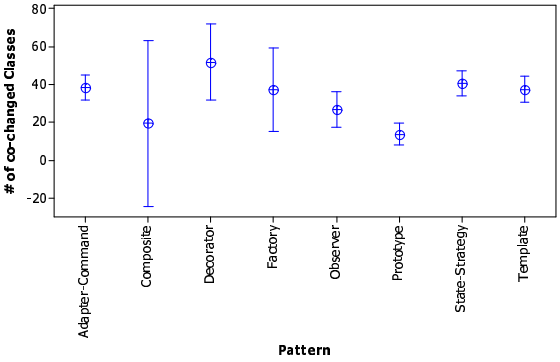


(d) Lines by Pattern

Figure 10: JHotDraw: co-changed classes and lines by purpose and pattern (without considering the Singleton)

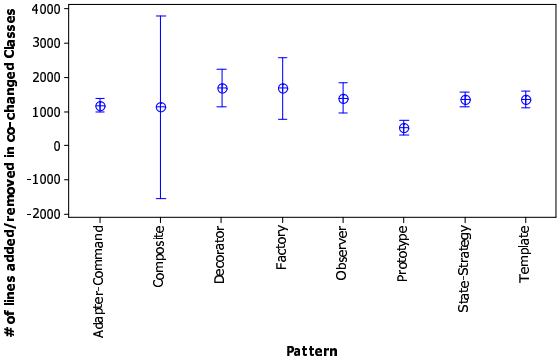


(a) Classes by Purpose



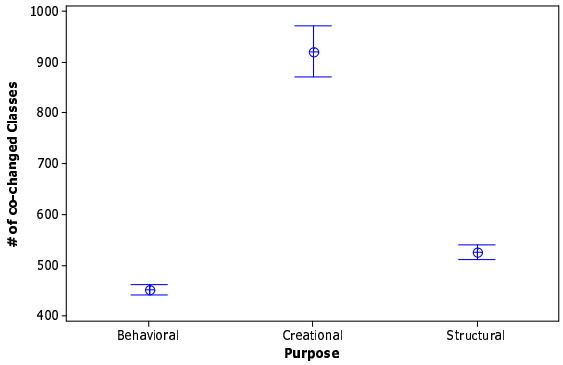
(b) Classes by Pattern

(c) Lines by Purpose

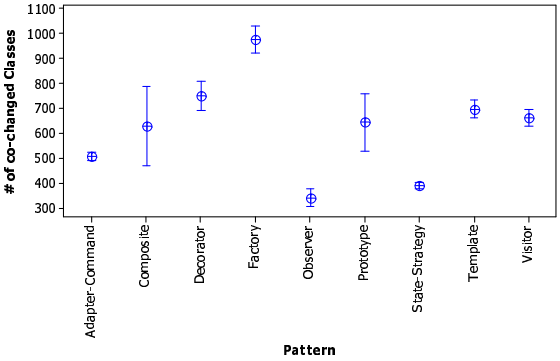


(d) Lines by Pattern

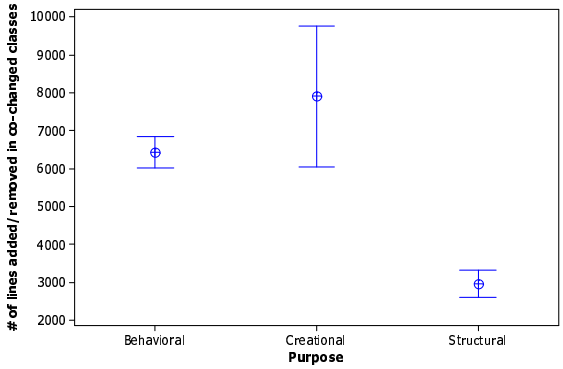
Figure 11: ArgoUML: co-changed classes and lines by purpose and pattern (without considering the Singleton)



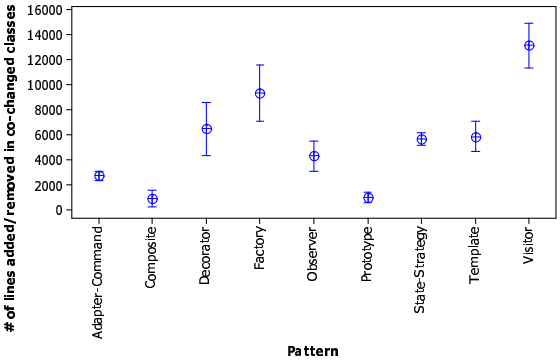
(a) Classes by Purpose



(b) Classes by Pattern



(c) Lines by Purpose



(d) Lines by Pattern

Figure 12: Eclipse-JDT: co-changed classes and lines by purpose and pattern (without considering the Singleton)