Relating the Evolution of Design Patterns and Crosscutting Concerns

Lerina Aversano, Luigi Cerulo, Massimiliano Di Penta
aversano@unisannio.it, lcerulo@unisannio.it, dipenta@unisannio.it
RCOST — Research Centre on Software Technology,
University of Sannio, Via Traiano 82100 Benevento, Italy

Abstract

Crosscutting concerns consist in software system features having the implementation spread across modules as tangled and scattered code. In many cases, these crosscutting concerns represent design pattern clients, i.e., invocations to pattern features. When a design pattern evolves, this can cause the addition or the change of scattered and tangled code, which contributes to the evolution of the crosscutting concern.

This paper empirically analyzes the relationship between design pattern evolution and the changes in the induced crosscutting concerns. Specifically, the paper investigates to what extent the crosscutting concern co-changes with the pattern, whether there is a relationship between the type of change and the induced crosscutting change, and whether different patterns induce different amount of crosscutting. The paper reports results from the analysis of Tomcat and JHotDraw evolution.

Keywords: Software Evolution, Mining Software Repositories, Design Patterns, Crosscutting Concerns, Empirical Study.

1 Introduction

Crosscutting concerns are software system features implemented as tangled and scattered code. Tangled code is the interference across multiple concerns in order to implement a new concern, while scattered code is the code of one concern spread across the system classes.

A common, pervasive example of crosscutting concern is represented by scattered code into design pattern [12] clients. Accessing wrapped code through an *Adapter*, accepting a *Visitor* into a data structure, notifying a model change to *Observers*, or invoking a software system piece of functionality through a *Command*, are just some examples of scattered code related to design pattern usage [22, 17].

One important benefit in the use of design patterns is the resilience to changes, avoiding that new requirements, and in general any kind of system evolution, causes major redesign. When a design pattern evolves, this can have consequences on the crosscutting concern of its clients. In some cases, the pattern interface changes (e.g., for a *Visitor* this can be due to changes in the accessed data structure). In other cases, the addition of new features to the pattern is made through sub-classing (for a *Visitor* this means that a new *Concrete Visitor* is available). Finally, the pattern interface and hierarchy may remain unchanged, while its implementation undergoes maintenance activities. These three kind of pattern changes may impact the crosscutting concern; in some cases the impact is expected due to the availability of new pieces of functionality offered by the pattern; in other cases — especially when the crosscutting changes due to pattern internal modifications — the change can be thought as a pattern modification side-effect and as a limited resilience of the pattern to changes.

While many benefits related to the use of design patterns have been stated, a little has been done to empirically investigate pattern change proneness [2] or whether there is a relationships between the presence of defects in the source code and the use of design patterns [27]. 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 have consequences on other classes and in particular on classes accessing 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 [11] — i.e., sets of artifacts changed together by the same author — from source code repositories triggers opportunities for this kind of studies.

This paper investigates the relationships existing between the evolution of design patterns and the evolution of crosscutting concerns they induce. The study stems from a previous work on the identification of crosscutting concerns through change sets mined from software repositories [8], and analyzes to what extent the crosscutting co-changes with the pattern, how the crosscutting evolves after pattern changes, and if there is any relationship between the type of change and the resulting impact on the crosscutting. Finally, the study investigates whether particular types of patterns induce more changes in the crosscutting concern. As case studies, we considered the crosscutting concerns identified by Marin *et al.* [21] on JHotDraw and Tomcat and, in particular, the subset of them related to design pattern clients. Results indicate a consistent change of crosscutting with the pattern. In particular, changes affecting the pattern interface almost always induce immediate co-change or delayed changes in the crosscutting, while pattern implementation changes do not always impact on the crosscutting and, where this happens, the crosscutting is co-changed with the pattern.

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, reports and discusses results. After a review of the literature in Section 4, Section 5 concludes the paper and outlines directions for future work.

2 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. This study exploits the advantage of analyzing a software system from different perspectives, as stated in [9], that are, in our case, both spatial (code dimension) and temporal (historic dimension).

2.1 Step 1: identification of crosscutting concerns and invoked pattern classes

The first step of our analysis process aims at selecting crosscutting concerns and patterns of which the crosscutting constitutes the client. We used the set of crosscutting concerns identified and discussed by Marin *et al.* [21] with a semi-automatic approach based on the fan-in metric, a measure of the number of methods that call some other method (a potential symptom of concern scattering across modules). Such an identified crosscutting concern (CC) is represented by a set of class methods calls (CC_c) which invoke a set of class methods (CC_m). For the purposes of this paper we consider crosscutting concerns containing invocation of methods belonging to design pattern classes.

2.2 Step 2: snapshots extraction and co-changes identification

After having identified a crosscutting concern, we analyze its changes by mining the version repository of the system under analysis. 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 [11]. Such a technique considers 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. Change sets can be extracted from a CVS/SVN history log using various approaches. We adopt a time-windowing approach that considers a change set as a sequence of file revisions that share the same author, branch, and commit notes, and such that the difference between the time-stamps of two subsequent commits is less or equal than 200 seconds [31].

2.3 Step 3: Location of pattern changes and determination of the kind of change

To determine in which snapshot a class c participating to a design pattern has been changed, we analyze the source code files changed in each snapshot, and perform a comparison between the class revision in two subsequent snapshots, S_{j-1} and S_j. By focusing more on structural changes, 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 $c \in S_{j-1}$ and $c \in S_j$ indicates that the class c — and thus the pattern — has been changed in correspondence of the snapshot S_j. The analysis is automatically performed by using a fact extractor based on the JavaCC parser generator[1] and a Perl script that compares facts (e.g., for each class the list of methods, attributes, associations, parents, descendants) of the two versions of c to identify the above mentioned differences.

2.4 Step 4: Locating crosscutting concern changes

The set of method calls of a crosscutting concern (CC_c) are well represented with the set of source code lines containing invocations to pattern methods. Unlike the previous, in this step we focus on line of code changes as more suitable to this kind of representation. In this way we are able to identify if a crosscutting method call identified in snapshot S_j, has been modified and/or added in a snapshot S_k, with $k < j$. We ignore its deletion as we are performing a backward analysis by starting from the snapshot (corresponding to a system release) were the crosscutting concerns have been identified.

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

Table 1. Case study history characteristics

SYSTEM	SOFTWARE TYPE	OBSERVED PERIOD	SNAPS	RELEASES	KNLOC min	KNLOC max	CLASSES min	CLASSES max
JHotDraw	Graphical editor	03/2001 ∼ 02/2004	177	5.2 – 5.4b2	13.5	36.3	164	489
Tomcat	Servlet container	04/2002 ∼ 03/2006	24,758	3.3 – 5.5	577.8	1,296.0	4,792	8,295

2.5 Step 5: Relating design pattern and crosscutting concern changes

The objective of this paper is to investigate on the relationship between pattern changes and changes in the induced crosscutting method call changes. To this aim, we use the results obtained in the previous steps to identify the snapshots in which the patterns and the crosscutting method calls changed together, and those in which they change separately. For each snapshot and for each crosscutting concern we determine: the kind and the size of pattern change, and the number of added crosscutting methods calls. Also, since the crosscutting concerns may or may not co-change with the pattern, we consider for them an *extended change*, including all the system snapshots from a pattern change to the next pattern change (or to the last snapshot if the pattern does not change anymore). In other words, this allows for analyzing the effect of a pattern change on the crosscutting even if this effect is not immediate, assuming that, until the pattern does not change again, the crosscutting change is due to the pattern last change.

3 Empirical Study

This section describes the empirical study context, defines the research questions, and discuss the obtained results.

3.1 Context description

We selected two open-source systems, JHotDraw and Tomcat which can be classified as small and large size systems, respectively. We extracted from such systems only the HEAD development trunk, excluding branches. Table 1 reports, for each system, the number of extracted snapshots, the range of analyzed releases, the number of non-commented klines of code (KNLOC), and the number of classes (excluding anonymous-classes).

JHotDraw[2] 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.

[2]*http://www.jhotdraw.org*

Table 2. Concerns in JHotDraw $v.5.4b1$

CONCERN/PATTERN	TARGET SIZE (nr. of methods)	FAN-IN (max)
Adapter	1	37
Command	2	24
Composite	12	24
Decorator	6	57
Observer	10	37
Persistence	6	22
Command (Undo)	3	25

Table 3. Concerns in Tomcat $v.5.5.17$

CONCERN/PATTERN	TARGET SIZE (nr. of methods)	FAN-IN (max)
Chain of responsibility (pipeline)	24	18
Composite	9	37
Observer (life-cycle)	5	42
Observer (container)	6	56
Redirector (request)	1	25
Redirector (response)	1	17
Visitor	1	28

Apache Tomcat[3] is a servlet container developed within the open-source Apache Jakarta project. The main elements of Tomcat are: the servlet container, *Catalina*, the JSP engine, *Jasper*, and the Tomcat connectors. We extracted a total of 24758 snapshots from release 3.3 to release 5.5.17, in the time interval between April, 2002 and March, 2006. In that interval the size of the system grew almost linearly from 577.8 KNLOC at release 3.3 to 1296.0 KNLOC at release 5.5.

Tables 2 and 3 show the crosscutting concerns having as target a design pattern identified, respectively, in releases $5.4b1$ ($152th$ snapshot) and 5.5.17 ($22960th$ snapshot) of JHotDraw and Tomcat. Such crosscutting concerns are those identified and widely described by Marin *et al.* in [21], which reports the same characteristics in terms of number of called methods, i.e. target size, and the maximum number of fan-in calls.

[3]*http://tomcat.apache.org*

3.2 Research Questions

This empirical study aims at answering the following research questions:

- *RQ1: To what extent design patterns co-change with their induced crosscutting concerns?* This research question analyzes whether the crosscutting concerns tend to change in correspondence of pattern changes (i.e., in the same change set) or after the pattern change (i.e., in the extended change set).
- *RQ2: Which is the relationship between different design pattern change types and their induced crosscutting concern changes?* This research question analyzes whether some design pattern change types (method addition or removal, attribute addition or removal, subclass addition or removal, or method implementation change) cause more changes in the induced crosscutting concern than others.
- *RQ3: Which is the relationship between different design pattern types and their induced crosscutting concern changes?* This research question analyzes whether different design patterns cause more change in the induced crosscutting concern than others.

This study is intended to be an explorative case study [28], aimed at answering research questions by pattern matching — or statistical tests where necessary — on data extracted by means of the analysis explained in Section 2, rather than attempting to reject null hypotheses.

3.3 Results

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

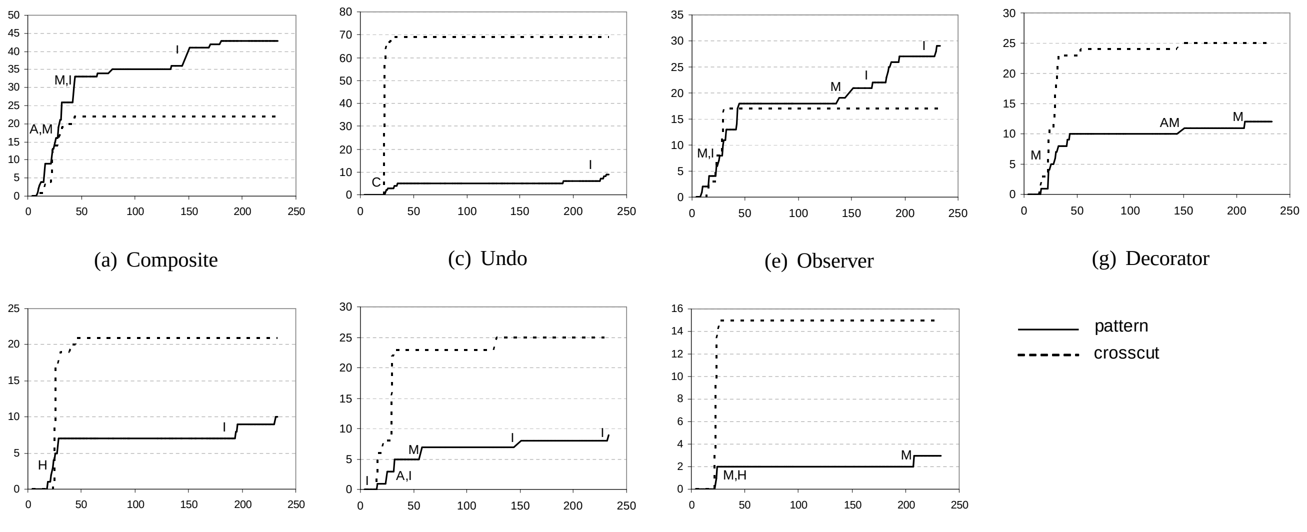
Figures 1 and 2 show, for JHotDraw and Tomcat respectively, the cumulative changes performed on design pattern classes and the cumulative changes (additions and modification) of their induced crosscutting method calls. The x-axis of each sub-figure represents the evolution time, i.e., the sequence of snapshots in which the changes occurred. The y-axis indicates the cumulative number of changes made in the pattern (solid line) or in the induced crosscutting (dashed line). Each kind of change in the pattern — i.e., the pattern class is added or removed, a subclass is added or removed, an attribute or a method is added or removed, or a method undergoes a change in its implementation — counts one, as well as the addition or change of a method invocation in the crosscutting concern. Of course, this does not provide a clear figure of the amount of change happened, while it provides an indication of whether the pattern interface or implementation changed, and whether a method was added or changed in the crosscutting. This, as shown in Figures 1 and 2, provides a visual representation of the temporal sequence of changes happened in the pattern and in the induced crosscutting. In correspondence of slopes, the pattern curve is labeled with the type(s) of change the pattern underwent:

- **C**, *class addition/removal*. It occurs when a new pattern class is added or removed.
- **H**, *hierarchy change*. It occurs when a subclass of the pattern class inducing the crosscutting is added and/or removed.
- **M** *method change*. It occurs when a pattern class method is modified, removed, and/or added.
- **A**, *attribute change*. It occurs when a pattern class attribute is modified, removed, and/or added.
- **I**, *method implementation change*. It occurs when the implementation of a pattern class method changes.

According to the information hiding principle, it is expected that changes in the pattern implementation should induce less changes in the crosscutting, while changes in the pattern interface should induce more changes in the crosscutting. We therefore consider two cases:

1. if design pattern classes undergo to changes of type (M)ethod, (C)lass, and/or (H)ierarchical, then its induced crosscutting concern will be affected at the same time (co-change), or with a short delay, and in any case before the next pattern change (extended change).
2. if design pattern classes undergo to changes of type (A)ttribute and/or (I)mplementation, then nothing should happen to its induced crosscutting concern.

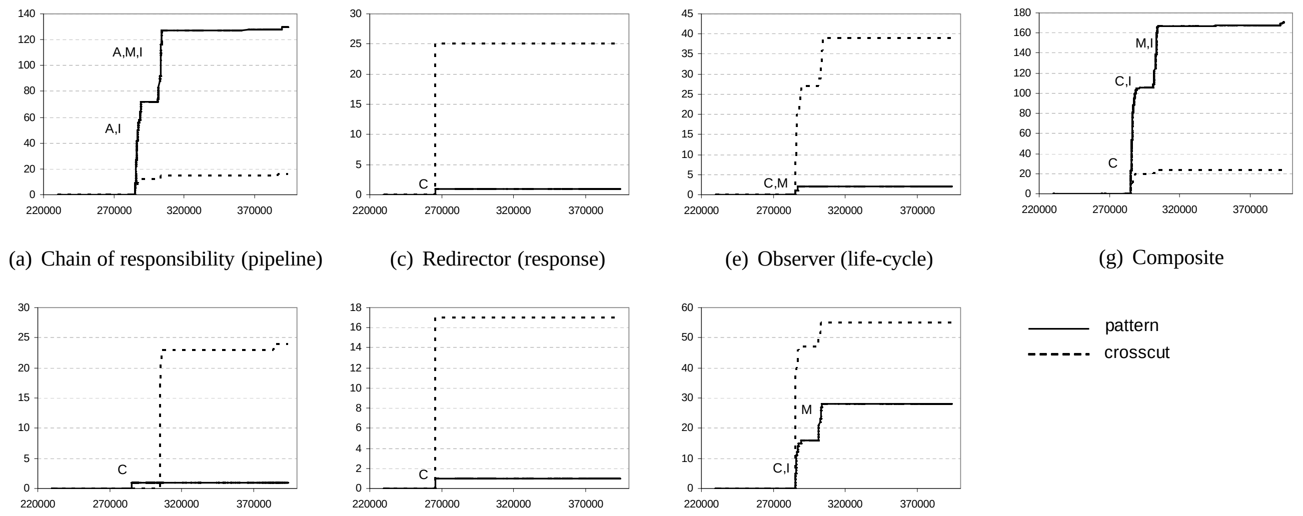
By analyzing the plots, it can be noted that the above two conditions were almost always met. In JHotDraw (see Figure 1) consistent changes can be explicitly observed for Composite, Undo, Command, and Persistence. However, as shown in Figures 1(e), 1(g), and 1(f), it can be noted that Observer, Decorator, and Adapter undergo to a set of (M)ethod type changes, which do not affect their induced crosscutting code. Such missed changes in crosscutting concern code can be explained by looking at source code and CVS messages. In snapshot 208, the *java.awt.Cursor* class has been systematically replaced with *CH.ifa.draw.contrib.framework.Cursor*, causing changes in Adapter and Decorator. Such a re-factoring did not affect their crosscutting code, because of homonymy of the Cursor class name. In snapshot 138, a merge conflict has been



(a) Composite (c) Undo (e) Observer (g) Decorator

(b) Command (d) Persistence (f) Adapter

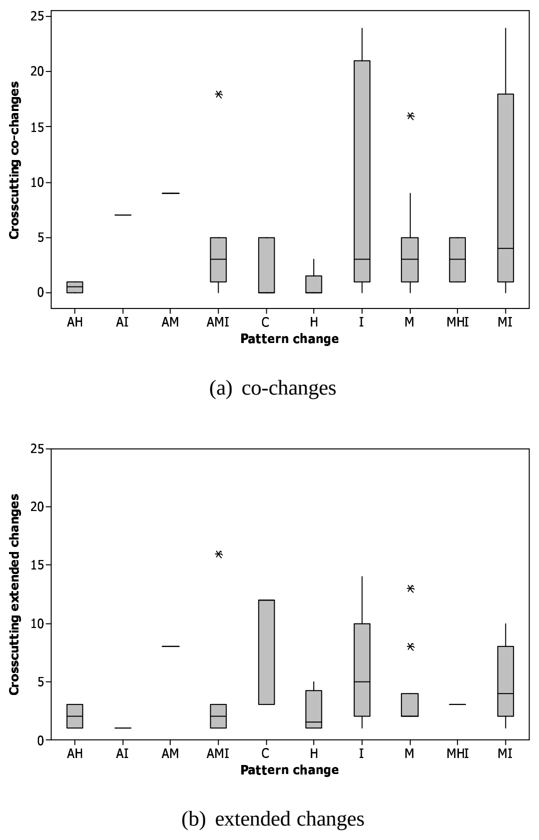
Figure 1. JHotDraw – cumulative changes of patterns and their induced crosscutting concerns



(a) Chain of responsibility (pipeline) (c) Redirector (response) (e) Observer (life-cycle) (g) Composite

(b) Visitor (d) Redirector (request) (f) Observer (container)

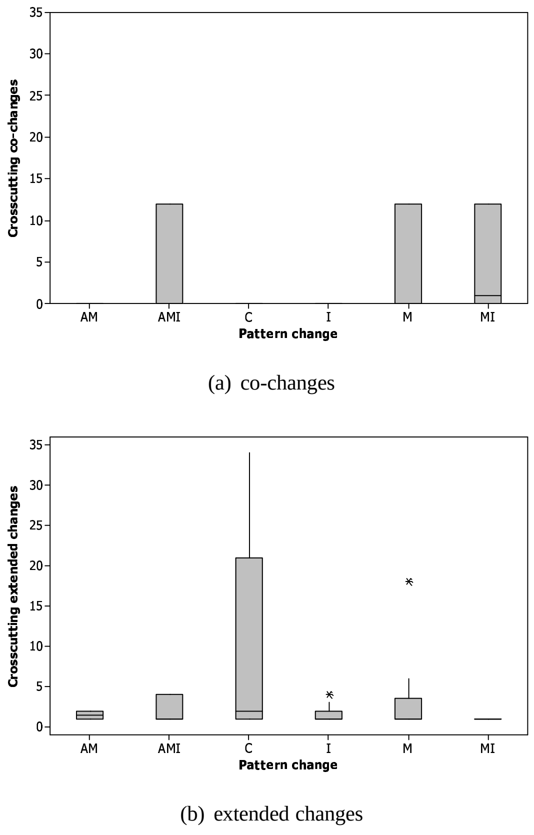
Figure 2. Tomcat – cumulative changes of patterns and their induced crosscutting concerns



(a) co-changes

(b) extended changes

Figure 3. JHotDraw – crosscutting (a) co-changes and (b) extended changes for different pattern change types



(a) co-changes

(b) extended changes

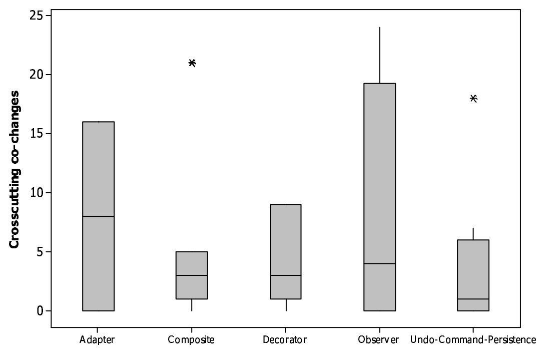
Figure 4. Tomcat – crosscutting (a) co-changes and (b) extended changes for different pattern change types

resolved with a number of methods added and removed in Observer classes, however without affecting its behavior. As a result, the crosscutting code was not affected. For Tomcat, as shown in Figure 2, in many cases a change in the pattern immediately induces a co-change in the crosscutting: this is the case of Redirectors, Observer (container) and Composite. In other cases, i.e., Chain of responsibility, Observer (life-cycle), and Visitor, the part of the crosscutting undergo a delayed change. This is especially the case of Visitor, started to be used about 20,000 snapshots after its insertion.

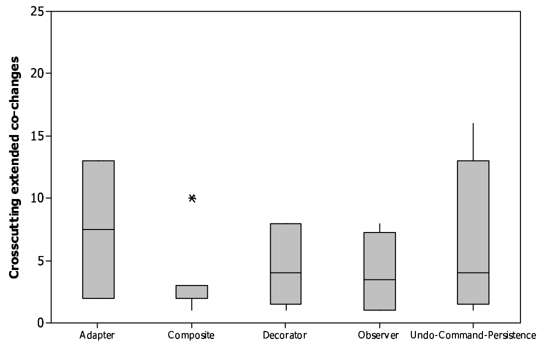
Figures also show that, for almost all patterns, after an initial burst of activity and change, the pattern and its call-dependent code tend to remain stable. This indicates that, overall, patterns work well as a mechanism to mitigate some kinds of evolution and their impact.

While Figures 1 and 2 provide a temporal view of the pattern and crosscutting changes, to better answer *RQ1* and RQ2 it would be useful to analyze, for different pattern changes (or combinations of changes) overall figures of the amount of crosscutting (i) co-changed with the pattern or (ii) changed within the pattern *extended* change. Results are shown as boxplots for JHotDraw in Figure 3. The x-axis indicates the different kind of changes, while the y-axis represent boxplots of the cumulative number of changes of that type for all the patterns in the system. As the boxplots show, not only different types of pattern changes induce different amount of crosscutting co-changes. Also, it is possible to note differences between the crosscutting addition/modification occurred in the pattern co-change (Figure 3(a)) and extended change (Figure 3(b)). The most evident difference, though not statistically significant, is visible for pattern class additions, i.e., (C)lass change type: the amount of induced co-change is higher in the extended change than in the co-change. The same applies for change in the pattern class (H)ierarchy; in this case the difference is also statistically significant[4] according to the Mann-

[4]In the context of this paper we used a significance level of 95%.

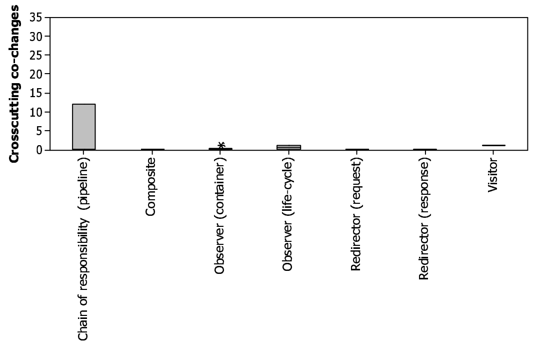


(a) co-changes

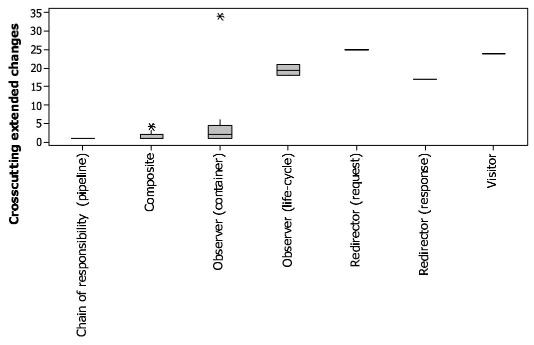


(b) extended changes

Figure 5. JHotDraw – crosscutting (a) co-changes and (b) extended changes for different patterns



(a) co-changes



(b) extended changes

Figure 6. Tomcat – crosscutting (a) co-changes and (b) extended changes for different patterns

Whitney test (p-value=0.046). Differences in the opposite direction can be noted for (I)implementation changes, inducing more immediate crosscutting co-changes (as shown in Figure 1, this kind of change is visible in particular for Composite, Observer). This can be explained by the fact that changes in the pattern interface (i.e., addition of classes of sub-classing) are mainly related to the addition of new features, accessed by other subsystems at a later stage (extended change) with the addition of method invocations in the crosscutting. Vice versa, implementation changes should not affect the crosscutting; when this happens, it can be due to corrective maintenance that needs to be quickly propagated. We did not find evidence of such a conjecture in the JHotDraw bug tracking system, probably because, being JHotDraw a relatively small open source project, the corrective interventions were propagated in the crosscutting by the same person who maintained the pattern without posting the problem on the bug tracking system.

Figure 4 shows boxplots of crosscutting co-changes and extended changes for Tomcat, again grouped by type of pattern change. As for JHotDraw, also in Tomcat different types of pattern changes induce different co-changes and extended changes in the crosscutting. The most evident and significant (p-value=$< 2.2 \cdot 10^{-16}$) difference occurs when a new pattern (C)lass is added: also in this case the number of changes is higher when considering the extended change. Moreover, in Tomcat it is possible to note that the amount of changes involving methods (AMI[5], M, MI[6]) is higher in the co-change than in the extended change, even though such a difference is not statistically significant. Differently from JHotDraw, for Tomcat implementation changes (I) tend to induce more extended changes than co-changes in the crosscutting. This because Tomcat is a larger system, involving more developers, and thus the maintenance of classes impacted by pattern change is delayed and, above all, it might be performed by a developer different from who maintained

[5]AMI=changes in Attributes, Method signatures, and Implementations
[6]MI=changes in Method signatures and Implementations

the pattern.

It is now possible to provide an answer to both **RQ1** and **RQ2**. We found, for both case studies, a consistent co-change between patterns and the induced crosscutting: except for some cases in JHotDraw (discussed above), changes in the pattern implementation (A or I) tend not to induce changes in the crosscutting, while changes in the pattern interface (C, H, M) induce immediate co-changes or delayed (extended) changes. Whenever implementation changes impact on the crosscutting, this happens immediately (at least where this is possible, like in smaller systems like JHotDraw) since it can be due to corrective maintenance that can impact pattern clients as well.

To answer **RQ3**, we analyzed the crosscutting induced co-change and extended change for different types of pattern. Boxplots for JHotDraw are shown in Figure 5. As the figure shows, and as it was confirmed by a multiple-means, non-parametric comparison test (Kruskal-Wallis) there is no significant difference in terms of induced crosscutting co-change or extended changes among different pattern. Only the Adapter induces a higher (although not significantly higher) number of co-changes than other patterns, due to its role: when the *Adaptee* (the class that provide the adapted interface), interface changes, clients need to be maintained. Also, no difference was found if comparing, for each pattern, the co-change with the extended change. Figure 6 shows boxplots for Tomcat. It can be noted how changes related to Chain of responsibility induce immediate crosscutting co-changes rather than delayed (extended) changes. This because such a pattern decouples requests between sender and receivers, and its changes need therefore to be immediately propagated. Observer (life-cycle), Redirectors, and Visitor induced more extended (i.e., delayed) that (immediate) co-changes. This is understandable for Observer and Visitor, since clients do not need to use these patterns as they change or as they are introduced in the system: a *Subject* — i.e., an object whose changes are observed by the observer — can attach an Observer whenever there is the need to notify it some changes in the *Subject* state. For the same reason, a data structure (*Element*) accepts a Visitor if the operation performed by such a Visitor needs to be performed on that data structure. Although Redirectors are similar to Chain of responsibility, they induce more extended changes than co-change. However, this is not significant since the data set only contains one Redirector (request) and one Redirector (response).

In summary, it is possible to answer **RQ3** stating that, *although in general it is not possible to identify design patterns inducing more crosscutting changes than others*, there are patterns, essentially having a decoupling responsibility — such as Chain of responsibility or Adapter — that induce immediate co-changes, while others — like Visitor or Observer — induce changes only when the client need to access the introduced/maintained feature.

3.3.1 Threats to Validity

This section discusses threats to validity that can affect the results reported in Section 3.3, following a well-known template for case studies [28]. Regarding *construct validity*, threats can be due to the measurement performed, in particular related to (i) how crosscutting and patterns were identified; (ii) how change sets were identified; and (iii) how crosscutting and pattern changes were analyzed across releases. For crosscutting and pattern identification we relied on an available oracle [21] that was already assessed. We analyzed change sets as a way to assess the impact of pattern change on the crosscutting. Of course, co-change could also happen accidentally, and more sophisticated impact analysis techniques could have been used [1], although change-sets and bug issues are used for this purpose [7, 31]. The pattern changes analysis relied on information extracted by our analyzer based on the JavaCC Java grammar, and on CVS/SVN diffs. As discussed in Section 2.4, we limited the analysis of crosscutting concerns to modifications and additions, having performed a backward analysis. Relationships between pattern changes and crosscutting removal remains to be investigated.

Threats to *internal validity* did not affect this particular kind of study, being it an explorative study [28].

Threats to *external validity* are related to what extent we can generalize our findings. We considered two different software systems, differing for their domain (graphical editor vs. servlet container) and size (small vs. medium-large), 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, the analysis here was limited to a limited set of patterns accessed by means of crosscutting belonging to the Marin *et al.* [21] dataset.

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

4 Related Work

In our knowledge, there are no papers aimed at empirically analyzing the relationship between the evolution of design pattern and their induced crosscutting concerns. A number of papers investigate on the relationship between design patterns and their homologue implemented as aspect modules, such as Hannemann and Kiczales [17], and Garcia *et al.* [13]. The first shows that in 17 of 23 cases there is an improvement in code quality when a design pattern is implemented as an aspect, the latter measures such quality in both cases showing that aspect oriented code exhibits higher code quality. As a future work, we aim to extract the evolution of such aspect oriented re-factoring changes to show if the code quality improves in time by looking, for example, at defect density. Regarding the analysis of design pattern evolution, Bieman *et al.* [2] 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; Vokáč [27] 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; Prechelt *et al.* [24] performed a series of controlled experiments with the aim of comparing design patterns with alternative, simpler solutions to perform maintenance tasks.

Historical co-change analysis has provided new opportunities for a number of issues: predict change propagation [29, 31], observe clone [14, 18] and crosscutting concern [6] evolution, identify crosscutting concerns [4, 8], detect of logical coupling between modules [11], find common error patterns [20], or identify fix inducing changes [19].

Although we relied on crosscutting concerns mined using the Marin *et al.* approach [21] based on the fan-in metric, this can be done using alternative approaches. *Aspect Browser* [15] uses text-based pattern matching to identify aspects. A developer specifies a regular expression that describes the code belonging to the aspect and the tool identifies the code conforming to the regular expression. Prior knowledge of the system strongly affects the usefulness of the achieved results [23]. Various extensions of this tool has been developed by introducing different mining heuristics, such as type ranking and control flow information [16, 30]. Ettinger *et al.* [10] propose a program slicing technique to identify entangled code. The slice is computed from an expression or a statement pointed out by a developer. Bruntink *et al.* [5] propose the use of clone detection for the identification of crosscutting concerns, comparing the performance of different clone detection techniques, namely AST-based and token-based. Aspect mining using dynamic analysis has been proposed by Breu and Krinke [3]: the idea is to detect particular patterns occurring in an execution trace. An approach for aspect mining using formal concept analysis on execution traces was proposed by Tonella *et al.* [25]. Also aspects were mined by detecting patterns in execution traces using formal concept analysis by Tourwé *et al.* [26].

5 Conclusions and Work-in-Progress

This paper reported an empirical study investigating on the relationship between design pattern evolution and changes in the induced crosscutting concerns. Results indicate that different type of changes in design patterns classes cause a different behavior on the induced crosscutting concerns. It was found that, in most cases, a change performed on a design pattern class — involving the pattern interface or class hierarchy — causes a consistent change of the induced crosscut. If changes are of type (A)ttribute and/or (I)mplementation — i.e., are hidden to pattern clients — then nothing happens to the induced crosscutting concern. When analyzing the distribution of co-changes and delayed (extended) changes, it was found that, whenever possible, implementation changes, if impacting crosscutting concerns, are propagated immediately, i.e., within the pattern co-change, while other changes — such as hierarchy changes and addition of pattern classes — tend to induce late changes in the crosscutting. When analyzing the effect induced by different pattern types, overall no significant difference was found, although pattern having decoupling responsibilities tend to induce their impact on the crosscutting within the co-change rather than delaying it.

The analysis process proposed in this paper poses the basis for further studies aimed at increasing the external validity of results and leading to more general conclusions. Future work also aims at improving the accuracy of results by considering more accurate impact analysis or dependency analysis techniques. In this paper we mainly focused on what happened in the past with respect to a snapshot where a crosscutting concern is identified. We aim to include, by performing a *forward analysis*, what happens in the future, e.g., crosscutting concern deletions.

6 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'Universit e della Ricerca) under grant PRIN2006-2006098097. Authors would like to thank the anonymous reviewers for their useful feedbacks.

References

[1] 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.

[2] 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.

[3] S. Breu and J. Krinke. Aspect mining using event traces. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 310–315, 2004.

[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] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Software Eng.*, 31(10):804–818, 2005.

[6] 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.

[7] 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.

[8] 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.

[9] L. Cerulo. *On the Use of Process Trails to Understand Software Development*. PhD thesis, University of Sannio, 2006.

[10] R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *AOSD*, pages 93–101, 2004.

[11] 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.

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

[13] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM Press.

[14] 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.

[15] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada*, pages 265–274, 2001.

[16] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. In P. Tarr and H. Ossher, editors, *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*, May 2001.

[17] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.

[18] 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.

[19] 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.

[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] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Methodol.*, 2007.

[22] M. P. Monteiro and J. M. Fernandes. Refactoring a Java code base to AspectJ: An illustrative example. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 17–26. IEEE Computer Society, 2005.

[23] G. C. Murphy, W. G. Griswold, M. P. Robillard, J. Hannemann, and W. Leong. Design recommendations for concern elaboration tools. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 507–530. Addison-Wesley, Boston, 2005.

[24] 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.

[25] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the Working Conference on Reverse Engineering*, pages 112–121, 2004.

[26] T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In *Workshop on Source Code Analysis and Manipulation*, pages 97–106, 2004.

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

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

[29] 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.

[30] C. Zhang and H.-A. Jacobsen. PRISM is research in aSpect mining. *OOPSLA Companion*, 39(10):20–21, Oct. 2004.

[31] 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.