The Relationship between Design Patterns Defects and Crosscutting Concern Scattering Degree: an Empirical Study

Lerina Aversano[1], Luigi Cerulo[2], Massimiliano Di Penta[1]

aversano@unisannio.it, lcerulo@unisannio.it, dipenta@unisannio.it

[1] Dept. of Engineering, University of Sannio, Via Traiano – Benevento, Italy

[2] Dept. of Biological and Environmental Studies, University of Sannio,

Via Port'Arsa, 11 – Benevento, Italy

Abstract

Design patterns are solutions to recurring design problems, aimed at increasing reuse, code quality and, above all, maintainability and resilience to changes. Despite such advantages, the usage of design patterns implies the presence of crosscutting code implementing the pattern usage and access from other system components. When the system evolves, the presence of crosscutting code can cause repeated changes, possibly introducing defects.

This paper reports an empirical study, in which it is showed that, for three open source projects, the number of defects in design-pattern classes is in several cases correlated with the scattering degree of

their induced crosscutting concerns, and also varies among different kinds of patterns.

Keywords: Software engineering, Software maintenance, Software metrics, Software quality.

1 Introduction

Object-oriented design patterns are recurring design solutions for object-oriented systems which can improve several quality attributes, such as reusability, maintainability, and comprehensibility [13]. However, the usage of object-oriented design patterns can cause the presence of crosscutting code spread across classes using the pattern [7] (hereby referred as "pattern clients").

Let us explain this issue with two examples. Observer design patterns *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"* [13]. In the Observer pattern, an operation that changes any Subject must trigger notifications to its Observers. Chain Of Responsibility patterns *"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request"* [13]. In the Chain Of Responsibility pattern, all Handlers need to be able to accept requests or events and to forward them to the successor in the chain. For both Observer and Chain of Responsibility patterns, there are features that crosscut different pattern structures, such as notification actions in Observer Subjects and the event handling mechanism in Chain Of Responsibility handlers.

The implicit crosscut structure of patterns has been recognised in literature. Hannemann and Kiczales [18] argued that, in 17 out of the 23 Gamma *et al.* design patterns, there is an improvement in code quality when a design pattern is implemented as an aspect. Garcia *et al.* [14] performed an empirical study showing that when design patterns are implemented with aspects—e.g., by using AspectJ—code quality metrics improve. The crosscutting structure of design patterns is quite complex to identify, al-

though aspect mining techniques can be applied to this aim. Several aspect mining techniques are able to identify such crosscutting structure in existing software systems (e.g., [23]). Accessing adapted code through an Adapter pattern, 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 a few examples of scattered code related to the client's usage of design patterns. 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. After a first exploratory study in the evolution of design patterns [1], we reported results on an empirical analysis of the relationship between design pattern evolution and the changes in the induced crosscutting concerns [2]. In particular, results showed that design pattern clients tend to co-change with the pattern itself, or at least in the time interval between two subsequent pattern changes.

Benefits and problems related to the use of design patterns, and to the presence of scattered code, have been largely investigated in literature, but separately. For instance, Vokáč [29] investigated the defect frequency in design patterns code, finding that, especially for some design patterns, the fault proneness of the source code is lower than non-pattern code. Guéhéneuc and Albin-Amiot [15] found that a proper usage of design patterns reduces the number of defects. On the other hand, other studies—in particular a recent study by Eaddy *et al.* [9]—indicated the presence of a correlation between the scattering degree of a concern and the likelihood its implementation has to contain defects. Thus, crosscutting concerns make the code more difficult to be maintained, because changes need to be made in several distinct locations.

The question that now arises is whether the correlation between crosscutting concerns scattering degree and code fault proneness can also be found for crosscutting concerns induced by design patterns. While design patterns represent recurring—and often known by developers—design solutions, the in-

duced crosscutting concerns could affect their fault proneness. On one hand, as indicated by Eaddy *et al.* [9], an increasing scattering degree of the crosscutting concern can in turn increase the number of defects, since developers have to change scattered code. On the other hand, when design pattern clients—i.e., classes invoking the design pattern—undergo changes, it is not guaranteed that the pattern code is left unchanged. As reported in a previous study [1], there are cases in which the presence of a design pattern does not necessarily make the system robust to changes, and this is particularly true for patterns playing a crucial role for the application. For example (i) in JHotDraw, Observers are used to realise the Model-View Controller architecture, and Composites to handle composite figures; (ii) in ArgoUML Commands are massively used to decouple the Graphical User Interface menus from the respective actions; and (iii) in Eclipse, Visitors are used to perform source code analysis and transformation. Thus, changes to crosscutting code related to many different design pattern clients can cause changes to the design pattern itself and increase its fault proneness.

This paper investigates whether the presence of defects in design patterns' code is correlated to the scattering degree of their induced crosscutting concerns. Design patterns have been reverse engineered by us from the source code using the tool proposed by Tsantalis *et al.* [28]. Results from the analysis of three open source systems, JHotDraw, ArgoUML, and Eclipse-JDT, indicate a significant correlation between the number of defects occurring in the design pattern and the spread of crosscutting concerns, although in some cases such a relationship depends on the nature of the design pattern.

Below we provide definitions, taken from the existing literature, of the main concepts that will be referred in the remainder of this paper:

- **Crosscutting concern:** from [11]: *"Software development addresses concerns, both concerns at the user/requirements levels and at the design/implementation level. Often, the implementation*

of one concern must be scattered throughout the rest of an implementation. We say that such a concern is crosscutting. Note that what is crosscutting is a function of both the particular decomposition of a system and the underlying support environment. A particular concern might crosscut in one view of an architecture while being localized in another; a particular environment might invisibly support a concern (for example, security) that needs to be explicitly addressed in another."

- **Aspect:** from [11]: *"An aspect is a modular unit designed to implement a concern. An aspect definition may contain some code (or advice, which follows) and the instructions on where, when, and how to invoke it. Depending on the aspect language, aspects can be constructed hierarchically, and the language may provide separate mechanisms for defining an aspect and specifying its interaction with an underlying system.*"

- **Defect:** or fault, is "*an incorrect step, process, or data definition in a computer program.*" [21]. In this context, we refer to defects that have been fixed by means of a change committed in a versioning system—Concurrent Versions Systems (CVS)[1] or SubVersioN (SVN)[2] for example—and which commit note refers to a bug report posted on a bug-tracking system [31], *Bugzilla* in our case.

- **Design pattern:** it is a recurring design solution for a software system. In particular, an object-oriented design pattern—and in the remainder of this paper we use the term "design pattern" to refer object-oriented design patterns only—is a recurring design solution for an object-oriented systems [13]. Such a design solution is identified in terms of relationships between classes, re-

[1] http://ximbiot.com/cvs/

[2] http://subversion.tigris.org/

sponsibilities, and collaborations.

The remainder of this paper is organised as follows. Section 2 reports a review of related work presented in literature. Section 3 describes the analysis process performed to extract data needed for the empirical study. Section 4 defines the empirical study, while Section 5 reports and discusses the results obtained. Finally, Section 6 concludes the paper and outlines directions for future work.

2 Related Work

This section discusses literature which aims at investigating the quality and the evolution of design pattern code, and the fault proneness of crosscutting concerns.

2.1 Design Pattern fault proneness

A number of studies investigate on the relationship between design patterns and their homologue implemented as aspect modules [14, 18]. We share with those authors the idea that design patterns exhibit better code quality when their induced crosscutting concern are properly managed, for example by using aspect oriented techniques. In a previous work, [2] we analysed the relationship between design pattern evolution and the changes in their induced crosscutting concerns. Whereas in the previous work we investigated to what extent crosscutting concerns co-change with the patterns, in this paper we focus on issues related to the defects which such crosscutting concerns may induce.

Empirical investigation on the evolution of design patterns is crucial to understanding the evolution of object-oriented systems; despite that, only a few quantitative results are available. Bieman *et al.* [5] analysed four small size systems and one large size system to identify the observable effects of the use of design patterns, such as pattern coupling with other code and change proneness. They found that, in

general, design patterns are loosely coupled, with some exceptions (e.g., Singleton). Vokáč [29] analysed the corrective maintenance of a large commercial product over three years, comparing the defect rate of classes participating to design patterns with the defect rate of other classes. He found that some patterns, such as Observer and Singleton, are correlated with large code structures and are likely to be more fault prone. Prechelt *et al.* [27] performed a series of controlled experiments with the aim of comparing design patterns with alternative, simpler solutions to perform maintenance tasks. They found that the code developed using design patterns contained a lower number of defects than the code developed using alternative design strategies.

In a recent paper [1] we performed a large empirical study aimed at investigating the pattern change frequency, the kinds of changes the patterns undergo and the capability of patterns of making a system robust to changes. The present work shares with [1] the idea that patterns particularly crucial for the application tend to change more frequently and to have a higher set of classes co-changing with them. This requires also to pay attention to the number of defects in the source code and the induced crosscutting concern.

2.2 Crosscutting Concern fault proneness

Recently, Eaddy *et al.* [9] presented an empirical study addressing the role of crosscutting on the number of defects in a program. They defined a concern model and conducted an analysis on three systems, finding a significant and moderate correlation between the scattering degree of a crosscutting concern and the fault proneness of the concern code. There are differences between the present work and the work of Eaddy *et al.* [9]:

- we focus on crosscutting concerns induced by design patterns and, instead of looking at defects

occurring in the crosscutting concern code, we look at defects occurring to the design pattern code;

- instead of considering the scattering degree for a given release, we considered the scattering degree average value across a series of snapshots extracted from the CVS repository. This is because for our study just considering its value for an arbitrary release is not meaningful;

- we perform a separate analysis for 10 different kinds of design patterns.

The relationship between the scattering degree and the code fault proneness is not surprising, since the scattering degree causes an increase of the Chidamber and Kemerer Coupling Between Objects (CBO) metric [6], and authors such as Gyim*ó*thy *et al.* [17] and Basili *et al.* [4] found a significant correlation between CBO and fault proneness. Other authors correlated other metrics different from the scattering degree to the source code fault proneness. For instance, Harrison *et al.* [19] performed a controlled experiment to analyse the effect of varying levels of inheritance on changeability, showing that larger systems are equally difficult to understand whether or not they contain inheritance. More recently, Nagappan and Ball [26] presented a technique to predict system defect density using a set of relative code churn measures, i.e., changes made to a component over a period of time.

3 Extraction process

This section describes the process to extract, from the CVS or SVN repository containing the source code of the system under analysis, the data required to perform the empirical study presented in this paper. In particular, the data extraction process requires to:

1. identify change sets from CVS/SVN repositories;

2. identify design patterns and design pattern clients from software releases; and

3. perform a fine-grained analysis of design pattern changes.

3.1 Change set identification

The first step aims at extracting from CVS/SVN repositories logical coupled changes performed by developers working on a bug fix or an enhancement feature [12]. To this aim, a number of techniques have been proposed. Such techniques consider the evolution of a software system as a sequence of *Snapshots* ($S_0, S_1, \ldots, S_n$) generated by a sequence of source code changes, also known as *Change Sets* (Δ_1, Δ_2, ..., Δ_n), representing the logical changes performed by a developer. The identification of a *Change Set* is based on a time-window approach that considers [32]: *"two subsequent changes δ_i and δ_{i+1} by the same author and with the same rationale are part of one transaction Δ if they are at most 200 seconds apart"*. In other words, a change set puts together CVS/SVN commits if (i) they have been done by the same author (i.e., same CVS/SVN userid); (ii) the commit note it the same; and (iii) there not exists, in the change set, any commit having a timestamp more than 200 seconds far away from the closest one. The idea is to put together commits related to what it is assumed to be a unique change task, for which the author however performs multiple commits. In this paper we identify change sets limiting our analysis to the main development trunk, which excludes branches and is labelled as "HEAD" in CVS repositories.

3.2 Design pattern and pattern client identification

The second step aims at identifying design patterns in each release of the system under analysis. For our purposes, we are interested in analysing changes which occurred in the design pattern classes, that for a pattern instance k belong to the set $DP_k = \{M_k, X_k\}$, where M_k contains the main class participant(s), usually two abstract classes, and X_k all classes that extends M_k (i.e., concrete classes). For

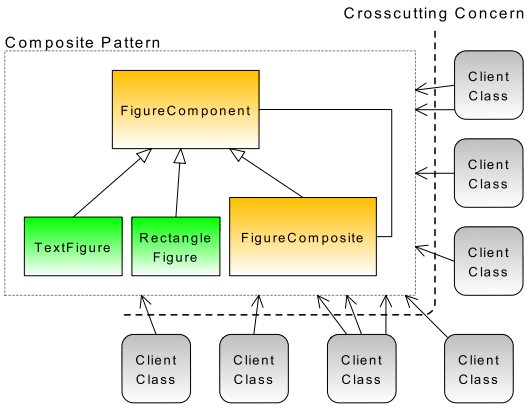


Figure 1. Main elements of a design pattern.

example—see Figure 1—if we consider the Composite design pattern from the Gamma *et al.* book [13], M_k contains the *FigureComposite* and *FigureComponent* abstract classes, X_k contains all the concrete subclasses, such as *RectangleFigure*, *TextFigure*, or *BoxFigure*.

We identify classes in M_k by using a graph-matching based approach proposed by Tsantalis *et al.* [28], which is based on similarity scoring between graph vertexes. The tool identifies the two main participants (i.e., super classes) of each pattern, and is able to detect the following kinds of patterns: Object Adapter-Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State-Strategy, Template Method, and Visitor. We chose this tool due to its availability, its ability to detect a reasonable range of different design patterns, and its ability to analyse the three software systems under study. There are several other design pattern detection approaches available in literature [8, 16, 20, 22], however no tool is currently available for these approaches.

As mentioned in the introduction, this work aims at relating bug-fixing changes in the set DP_k with the number of design pattern client classes C_k, i.e., the client classes which use instances of M_k and/or X_k. The set of C_k is identified in an over-conservative way. In particular we consider, for each class: (i) attribute types for the class and for the parent classes, i.e., associations, (ii) method parameter types

and local variable types, (iii) casting and constructor invocations. This quickly provides a superset of possible dependencies from/to the pattern, although more precise approaches can be used to refine the dependency set [24].

3.3 Fine-grained analysis of design pattern changes

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 in release j represents an evolution of a design pattern instance identified in the previous 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. Similarly to what we previously did to build the pattern history [1], we assume that a pattern instance in a release represents the evolution of a pattern instance in the previous release if and only if (i) the kind 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.

With such information, we are able to identify the set of snapshots where a design pattern instance changed, and to rebuild, for each snapshot, the overall pattern structure, i.e., clients and main participant descendants. We indicate with $\Gamma(DP_k)$ the set of snapshots S_i where at least one of the classes belonging to DP_k was changed. Moreover, as the design pattern structure may change across snapshots, we indicate with DP_k^i the set of classes belonging to the pattern instance DP_k at snapshot S_i and with C_k^i the set of DP_k client classes. The set of classes that have been modified between two subsequent snapshots S_{i-1} and S_i can be identified by comparing the revisions of classes belonging to both, $i-1$ and i, snapshots. A change occurred if such revisions differ, according to a context *diff*. In addition, source code analysis is performed to filter out changes related to re-styling, indentations and changes in the comments. The

analysis is performed by using a fact extractor based on the JavaCC parser generator[3] and a Perl script that compares facts extracted from class revisions to identify the above mentioned differences.

4 Empirical Study Definition

The *goal* of this empirical study is to perform a fine-grained analysis of the relationship between the spread of design pattern-related crosscutting concerns and the presence of defects in the design patterns code. The *quality focus* concerns fault proneness of design pattern-related source code. The *perspective* of the study is of a researcher interested to investigate to what extent the benefits introduced by design patterns are counterbalanced by the increased fault proneness in the crosscutting concerns they induce.

4.1 Context description

The *context* of this study consists of three open-source systems, *JHotDraw*, *ArgoUML*, and *Eclipse-JDT* which can be classified as a small, medium, and large system, respectively.

JHotDraw[4] is a Java framework for drawing 2D graphics. The project started in October 2000 with the main purpose of showing the design pattern programming in a real context. We extracted a total of 177 snapshots from release 5.2 to release 5.4 BETA2, in the time interval between March 2001 and February 2004. In that interval the size of the system grew almost linearly from 13.5 KNLOC at release 5.2 to 36.5 KNLOC at release 5.4 BETA2. With a similar trend the number of detected design patterns grew from 93 at release 5.2 to 158 at release 5.4 BETA2. Such design patterns were never deleted from the system.

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

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

ArgoUML[5] is an open source UML modelling 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 an interval we extracted 5525 snapshots. The number of classes grew almost linearly from 801, in September 2000, to 2374, in December 2005, except in the interval between releases $0.11.x$ and $0.13.x$, where an almost exponential increment can be observed. The number of KNLOC has grown from 99.5 to 159.5 in the same interval. Although the total number of detected design patterns grew from 117 to 255, for some of them—Factory-Methods, Singletons, and Adapter-Commands—such a number oscillates over the time.

Eclipse Java Development Toolkit (Eclipse-JDT)[6] is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse platform. The number of CVS accounts is 47, which is probably the number of different developers that contribute to the project. We extracted 19750 snapshots in the time interval between November 2001 and June 2004, when Eclipse 3.0 was released. In this time interval, KNLOC grew almost linearly from 205.5 to 449.4. The number of detected design patterns reaches the maximum of 831 at release 2.1. In such a release, the patterns Factory-Methods, Prototypes, Adapter-Commands, Composites, Decorators, Observers, and Visitors reached their maximum number of detected instances. Instead, the Singletons, State-Strategies, and Template-Methods, grew almost linearly.

Table 1 reports, for each system, the number of extracted snapshots, the range of analysed releases, the

[5] *http://argouml.tigris.org*

[6] *http://www.eclipse.org/jdt/*

Table 1. Case study history characteristics.

System	Snaps	Releases	Bug fixes	KNLOC	Classes
JHotDraw	177	5.2–5.4B2	38	13.5–36.3	164–489
ArgoUML	5525	0.9–0.20	930	99.5–159.5	801–2373
Eclipse-JDT	19750	1.0–3.0	7010	205.5–534.4	2089–6949

Table 2. Detected design patterns.

System	Design Patterns										Total
	Creational			Structural			Behavioural				
	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

number of bug-fixes, 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 across the analysed 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.

4.2 Precision of design pattern detection

As noticed, our study relies on design patterns as detected by the Tsantalis *et al.* [28] tool. Of course, our results can be affected by the precision and recall of this tool, since the tool could detect false positive or produce false negatives, i.e., it might not detect design patterns that actually exist in the analysed software systems.

In their work Tsantalis *et al.* [28] showed that for JHotDraw—the only open source system in our knowledge where design patterns are documented—the precision is 100% and recall is 100%, i.e., all

the documented design patterns are recovered. The precision is always 100%, i.e., there are no false positive, except for Factory Method where it is 66.7%, and for State-Strategy where it is 95.6%. To further extend the results of Tsantalis *et al.*, we also manually inspected all the 259 design patterns detected in release 0.14 of ArgoUML (the release with higher number of design pattern instances). The inspection was performed by at least 2 independent inspectors for each design pattern instance, and a discussion was held if their views conflicted. The results of our manual inspection indicated a precision of 86%, with a minimum of 70% for Adapter-Commands.

4.3 Research questions

This empirical study aims at answering the following two research questions:

- **RQ1:** *Is there any correlation between the scattering degree of pattern-induced crosscutting concern and the number of defects in the design pattern code?* This research question aims at investigating whether the number of defects in the design pattern source code increases with the number of clients, i.e., when design pattern clients become a spread crosscutting concern.

- **RQ2:** *Does the correlation between scattering degree and number of defects investigated in* **RQ1** *depend on the kind of design pattern?* This research question investigates whether some particular kinds of design patterns exhibit a different number of defects when clients become a spread crosscutting concern.

From the above research questions, it is possible to formulate the following hypotheses to be tested:

- H_{01} there is no significant relationship between the scattering degree of design pattern clients and the number of defects in the design pattern classes.

Table 3. JHotDraw – Descriptive statistics of scattering degree and number of defects.

PATTERN	Scattering degree			Number of defects		
	MIN-MAX	MEAN	MEDIAN	MIN-MAX	MEAN	MEDIAN
Factory-Method	2.3-12.9	5.7	4.2	1-5	3	2
Prototype	3.5-13.0	6.7	5.2	1-6	4	4
Singleton	2.0-2.0	2.0	2.0	1-1	1	1
Adapter-Command	1.0-20.2	7.0	5.3	1-6	3	3
Composite	6.3-7.4	6.9	6.9	5-6	5	5
Decorator	0.7-12.0	4.1	3.2	1-3	2	1
Observer	1.0-10.4	4.6	3.9	1-6	3	3
State-Strategy	–	–	–	1-6	3	3
Template-Method	–	–	–	1-5	2	1
Visitor	–	–	–	–	–	–

- H_{02} there is no significant relationship between the kind of pattern and the number of defects the design pattern classes exhibit.

- H_{03} there is no significant interaction between the kind of pattern and the scattering degree of design pattern pattern client code.

4.4 Variable selection

The variables of interest for this study are the *number of defects* $d(DP_k)$ in design pattern classes, and the induced crosscutting concern *scattering degree* $sd(DP_k)$. Descriptive statistics for these variables are reported in Table 3, Table 4, and Table 5 for JHotDraw, ArgoUML, and Eclipse-JDT, respectively.

Number of defects

The literature reports approaches to classify whether a source code change is a bug fix or not. Such methods search for keywords such as “Fixed” or “Bug” [25] occurring in the CVS or SVN notes. Once

Table 4. ArgoUML – Descriptive statistics of scattering degree and number of defects.

Pattern	Scattering degree			Number of defects		
	Min-Max	Mean	Median	Min-Max	Mean	Median
Factory-Method	0.0-0.5	0.2	0.1	1-61	17	3
Prototype	0.0-1.0	0.6	0.5	10-83	57	59
Singleton	0.0-58.5	1.7	0.7	1-61	16	17
Adapter-Command	0.0-6.1	0.8	0.4	1-85	23	16
Composite	0.0-0.0	0.0	0.0	1-1	1	1
Decorator	0.0-0.6	0.1	0.0	1-31	7	2
Observer	0.0-58.5	7.9	0.7	1-11	4	2
State-Strategy	0.0-4.0	0.6	0.2	1-62	18	10
Template-Method	0.0-0.9	0.2	0.1	1-34	13	8
Visitor	0.0-0.0	0.0	0.0	2-2	2	2

a commit is classified as a bug fixing, it is counted as belonging to a design pattern if the change affects any of the design pattern classes, including the main pattern participants identified by the design pattern detection tool [28] and their subclasses. In the following we indicate with B the set of snapshots where a design pattern instance undergoes a bug fixing.

Scattering degree

A number of metrics have been proposed to measure crosscut attributes, such as the concern diffusion, level of concentration, scattering degree, and degree of focus [9, 10, 14]. In the context of our analysis we are interested in the degree of scattering of a design pattern over its clients, that is the number of *callers* spread among different classes with respect to the number of *callees* main participants of a design pattern. This measure is related to the fan-in metric, a measure of the number of methods that call some other method (a potential symptom of concern scattering across modules) [23].

We define the average scattering degree of a design pattern instance as follows:

Table 5. Eclipse-JDT – Descriptive statistics of scattering degree and number of defects.

PATTERN	Scattering degree			Number of defects		
	MIN-MAX	MEAN	MEDIAN	MIN-MAX	MEAN	MEDIAN
Factory Method	0.4-13.5	5.3	4.8	1–89	13	6
Prototype	2.5-8.4	5.0	4.7	3–20	11	11
Singleton	1.1–21.1	5.3	4.3	1–51	10	7
Adapter-Command	0–35.9	5.9	4.5	1–98	10	6
Composite	2.0-7.9	3.8	3.3	3–22	9	3
Decorator	0.5-28.7	5.2	3.7	1–34	10	8
Observer	1.0-34.9	7.7	5.3	1–101	17	10
State-Strategy	0.0-43.0	6.2	4.7	1–77	10	6
Template Method	0.0-41.7	5.1	3.7	1–98	12	6
Visitor	0.4-27.5	7.4	7.1	3–14	7	6

$$sd(DP_k) = \frac{\sum_{i \in B \cap \Gamma(DP_k)} caller(C_k^{i-1})}{\sum_{i \in B \cap \Gamma(DP_k)} callee(M_k^{i-1} \cup P_k^{i-1})}$$

where $caller(C_k^{i-1})$ is the number of callers, and $callee(M_k^{i-1} \cup P_k^{i-1})$ is the number of callee design pattern participants, in the snapshot preceding a bug fix. Such a number is greater than zero and has the following interpretation:

- $0 < sd(DP_k) < 1$, the number of callees is greater than the number of callers. In such a case the design pattern is not a spread crosscutting concern. We avoid to consider such cases as they are out of scope of this paper.

- $sd(DP_k) = 1$, the number of callees is equal to the number of callers. Such a threshold indicates the case where design patterns and their clients are equally distributed.

- $sd(DP_k) > 1$, the number of client callers increases and are greater than the number of pattern callees. The pattern becomes a spread crosscutting concern when $sd(DP_k) >> 1$.

5 Results and discussion

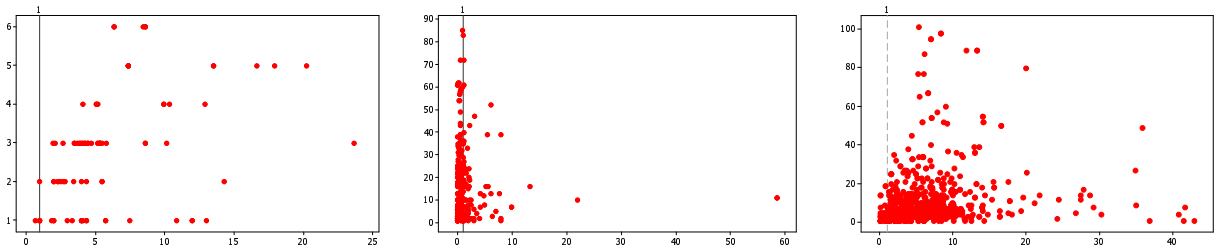
This section reports results of the empirical study defined in Section 4, with the aim of answering the research questions of Section 4.3. To allow for replication, we make available raw data used for the analyses reported below[7].

5.1 RQ1: Is there any correlation between the scattering degree of pattern-induced crosscutting concern and the number of defects in the design pattern code?

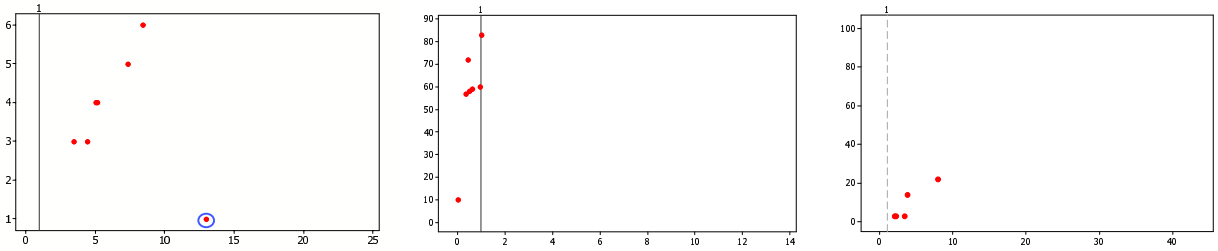
To answer **RQ1**, we analyse the correlation between the number of defects and the scattering degree of crosscutting concerns $sd(DP_k)$ induced by design patterns. In particular, we compute the Spearman (non-parametric) correlation between the number of defects and the scattering degree. Results are reported in Tables 6, 7, and 8, for JHotDraw, ArgoUML, and Eclipse-JDT respectively (significant correlations are shown in bold face). Rows without values for some systems (e.g., Composite and Visitor for JHotDraw) are related to patterns that do not induce scattering degree, as it can also be noticed from Table 3 and Table 4. Finally, Figure 2 shows—for the entire data sets (All) and for specific design patterns for which the correlation level is higher than for others—scatter plots of scattering degree $sd(DP_k)$ (x-axis) and number of defects $d(DP_k)$ (y-axis).

A correlation coefficient can be considered very low if ≤ 0.50, low between 0.51 and 0.79, moderate between 0.80 and 0.89, high if ≥ 0.90. We also compute the statistical significance of correlations, to determine the likelihood that our results have been obtained by chance.

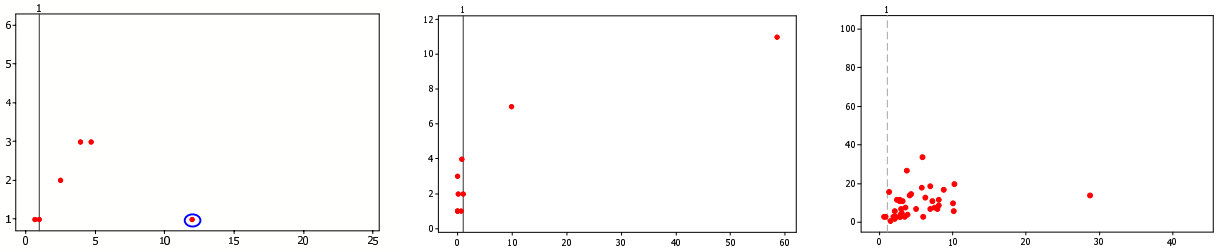
[7] *http://www.rcost.unisannio.it/mdipenta/scattering.zip*



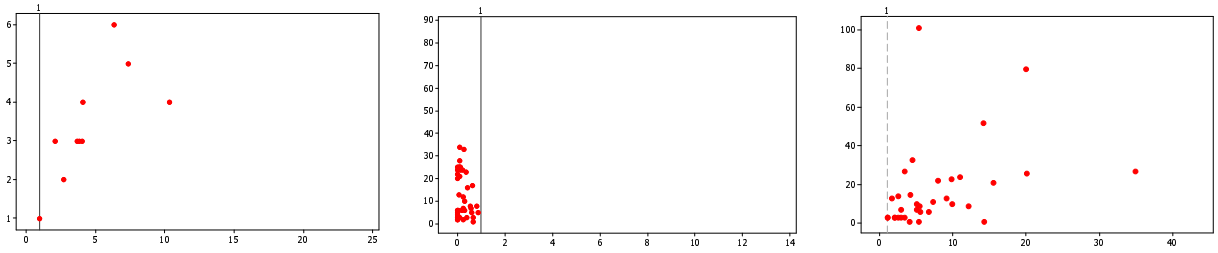
(a) JHotDraw – All (e) ArgoUML – All (i) Eclipse-JDT – All



(b) JHotDraw – Prototype (f) ArgoUML – Prototype (j) Eclipse-JDT – Composite



(c) JHotDraw – Decorator (g) ArgoUML – Observer (k) Eclipse-JDT – Decorator



(d) JHotDraw – Observer (h) ArgoUML – Template-Method (l) Eclipse-JDT – Observer

Figure 2. Scatter plots of number of defects (y) vs. scattering degree (x).

Table 6. JHotDraw – Spearman rank correlation between number of defects and scattering degree.

PATTERN	Spearman corr.	p-value
All	**0.51**	< 0.01
Factory-Method	0.49	0.32
Prototype	**0.97**	< 0.01
Singleton	–	–
Adapter-Command	**0.66**	< 0.01
Composite	–	–
Decorator	**0.94**	0.01
Observer	**0.87**	< 0.01
State-Strategy	**0.39**	0.04
Template-Method	–	–
Visitor	–	–

For JHotDraw (Table 6), considering the overall data set, the Spearman correlation is significant, although low, for all kinds of patterns (0.51, p-value$<$ 0.01). Correlation has been computed after removing outliers of Prototype and Decorator, clearly visible from scatter plots in Figure 2(b) and Figure 2(c)

The remaining rows detail the values of the Spearman correlation for each kind of design pattern. It can be noted that, for some patterns, it is not possible to compute the correlation due to the limited number of instances detected (see Table 2). There is a statistically significant correlation for Prototype (high), Adapter-Command (low), Decorator (high), Observer (moderate), and State-Strategy (low). Values are not significant for Factory-Method. This is mainly due to the limited number of instances, as the scatter plots shown in Figure 2(b) and Figure 2(c) suggest the presence of such a relationship.

The regression analysis performed on JHotDraw data indicates the presence of a significant linear regression for the following kinds of design patterns:

Template, with $R^2 = 99.30\%$ (p-value < 0.01):

$$d(DP_k) = -4.10 + 1.23 \cdot sd(DP_k)$$

Observer, with $R^2 = 50.60\%$ (p-value = 0.02):

$$d(DP_k) = 1.72 + 0.37 \cdot sd(DP_k)$$

Adapter, with $R^2 = 48.50\%$ (p-value < 0.01):

$$d(DP_k) = 1.47 + 0.19 \cdot sd(DP_k)$$

The above regression equations indicate that when the scattering degree increases, the number of defects in the pattern increases linearly with it, no matter what is the size of the design pattern code. The R^2 (coefficient of determination) is the square of the sample correlation coefficient between the outcomes and their predicted values. It measures how well the regression line approximates the actual data points (R^2=100% indicates a perfect fit).

ArgoUML results are shown in Table 7. The overall correlation for all data is negative (-0.28) and not significant. When considering each kind of pattern separately, we find a significant correlation for Prototype (low/moderate), Observer (moderate) and Template-Method (low). This indicates that, for ArgoUML, the presence of a correlation between the number of defects and the design pattern client scattering degree depends on the nature of the particular design pattern. Interestingly, patterns such as Factory-Method, Adapter-Command and Decorator, for which the correlation is not significant, are not necessarily less change-prone; on the contrary, as shown in [1] they are crucial patterns for this system and also the most frequently changed.

For both Prototype and Observer, it is possible to build a linear regression model:

Table 7. ArgoUML – Spearman rank correlation between number of defects and scattering degree.

PATTERN	Spearman corr.	p-value
All	-0.28	< 0.01
Factory-Method	0.32	0.68
Prototype	**0.79**	0.04
Singleton	-0.52	< 0.01
Adapter-Command	0.19	0.34
Composite	–	–
Decorator	-0.50	1.00
Observer	**0.81**	0.04
State-Strategy	0.04	0.80
Template-Method	**0.51**	< 0.01
Visitor	–	–

Prototype, with $R^2 = 60.70\%$ (p-value = 0.04):

$$d(DP_k) = 26.80 + 52.60 \cdot sd(DP_k)$$

Observer, with $R^2 = 75.40\%$ (p-value < 0.01):

$$d(DP_k) = 1.80 + 0.53 \cdot sd(DP_k)$$

It is necessary to note that classes using a Singleton do not really induces a crosscutting concern, as they just invoke Singleton methods, often in a different way. Such *callers* are not strictly related each other in a concern.

Eclipse-JDT results are shown in Table 8. Results indicate that only for the Composite a moderate/high correlation (0.89) can be found, while it is low in other cases. Results of Composite should be interpreted with caution, as the number of involved design pattern instances is very low (five). The correlation is higher than for other patterns (though still low) for Decorator (0.50), Singleton (0.49), and

Table 8. Eclipse-JDT – Spearman rank correlation between number of defects and scattering degree.

PATTERN	Spearman corr.	p-value
All	**0.29**	<0.01
Factory Method	**0.27**	0.04
Prototype	0.43	0.22
Singleton	**0.49**	<0.01
Adapter-Command	**0.34**	<0.01
Composite	**0.89**	0.04
Decorator	**0.50**	<0.01
Observer	**0.48**	<0.01
State-Strategy	**0.14**	0.02
Template Method	**0.38**	<0.01
Visitor	0.17	0.16

Observer (0.48). Scatter plots for these patterns—excluding the Singleton, for the reasons explained above—are shown in Figure 2. Finally, we performed a regression analysis on the pattern having a high correlation, i.e., the Composite, obtaining the following model:

$R^2 = 84.98\%$ (p-value = 0.02):

$$d(DP_k) = -3.79 + 3.34 \cdot sd(DP_k)$$

For other patterns (e.g., Singleton, Decorator, or Observer) the obtained linear regression model is significant, but less useful to explain the variability (i.e., the R^2 is below 10%).

Overall, the obtained results allow for rejecting H_{01} for JHotDraw, although the correlation is moderate or high only for some patterns. There are patterns for which, mainly due to the limited number of instances, the correlation is not statistically significant. For ArgoUML, it is possible to reject the null hypothesis only for Prototype, Observer, and Template-Method (for Template-Method, however, the

correlation is low). The smaller scattering degree of ArgoUML pattern clients (Table 4), probably due to the different design adopted, makes the correlation between scattering degree and defects significant only in a few cases. Finally, although it is possible to reject H_{01} for Eclipse-JDT, it must be noted that the correlation level obtained is generally low.

5.2 RQ2: Does the correlation between scattering degree and number of defects investigated in RQ1 depend on the kind of pattern?

To answer **RQ2**, we analyse the influence of the kind of pattern on the relationship between scattering degree and number of defects. To this aim, we perform a two-way Analysis of Variance (ANOVA). This kind of test is used to analyse the effect of a co-factor—in our case the kind of pattern—on the dependent variable (defect-proneness) and its interaction with the main factor (crosscutting concern scattering degree). Results are shown in Table 9 and Table 10, and Table 11 for JHotDraw, ArgoUML, and Eclipse-JDT, respectively.

Results indicate that, for all the three systems, the kind of pattern has a significant influence on the number of defects, and also the interaction between the two variables is significant. Overall, it is possible to reject the null hypothesis H_{02}, i.e., there is a significant effect of the kind of pattern on the number of defects, and the null hypothesis H_{03}, i.e., there is a significant interaction between the scattering degree and the pattern type. This indicates that the relationship between the number of defects in the design pattern code and the scattering degree significantly varies with the kind of pattern. This confirms what we already noticed in **RQ1**, i.e., especially for ArgoUML and Eclipse-JDT we have different results for different kinds of patterns.

Table 9. JHotDraw – Two-way ANOVA of number of defects by scattering degree & kind of pattern.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Scattering	1	35.97	35.97	24.83	< 0.01
Pattern	9	35.97	3.99	2.76	< 0.01
Scattering:Pattern	7	23.77	3.39	2.34	0.03
Residuals	69	99.96	1.44		

Table 10. ArgoUML – Two-way ANOVA of number of defects by scattering degree & kind of pattern.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Scattering	1	559	559	3.04	0.08
Pattern	7	13880	1983	10.77	< 0.01
Scattering:Pattern	7	4251	607	3.29	< 0.01
Residuals	283	52089	184		

5.3 Discussion

It is known that a key benefit of using design patterns is their ability to anticipate changes regarding both new and existing requirements, maximising reuse and making the system robust to such changes [13]. However, the use of design patterns induces the presence of crosscutting concerns, as also documented by Marin *et al.* [23]; changes occurring in these crosscutting concerns are, very often, related to changes occurring to design pattern code [2]. When developers need to change crosscutting features spread among different design pattern participants and clients, bug fixing is more difficult to perform [9].

As in the case of Eaddy *et al.*, we found that design-pattern induced crosscutting concern scattering degree can have, in some cases, a moderate to high correlation with the presence of defects. Not only, as Eaddy *et al.* found, the scattering degree is correlated to the fault proneness of the crosscutting code; our results also suggest that the scattering degree is correlated to the fault proneness of the design pattern code. A possible explanation can be the following: as the pattern starts to be used by more and more clients—and thus the scattering degree increases—this may require changes to the pattern

Table 11. Eclipse-JDT – Two-way ANOVA of number of defects by scattering degree & kind of pattern.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Scattering	1	5481.57	5481.57	31.19	< 0.01
Pattern	9	3690.24	410.03	2.33	0.01
Scattering:Pattern	9	6429.76	714.42	4.06	< 0.01
Residuals	829	145699.84	175.75		

implementation or even to the pattern interface. Such changes can introduce defects, since they can alter the pattern behaviour other clients are expecting. This partially contradicts some design pattern benefits claimed by Gamma *et al.*, such as the robustness to changes. This finding is also supported by our previous studies [1], where we found that some patterns—and in particular those crucial for the application's purpose—tend to be less robust to changes, i.e., they are subject to changes having a large impact on the pattern client.

For JHotDraw, we found a significant and high correlation for the Decorator pattern. This is not surprising: as the system evolves—and thus the crosscutting concern scattering degree likely increases—the Decorator tends to scatter functionalities across many objects, making the pattern difficult to be understood during maintenance tasks, thus causing the introduction of defects. Gamma *et al.* say:

> *A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.*

Prototypes and Observers exhibit, for both JHotDraw and ArgoUML, a significant and moderate to high correlation between scattering degree and number of defects. The correlation is significant, although low (~ 0.50), for Eclipse-JDT. Observers play a crucial role in JHotDraw, being responsible of

updating the user interface when figure models change [1] and have a similar responsibility in ArgoUML and Eclipse-JDT. Every time there is a need to change a visualisation in JHotDraw, a UML diagram view in ArgoUML, or an Eclipse-JDT GUI option (concrete observers, playing the role of clients in the Observer design pattern), this could have an impact on the model (concrete subjects). Multiple changes to the model triggered by different clients—whose number increases with the scattering degree—can also increase the pattern fault proneness. Moreover, as Gamma *et al.* suggest:

> *. . . dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.*

The high scattering degree of Observers, and the correlated fault proneness, confirm results of Vokáč [29] that, in a different study on different (commercial) software systems, found that, as mentioned in Section 2, Observers tend to be highly correlated with large code structure and because of that be fault prone. Reasons for fault proneness of Prototypes are less intuitive. However, it is known that different clients might require customised prototyping operations, for example parametric *Clone* interface, as indicated by Gamma *et al.*:

> *While some clients are perfectly happy with the clone as is, others will want to initialize some or all of its internal state to values of their choosing. You generally can't pass these values in the Clone operation, because their number will vary between classes of prototypes. Some prototypes might need multiple initialization parameters; others won't need any. Passing parameters in the Clone operation precludes a uniform cloning interface.*

As new clients are introduced—as it can be observed from the increasing scattering degree—there is need for more and more customised prototyping operation, making necessary changes that could potentially introduce defects to the pattern classes.

Eclipse-JDT exhibited a high correlation for Composites, although the number of design pattern instances for which this phenomenon was observed is quite limited (five) to draw general conclusions. By analysing the commit notes, it was found that bug fixes were mainly related to data structures handling Abstract Syntax Trees (ASTs), or to GUI-features, e.g., handling the clipboard. For many of these bug fixes—in particular those related to ASTs—other patterns, in particular Visitors, underwent changes as well. As mentioned in the introduction, Visitors play a crucial role in Eclipse-JDT, for handling code analyses and transformations. However, in a previous work [1] we found that although Visitors change frequently in Eclipse-JDT, the amount of co-changes in classes depending on them—and thus the change impact—is quite limited. This probably because Visitors are only used in specific locations, where a particular code analysis feature is needed. This clearly reduces the design pattern scattering degree, and makes the correlation between scattering degree and number of bug fixes not significant.

Last, but not least, it is important to highlight the difference among results obtained for the three systems. For JHotDraw, a significant—and in some cases high—correlation was found for almost all the design patterns investigated, a high variability of results was found for ArgoUML and Eclipse-JDT. A possible explanation can be found in the different architecture and design the three systems have: MVC with a massive usage of design patterns for JHotDraw, monolithic architecture for ArgoUML, and plugin-based for Eclipse-JDT. In particular, JHotDraw is a system conceived to illustrate the usage of object-oriented design patterns, and for this reason there is a wide usage of all design patterns, and as a consequence a significant correlation, for most of the patterns, between crosscutting concern scattering degree and fault proneness. This is not the case in larger, more realistic systems like ArgoUML or Eclipse-JDT, where different patterns are used in different ways, and can therefore exhibit different levels of fault proneness.

5.4 Threats to Validity

This section discusses threats to validity that can affect the results reported in this paper following a well-known template for case studies [30].

Threats to *construct validity* concern the relationship between theory and observation. They can be due to the measurement performed, in particular related to design pattern identification, analysis of dependencies, the use of change sets, and the measurement of both number of defects and scattering degree. We are aware that our results can be influenced by the performance, in terms of precision and recall, of the Tsantalis *et al.* tool [28]. However, as discussed in Section 4.2, the tool precision is, overall, above 85%, limiting the influence of false positives on our results. Regarding the recall, the number of false negatives is quite low in JHotDraw, where patterns are documented. In case pattern implementations are variants of the simple cliché defined by Gamma *et al.*, some patterns may be missed during the detection phase. The dependence analysis to retrieve the set of pattern clients, as discussed in Section 3.2, considers a conservative set of classes for both pattern clients and targets. More precise analyses could have restricted this set and made our findings less pessimistic. Regarding defects, we rely that the number of bug fixes extracted from a CVS repository is an underestimate, as not all bug fixes are annotated in the CVS commit notes, but precise enough as a manual inspection of bug reports indicated a negligible number of false positives [3].

Threats to *internal validity* can be due to the influence of external factors on the relationship object of the study, i.e., the relationship between design pattern usage scattering degree and number of defects. We analysed the influence of at least one external factor, i.e., the kind of pattern, by means of a two-way ANOVA, and results indicated that such a factor had a significant influence and that it in some cases interacts with the scattering degree. Last, but not least, this is an observational study, where we only

make claims about correlations; we cannot however do any claim about the cause-effect relationship between crosscutting concerns scattering degree and design pattern code fault proneness.

Threats to *external validity* are related to the possibility of generalising 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 analyse further systems to draw more general conclusions. Finally, we considered a subset of 10 out of the 23 patterns from the Gamma *et al.* catalogue.

Regarding *reliability validity*, the source code of the two systems and the design pattern detection tool is publicly available. The way our analyses were performed is described in detail in Section 3, and we made raw data available to allow for replicating statistical analyses.

Conclusions of this study are supported by proper tests, i.e., non-parametric Spearman correlation to perform correlation analysis, linear regression to describe the relationship between number of defects and scattering degree, and two-way ANOVA to analyse the influence of external factors (e.g., kind of design pattern) on the relationship we studied. Although ANOVA is a parametric statistic, it is pretty robust to be applied even for data which is not normally distributed. Results for JHotDraw should be looked carefully because of the limited number of defects involved, although correlation was almost always found significant.

6 Conclusions and Work-in-progress

This paper reported an empirical study—performed analysing the CVS repository of three open source systems, JHotDraw, ArgoUML, and Eclipse-JDT—aimed at investigating on the correlation between the scattering degree of design pattern induced crosscutting concerns and the number of defects in the design

pattern code. For JHotDraw results indicate the presence of a positive and statistically significant relationship, moderate or high for some patterns like Prototype, Decorator and Observer. For ArgoUML, the correlation is statistically significant only for a limited number of patterns—with again a moderate correlation for Prototypes and Observers—also because of the lower scattering degree many pattern clients have for this system. For Eclipse-JDT, although the correlation is, overall, significant, the correlation level is only high for the Composite, and around 0.5 once again for Prototype, Decorator, and Observer.

In summary, results suggest that when patterns induce crosscutting concerns the number of defects on their classes could increase. This highlights the attention on the question regarding the harmfulness of design patterns when their clients become a spread crosscutting concern. Despite it has been claimed that the usage of design patterns can introduce many benefits such as high maintainability and resilience to changes, attention should be paid to effects they induce, e.g., an increase of number of defects in the design pattern classes.

The above finding needs to be considered carefully. It does not indicate that the usage of design patterns is harmful and that alternative design solutions would have lead to a less fault prone code. Previous studies, in fact, found that the scattering degree of crosscutting concerns is correlated with the code fault proneness regardless of the usage of design principles, among others the adoption of design patterns. Thus, it could be claimed that crosscutting concerns induced by design patterns are not more harmful than other crosscutting concerns, but design patterns do not always help in mitigating the presence of defects. This is because, despite the claimed benefits about the capability of design patterns making the system more robust to changes [13], recent studies found that this is not always true especially for patterns playing a crucial role for the application [1].

The analysis process proposed in this paper poses the basis for further studies that can be performed

towards different directions. Above all, it would be necessary to perform further studies aimed at increasing the external validity of our results, by replicating the study on a larger set of software systems. This would be useful not only to confirm or contradict the results presented in this paper, but also to relate these results to results indicating the changeability and criticality of patterns playing a crucial role for the application [1]. Also, future work will aim at relating the change propagation between design patterns and their induced crosscutting concerns we previously studied [2] with the correlation between crosscutting concerns' scattering degree and defect proneness found in this paper, and at relating the presence of design pattern-induced crosscutting concerns with specific kinds of changes occurring in the system, other than defect fixing, for example refactoring activities.

Last, but not least, this study only showed the presence of correlation between design pattern-induced crosscutting concerns and fault proneness, and discussed possible reasons of such a correlation case by case. Understanding whether the crosscutting concerns are actually the cause of fault is something that cannot still be claimed. Future work can look more carefully at that possible cause-effect relationship, by mining information from system change logs.

References

[1] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 385–394, New York, NY, USA, 2007. ACM Press.

[2] L. Aversano, L. Cerulo, and M. Di Penta. Relating the evolution of design patterns and crosscutting concerns. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 180–192. IEEE CS Press, 2007.

[3] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from CVS and Bugzilla repositories: the Mozilla case study. In *Proceedings of the 2007 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2007), October 22-25, 2007, Richmond Hill, Ontario, Canada*, pages 215–228. IBM, 2007.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[5] 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.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] J. O. Coplien. *Software Design Patterns: Common Questions and Answers*. Cambridge University Press, NY, USA, 1998.

[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. IEEE CS Press, 2005.

[9] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, 2008.

[10] E. Figueiredo, A. Garcia, C. Sant'Anna, U. Kulesza, and C. Lucena. Assessing aspect-oriented artifacts: Towards a tool-supported quantitative method. In *9th ECOOP Workshop on Quantitative Approaches in OO Software Engineering (QAOOSE.05), in conjunction with the ECOOP'05 Conference*, pages 58–69, Glasgow, Scotland, 2005.

[11] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison Wesley Professional, 2004.

[12] 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*, pages 13–23. IEEE Computer Society, 2003.

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

[14] 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.

[15] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *TOOLS USA 2001: Software Technologies for the Age of the Internet, 39th International Conference & Exhibition, Santa Barbara, CA, USA, July 29 - August 3, 2001*, pages 296–306. IEEE Computer Society, 2001.

[16] Y.-G. Guéhéneuc and G. Antoniol. DeMIMA: a multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008.

[17] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, 2005.

[18] 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.

[19] R. Harrison, S. Counsell, and R. Nithi. Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems. *Journal of Systems and Software*, 52(2-3):173–179, 2000.

[20] 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. IEEE CS Press, 2003.

[21] IEEE Standards Board. *IEEE Std 610.12.1990, IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[22] 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. IEEE CS Press, 1996.

[23] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Trans. Softw. Eng. Methodol.*, 17(1):#3, 1–37, 2007.

[24] 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.

[25] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[26] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, New York, NY, USA, 2005. ACM.

[27] 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.

[28] 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.

[29] M. Vokáč. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Software Eng.*, 30(12):904–917, 2004.

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

[31] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, page #9. IEEE Computer Society, 2007.

[32] 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.