Advances in Computational Design, *Vol. 2, No. 4 (2017) 293-311*
DOI: https://doi.org/10.12989/acd.2017.2.4.293

Empirical assessment of design patterns' fault-proneness at different granularity levels

Mawal A. Mohammed[1] and Mahmoud O. Elish*[2]

[1]Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
[2]Computer Science Department, Gulf University for Science and Technology, Mishref, Kuwait

(Received March 7, 2017, Revised September 14, 2017, Accepted October 1, 2017)

Abstract. There are several claimed benefits for the impact of design patterns (DPs) on software quality. However, the association between design patterns and fault-proneness has been a controversial issue. In this work, we evaluate the fault-proneness of design patterns at four levels: the design level, category level, pattern level, and role level. We used five subject systems in our empirical study. As a result, we found that, at the design level, the classes that participate in the design patterns are less fault-prone than the non-participant classes. At the category level, we found that the classes that participate in the behavioral and structural categories are less fault-prone than the non-participant classes. In addition, we found that the classes that participate in the structural design patterns are less fault-prone than the classes that participate in the other categories. At the pattern level, we found that only five patterns show significant associations with fault-proneness: builder, factory method, adapter, composite, and decorator. All of these patterns except for builder show that the classes that participate in each one of them are less fault-prone than the non-participant classes in that pattern. The classes that participate in the builder design pattern were more fault-prone than the non-participant classes and the classes that participate in several patterns: the adapter, the composite, and the decorator design patterns. At the role level, the most significant differences were between the classes that participate in some roles and the non-participant classes. Only three pairs of design pattern roles show significant differences. These roles are concrete-product vs. concrete-creator, adapter vs. adaptee, and adapter vs. client. The results recommend the use of design patterns because they are less fault-prone in general except for the builder design pattern, which should be applied with care and addressed with more test cases.

Keywords: design patterns; fault-proneness; software quality

1. Introduction

Design patterns are intended to encapsulate solutions to recurring design problems. They represent valuable design expertise that can be used in documenting and communicating these solutions. They were first introduced by Alexander *et al.* in their book Pattern Language in the context of architecture (Alexander *et al.* 1977). Later, the notion of design patterns was developed

*Corresponding author, E-mail: elish.m@gust.edu.kw in the context of object-oriented software design by Gamma, Helm, Johnson, and Vlissides (GoF) (Gamma *et al.* 1995). In their book, they classified design patterns into three categories: creational, structural, and behavioral. Each category includes a different set of patterns that share some common characteristics. Each pattern has one or more classes. Each class in each design pattern plays a role in that pattern.

Fault-proneness analysis is of a great importance for software quality control and assurance. The ability to analyze the fault-proneness of design patterns can help in minimizing costs and increasing the effectiveness of software testing. Some previous studies have suggested that the majority of faults occur in a few components of the software system (Boehm and Papaccio 1988, Porter and Selby 1990). This can be applicable to design patterns because the different patterns have different structures and different behaviors. These differences might have an effect on their proneness to faults. Some patterns might be more fault-prone than others (Vokac 2004, Ampatzoglou *et al.* 2011, Gatrell and Counsell 2011). Therefore, specifying which design patterns are more fault-prone is of great help for software designers and testers. Analyzing the relationship between design patterns and fault-proneness can provide designers, implementers, and testers with valuable knowledge that can eventually lead to less fault-prone software. For example, software designers and implementers will be able, utilizing this knowledge, to apply design patterns with care so that the produced design is less fault-prone. Testers, as well, can utilize this knowledge to target the problematic design patterns with more test cases.

The production of a software with a minimal number of faults is of special interest to the software community (Pham 2001). As a design construct, design patterns can play an important role in software fault-proneness. However, the relationship between design patterns and fault-proneness has not been fully studied, and the obtained results do not give a clear indicator on this relation. In addition, there is no agreement among the results of previous studies on the tendency of this relationship. Moreover, not all the design patterns, and not all the levels (design, category, pattern, and role) are addressed in the literature. In this work, we decided to go deeper in investigating the relationship between the fault-proneness and the design patterns by addressing the roles within patterns. Doing this enables us to get a deeper insight into the contribution of each role in fault-proneness of each pattern.

The main objective of the current study is to measure and compare the fault-proneness of design patterns on the different levels of design. At the design level, we measure and compare the fault-proneness of the classes that participate in design patterns versus the classes that do not participate in design patterns. At the category level, we measure and compare the fault-proneness among the different categories of design patterns. At the pattern level, we measure and compare the fault-proneness of each design pattern. Finally, at the role level, we measure and compare the fault-proneness of the classes that participate in each role in each design pattern.

2. Literature review

The relationship between the design patterns and the software quality has been discussed in a comparative literature survey in the literature (Ali and Elish 2013). Four quality attributes are addressed in the literature: maintainability (Prechelt *et al.* 2001, Vokac *et al.* 2004, Garzás and Piattini 2009, Juristo and Vegas 2011, Krein *et al.* 2011, Nanthaamornphong and Carver 2011, Prechelt and Liesenberg 2011, Hegedűs *et al.* 2012), change-proneness (Bieman *et al.* 2003, Aversano *et al.* 2007, Gatrell *et al.* 2009), performance (Rudzki 2005, Afacan 2011), and faults

Table 1 Summary of our work compared to the existing works in the literature

Work	Faults Aspect	# of Patterns	Subject Systems	Granularity level
(Vokac 2004)	Fault frequency	5	C++ system	Pattern level
(Ampatzoglou *et al.* 2011)	Fault frequency	11	Java systems	Pattern level
(Gatrell and Counsell 2011)	Fault-proneness	13	C# system	Design level and pattern level
This work	Fault-proneness	17	Java systems	Four levels: design level, category level, pattern level, and role level

(Vokac 2004, Ampatzoglou *et al.* 2011, Gatrell and Counsell 2011). In this study, we will discuss works that have addressed the relationship between the design patterns and faults (Vokac 2004, Ampatzoglou *et al.* 2011, Gatrell and Counsell 2011). (Mayvan *et al.* 2017) conducted a recent systematic mapping of the literature on the state of the art on design patterns. They observed primary studies that have assessed the impact of applying DPs on the quality of the software systems, rather than the quality of DPs themselves, which include (Ampatzoglou *et al.* 2015, Scanniello *et al.* 2015, Jaafar *et al.* 2016). (Vokac 2004) conducted a case study that compared the defect frequency of the classes that participate in five design patterns in a large commercial C++ project: singleton, template method, decorator, observer, and abstract factory. He found that there are significant differences in the defect rates among the different patterns that range from 63% to 154% on average. It was concluded that the complexity of the context is a major factor in the association between the defect rates and the design patterns. Some design patterns, such as singleton and observer tends mostly to be used in complex contexts. This leads the classes that participate in such patterns to be associated with more faults. In contrast, the abstract factory and template method design patterns are associated with less complex contexts, so they are associated with lower fault rates.

(Ampatzoglou *et al.* 2011) compared open-source Java projects with respect to defect frequency and debugging efficiency. The comparison was conducted at the system level. The authors addressed 11 patterns. Overall, they found no correlation between design pattern instances and defect frequency. However, two patterns showed a significant impact: adapter and observer. The adapter pattern had a negative impact on fault frequency. Their explanation was that the adapter pattern is used often in reuse activities. In such activity, the developers reuse pieces of code that they might not fully understand. This can result in faults in the system under development. In contrast, the observer pattern showed a positive impact on fault frequency.

(Gatrell and Counsell 2011) compared the fault-proneness of the classes that participate in the design patterns and the non-participant classes. The subject system was a C# industrial system consisting of 7,439 classes. They addressed 13 patterns and found a marginal difference between the classes that participate in the design patterns and the non-participant classes. In addition, they found that the adapter, factory method, and singleton patterns tended to be the most fault-prone among the studied patterns. Moreover, they found that the classes associated with more faults tend to be more change-frequent. They concluded that the propensity of the classes that participate in the design patterns to change is the reason that the classes that participate in some design patterns are more fault-prone than others.

Table 1 shows a comparison among the works existing in the literature and our work. The comparison was conducted in four aspects: the aspect of the faults addressed in the literature, the number of patterns addressed, the type of the subject systems, and the granularity level.

This paper differs from previous works in many ways. In the literature, the fault-proneness of design patterns is investigated in only one study, as far as we know. That study used only one subject system. In our study, we used five subject systems. In addition, their subject system was a C# system. In our study, we used Java systems. Moreover, we covered 17 patterns, more than all the other studies. Furthermore, we addressed the different levels (the design, category, pattern, and role levels), but other researchers have covered only one level.

3. The empirical study

This section describes the empirical study that was conducted. First, the research hypotheses are stated. Then the data collection process is described, followed by a discussion of the data analysis techniques used in this work. The results are then reported and analyzed. Finally, threats to validity are discussed.

3.1 Research hypotheses

In this study, we test the following hypotheses. For each hypothesis, H_0 represents the null hypothesis and H_1 represents the alternative hypothesis of the null hypothesis:

• Hypothesis 1

- H_0: There is no significant difference in fault-proneness between the classes that participate in design patterns (as a whole) and those that do not participate in any pattern

- H_1: There is a significant difference in fault-proneness between the classes that participate in design patterns (as a whole) and those that do not participate in any pattern

• Hypothesis 2

- H_0: There is no significant difference in fault-proneness among the classes that participate in the different categories of design patterns.

- H_1: There is a significant difference in fault-proneness among the classes that participate in the different categories of design patterns.

• Hypothesis 3

- H_0: There is no significant difference in fault-proneness between the classes that participate in each single pattern and those that do not participate in that pattern.

- H_1: There is a significant difference in fault-proneness between the classes that participate in each single pattern and those that do not participate in that pattern.

• Hypothesis 4

- H_0: There is no significant difference in fault-proneness among the classes that participate in the different roles of each design pattern.

- H_1: There is a significant difference in fault-proneness among the classes that participate in the different roles of each design pattern.

3.2 Data collection and description

In the area of empirical software engineering, data collection is the grand challenge that hinders

Table 2 Summary of subject systems information

Systems	JHotDraw v5.1	JUnit v3.7	Lexi v0.1.1 alpha	Nutch v0.4	PMD v1.8
Size in classes	155	78	24	165	446
% of faulty classes	29%	11.5%	44%	44.8%	52.2%
% of the classes that participate in design patterns	74%	57.6%	28%	13.3%	10.6%

the development of this area. There are three reasons for this problem. First, software development organizations are more concerned with getting the software system working and delivering it to the customers. They are not concerned with collecting data for research purposes. Second, the data-collection process is time-consuming and costly. Third, even if the development organization collects data, the data are mostly kept private within the organization for security purposes (Wohlin 2013) (Bosu and MacDonell 2013).

The previously stated reasons are common in the area of empirical software engineering research. In our case, we found that only three works have investigated the relationship between faults and design patterns (Vokac 2004, Ampatzoglou *et al.* 2011, Gatrell and Counsell 2011). The data used in two of these works are industrial data. These data sets are private. This gives another indication of the scarcity of pattern data. This scarcity was the reason that we restricted our work to Java systems. We could not find systems written in different programming languages with pattern and fault data. We also restricted our work to 17 of the 23 GoF patterns for the same reason. We could not find systems that implemented the other six patterns with pattern and fault data.

Data were collected from five open-source Java systems (JHotDraw v5.1, JUnit v3.7, Lexi v0.1.1 alpha, Nutch v0.4, and PMD v1.8). These systems are of different domains and sizes. The size of each system is 155, 78, 24, 165, and 446 classes, respectively. The percentages of faulty classes in each system are 29%, 11.5%, 44%, 44.8%, and 52.2%, respectively. The percentages of pattern classes in each system are 74%, 57.6%, 28%, 13.3%, and 10.6%, respectively. A summary of this information is shown in Table 2.

We collected the pattern data from the P-Mart repository (Guéhéneuc 2007) and the fault data from the Concurrent Versions System (CVS).

The process of preparing the data for analysis is as follows: First, all the classes that belong to each system are listed. Then we label each class as faulty or not. After that, we label each class as participating in a design pattern or not. If it participates, we state its categories, patterns, and roles.
The reason for choosing the aforementioned subject systems is the availability of their pattern information in the P-Mart repository (Guéhéneuc 2007). The P-Mart repository is a reliable source of pattern data. It has been used by researchers for different purposes (Arcelli Fontana *et al.* 2011), (De Lucia *et al.* 2009), (De Lucia, Deufemia *et al.* 2010), (Bernardi, Cimitile *et al.* 2013), (Guéhéneuc, Guyomarc'H *et al.* 2010). Before considering this option, the literature was surveyed to check the availability of pattern detection tools. We found ten tools (Guéhéneuc and Antoniol 2008), (Jing *et al.* 2007), (Lucia *et al.* 2009), (Niere *et al.* 2002), (Arcelli Fontana and Zanoni 2011), (Nija and Olsson 2006), (Guéhéneuc 2005), (Tsantalis *et al.* 2006), (Smith and Stotts 2003), (Dietrich and Elgar 2007). However, none of them was suitable for our work. For example, some of these tools either provide poor performance or performance evaluations are absent. Some other tools are not capable of detecting all the roles of design patterns. Other tools are only capable of

Table 3 Number of instances of each design patterns

Category	Pattern	#
Creational patterns	Abs. factory	0
	Builder	3
	Factory method	6
	Prototype	2
	Singleton	7
Structural patterns	Adapter	4
	Bridge	2
	Composite	4
	Decorator	2
	Facade	0
	Flyweight	0
	Proxy	1
Behavioral patterns	Chain of resp.	0
	Command	3
	Interpreter	0
	Iterator	3
	Mediator	0
	Memento	2
	Observer	9
	State	2
	Strategy	6
	Template	6
	Visitor	1
Total # of instances		63

Table 4 Descriptive statistics for faulty and participant classes

Systems	# of classes	Total LOC	# of faulty classes	# of participating classes in DPs
All systems	868	85,702	372 (42.8%)	238(27.4%)

detecting a small set of design patterns.

Given the limitations of design patterns detection tools, we decided to search for another source of pattern data. This led us to the P-Mart repository, a reliable source of pattern data that has been used in several works (Arcelli Fontana *et al.* 2011), (De Lucia *et al.* 2009), (De Lucia *et al.* 2010), (Bernardi *et al.* 2013), (Guéhéneuc *et al.* 2010). The P-Mart repository contains design pattern data for nine systems. We considered five of them because we could not find fault data for the other four.

We collected fault data from the development CVS files associated with each subject system. All five subject systems are hosted with their CVS files on the Sourceforge website for open-source projects. CVS files contain the commit entries associated with each system. Each class file in the system has a section in the CVS file, which documents the different commits made to that file. We collected the fault data by examining each commit for each class in each subject system searching for keywords such as "fault," "bug," "defect," etc. The fault density of each class was then computed by dividing the number of class faults by kilo lines of code (KLOC) in each class.

The number of instances of each design pattern in our dataset is shown in Table 3. There are 63 instances of design patterns.

To calculate the descriptive statistics in Table 4, we used the *understand* tool (SciTools 2014) and Microsoft Excel. The Understand tool is used to calculate executable LOC, and Microsoft Excel is used to calculate the percentages of faulty classes and the percentage of the classes that participate in the design patterns. To calculate the percentages of faulty classes, we divide the number of classes that have one or more faults by the number of all classes. The same procedure is done in calculating the percentage of the classes that participate in the design patterns. We divide the number of classes that participate in one or more design patterns by the number of all classes. The number of classes that participate in these instances is 238 (27.4%), as shown in Table 4. In addition, the faulty classes constitute 42.8% of the total number of classes.

3.3 Data analysis procedure

In this study, we evaluate the association between design patterns and fault-proneness. The dependent variable is the fault proneness (faulty or not), and the independent variable is a Boolean variable (participant in a design pattern or not). The independent variable is evaluated at four levels: the design, category, pattern, and role level. We evaluate the significant difference in fault-proneness between the classes that participate in design patterns and the non-participant classes at the design, category, pattern, and role levels.

For the data analysis, we used two non-parametric tests: the Mann-Whitney U test (Mann and Whitney 1947) and the Kruskal-Wallis test (Kruskal and Wallis 1952). To perform these tests, we used the SPSS software. The Mann-Whitney test is used to compare differences between two groups, whilst the Kruskal-Wallis test is used to compare differences among more than two groups. In our case, the Mann-Whitney test was used to compare the differences in fault-proneness between two groups, and the Kruskal-Wallis was used to compare the differences in fault-proneness among more than two groups.

At the design level, the Mann-Whitney test was used to compare the difference in fault-proneness between the classes that participate in design patterns and non-participant classes. In addition, we used the Mann-Whitney test to compare the difference in fault-proneness between the classes that participate in each design pattern and the classes that do not participate in that pattern. At the category level, we used the Kruskal-Wallis test to compare differences in fault-proneness among the classes that participate in the different categories of design patterns. The Kruskal-Wallis test was also used to compare differences among the different roles in each design pattern. The obtained p-values associated with comparing more than two groups using the Kruskal-Wallis test is corrected using the Bonferroni method.

4. Results and discussion

The results of the measurement and comparison of the fault-proneness of design patterns are presented in this section. The investigation of fault-proneness was conducted at four levels as

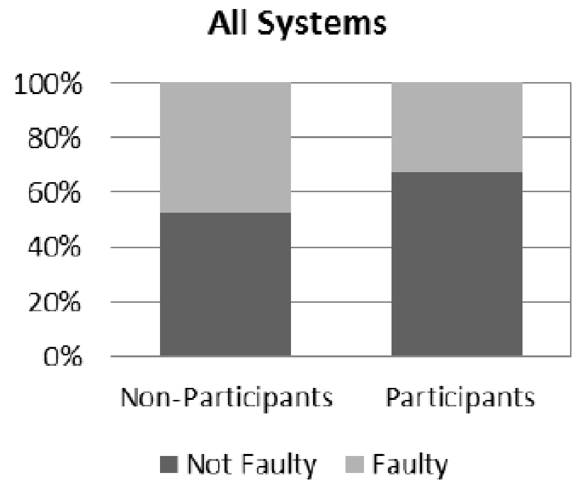

Fig. 1 Fault-proneness comparison of participant vs. non-participant classes

follows.

4.1 Design level

At the design level, we evaluated the difference in fault-proneness between the classes that participate in any design pattern and the classes that do not participate in any design pattern (i.e., participant vs. non-participant). This evaluation gives general insight on the association between design patterns and class fault-proneness.

As Fig. 1 shows, the participant classes were less fault-prone, as a percentage, than the non-participant classes. The p-value of the significance of the difference in the fault-proneness of the participating versus the non-participating classes is 0.000080. Thus, the obtained p-value is significant (<0.05), so we reject the null hypothesis of Hypothesis 1 and accept its alternative hypothesis.

The obtained results say, in other words, that using design patterns improve the quality of software systems by reducing the faults. This might be due to the fact that the design patterns are documented and well-known solutions to design problems. This makes their application and modification safer in terms of introduction of faults compared to the application of ad hoc solutions. In addition to that, the design patterns provide a language for communication among the system's developers and maintainers making understandability of the design and the code easier. However, this depends on the experience of the system developers and implementers with the design patterns.

4.2 Category level

At the level, we measured and compared the fault-proneness of the different categories of design patterns. In addition, we measured and compared the different categories of design patterns to the classes that do not participate in any design pattern. We found that the classes that participate in structural and behavioral design patterns are less fault-prone than the non-participant classes and the classes that participate in the creational design patterns, as shown in Table 5 and

Table 5 P-values associated with evaluating the fault-proneness of the different categories

All systems	Creational	Structural	Behavioral
Non-participant	0.1714	0.0000	0.0080
Creational	-	0.0000	0.0020
Structural	-	-	1

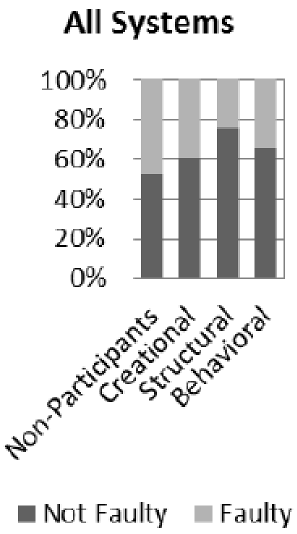

Fig. 2 Fault-proneness comparison of the different categories

Fig. 2. For those cases, we reject the null hypothesis of Hypothesis 2 and accept its alternative hypothesis. The results suggest that structural design patterns tend to be less fault-prone. This might result from the fact that structural design patterns are easier to understand or that structural design patterns are clearer.

The obtained results say in other words that the structural design patterns improve the quality of software by reducing the introduction of faults in the software systems. This might due to that the structural design patterns are solutions to structural problems that considered good and known solutions to such problems. These solutions lead to an overall improved structure. A good structure leads to easier to change systems which leads to less number of faults. The obtained results also suggest that the application of behavioral design patterns leads to fewer faults. This might be due to that the behavioral design patterns work on the behavioral aspects of the system which is difficult to understand. This difficulty might be lowered by characterizing some parts of the behavioral as patterns. These patterns encapsulate the characteristics of the design problems and the solutions making their sharing among the developing and the maintenance teams easier.

4.3 Pattern and role levels

At the pattern level, we evaluated the difference in fault-proneness between the classes that participate in each design pattern and the classes that do not participate. At the role level, we evaluated the difference in fault-proneness among the classes that participate in the different roles of each design pattern.

Table 6 Evaluation results for the creational patterns and their roles

Builder	
Builder classes vs. Non-builder classes	0.0090
Overall role comparison	0.0220
Concrete-builder vs. non-participant	0.0090
Factory Method	
Factory method classes vs. non-factory classes	0.0284
Overall role comparison	0.0110
Concrete-product vs. non-participant	0.0010
Concrete-product vs. concrete-creator	0.0130
Prototype	
Prototype classes vs. non-prototype classes	0.6373
Overall role comparison	0.8910
Singleton classes vs. non-singleton classes	0.9951

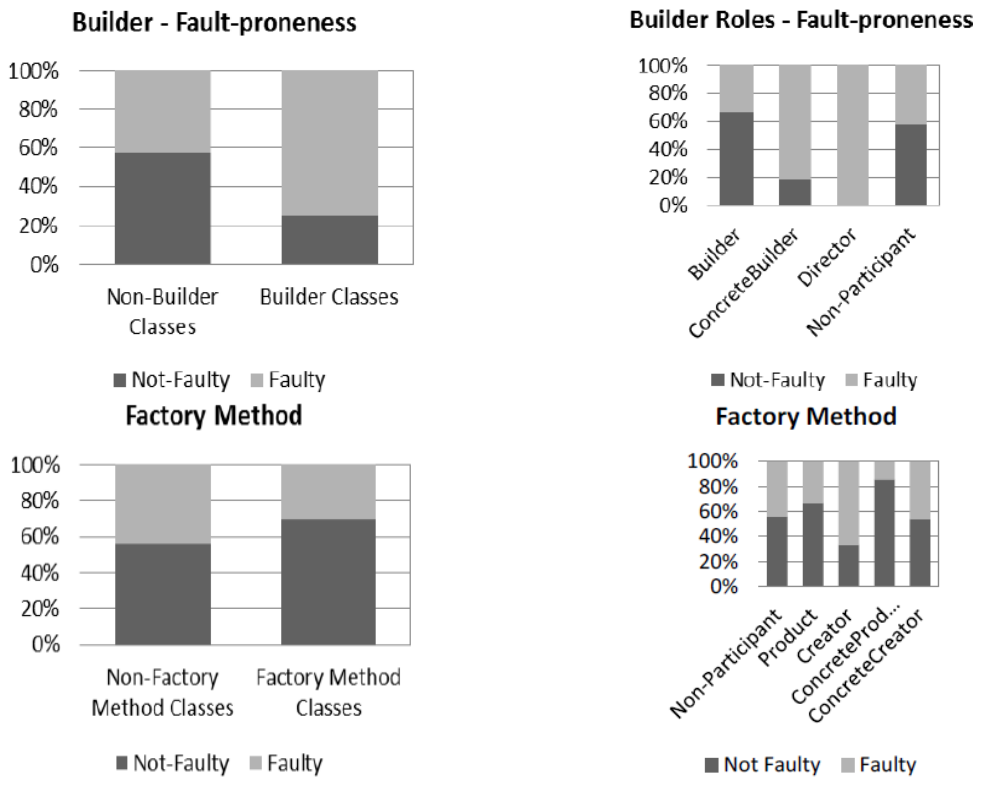

Fig. 3 Comparison of the fault-proneness of the creational design patterns and their roles

4.3.1 Creational design patterns

The evaluation of the difference in fault-proneness between the builder classes and the non-builder classes, and the factory method and the non-factory method classes resulted in significant

Table 7 Evaluation results for the structural design patterns and their roles

Adapter	
Adapter classes vs. non-adapter classes	0.0054
Overall role comparison	0.0020
Adapter vs. non-participant	0.0000
Adapter vs. adaptee	0.0370
Adapter vs. client	0.0080
Composite	
Composite classes vs. non-composite classes	0.0032
Overall role comparison	0.0090
Leaf vs. non-participant	0.0030
Decorator	
Decorator classes vs. non-decorator classes	0.0008
Overall role comparison	0.0030
Concrete-decorator vs. non-participant	0.0290
Concrete-component vs. non-participant	0.0010
Bridge	
Bridge classes vs. non-bridge classes	0.8672
Overall role comparison	0.1960
Proxy	
Proxy classes vs. non-proxy classes	0.1323
Overall role comparison	0.5190

p-values, as reported in Table 6. The obtained results suggest that there is a significant difference in fault-proneness between the builder classes and the non-builder classes, and the factory method and the non-factory method classes. Therefore, for these two patterns, we reject the null hypothesis of Hypothesis 3 and accept its alternative hypothesis. As shown in Fig. 3, the classes that participate in the builder design pattern are more fault-prone than the non-builder classes and the classes that participate in the factory method design pattern are less fault-prone than the non-factory method classes. The situation was different for the prototype and the singleton patterns. The p-values associated with evaluating differences in fault-proneness between the participant and the non-participant classes of the prototype and the singleton patterns are insignificant, as reported in Table 6. As indicated, the obtained results indicate that there are no significant differences in the fault-proneness at the pattern level. Therefore, for these two patterns, we accept the null hypothesis of Hypothesis 3.

Evaluating the difference in fault-proneness among the different roles of builder classes, and among the different roles of the factory method classes result in significant differences, as shown in Table 6. For the builder pattern, only one pair of roles results in a significant difference: concrete-builder vs. non-participant. For the factory method, only two pairs of roles show significant differences: concrete-product vs. non-participant and concrete-product vs. concrete-creator. Therefore, for these three pairs of roles, we reject the null hypothesis of Hypothesis 4 and accept its alternative hypothesis. As shown in Fig. 3, the concrete builder classes are more fault-

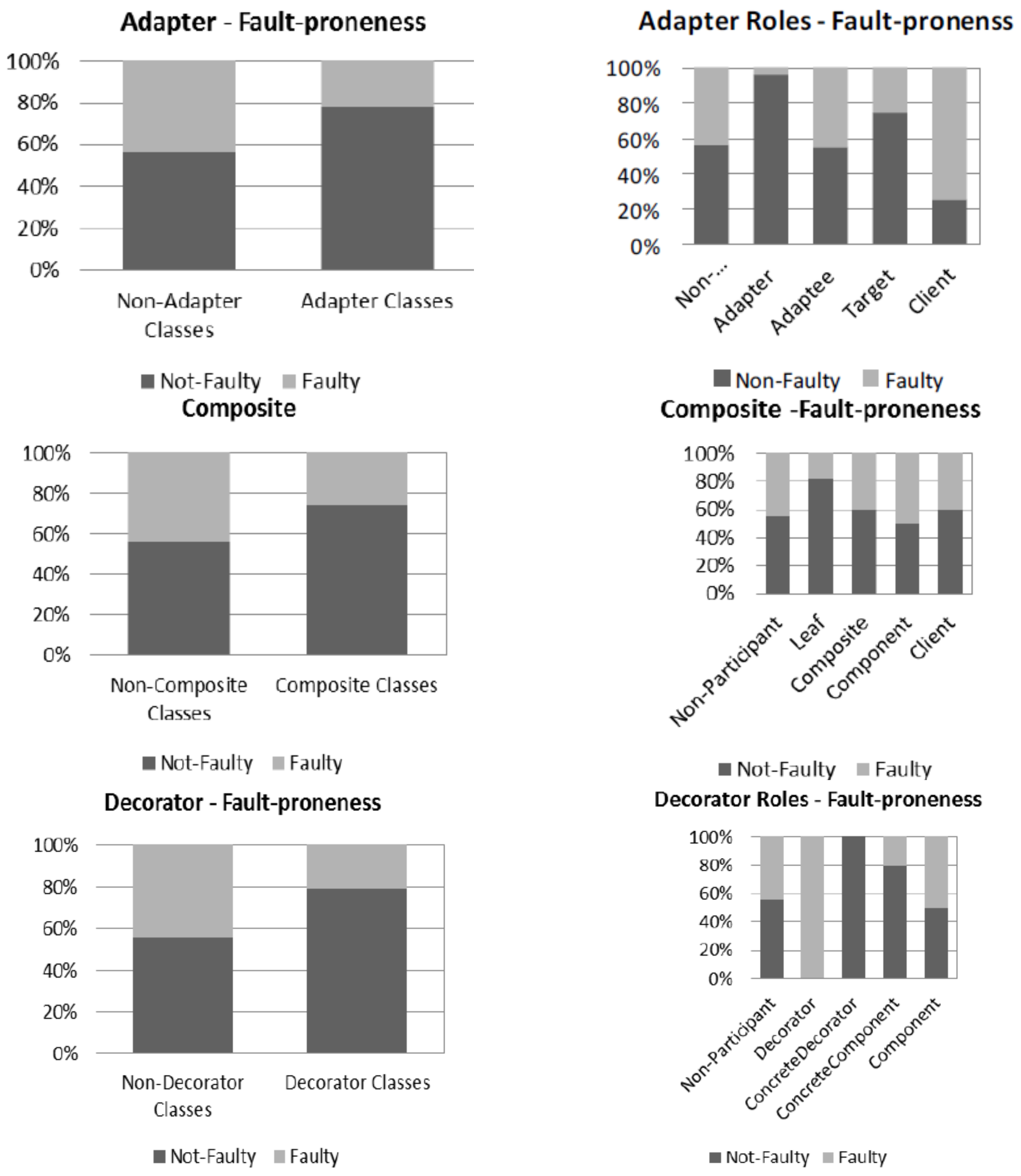

Fig. 4 Comparison of the fault-proneness of the structural design patterns and their roles

prone than the non-participant classes in builder patterns we can also see that the classes that participate in the concrete-product role are less fault-prone than those that participate in the concrete-creator role and less fault-prone than the non-participant classes. For the prototype design pattern, there is no significant difference among the different roles. For the singleton, it has only one role.

4.3.2 Structural design patterns

The p-values associated with the fault-proneness evaluation of the structural design patterns

Table 8 Evaluation results for the behavioral design patterns and their roles

Command	
Command classes vs. non-command classes	0.5772
Overall role comparison	0.0920
Iterator	
Iterator classes vs. non-iterator classes	0.8155
Overall role comparison	0.6120
Memento	
Memento classes vs. non-memento classes	0.1949
Overall role comparison	0.1490
Observer	
Observer classes vs. non-observer classes	0.6258
Overall role comparison	0.4670
State	
State classes vs. non-state classes	0.5430
Overall role comparison	0.2630
Strategy	
Strategy classes vs. non-strategy classes	0.5130
Overall role comparison	0.3090
Template Method	
Template method classes vs. non-template classes	0.5812
Overall role comparison	0.2180
Visitor	
Visitor classes vs. non-visitor classes	0.8930
Overall role comparison	0.4270

reveal that there are significant differences in fault-proneness at the pattern level and at the role level for the adapter, composite, and composite design patterns, as reported in Table 7. As shown in Fig. 4, the participant classes are less fault-prone than the nonparticipant classes in all of these patterns. Therefore, for these patterns, we reject the null hypothesis of Hypothesis 3 and accept its alternative hypothesis. For the bridge and the proxy patterns, the p-values associated with evaluating the difference between the participating and the non-participating classes are not significant. Therefore, we accept the null hypothesis of Hypothesis 3 for these two patterns.

In evaluating the difference among the classes that participate in the different roles, we found that three pairs of roles are associated with significant differences: adapter vs. non-participant, adapter vs. adaptee, and adapter vs. client. As shown in Fig. 4, the classes that participate in the adapter design pattern are less fault-prone than those that participate in the non-participant, adaptee, and client roles. For the composite pattern, only one pair of roles shows significant difference: leaf vs. non-participant. As shown in Fig. 4, the classes that participate in the leaf role are less fault-prone than the non-participant classes. For the decorator design pattern, two pairs of roles show significant differences: concrete-decorator vs. non-participant, and concrete-component vs. non-participant. As shown in Fig. 4, the classes that participate in the concrete-decorator and

Table 9 comparison of the fault-proneness of the different patterns comparison

Pair	p-value
Decorator vs. Builder	0.011
Adapter vs. Builder	0.035
Composite vs. Builder	0.035

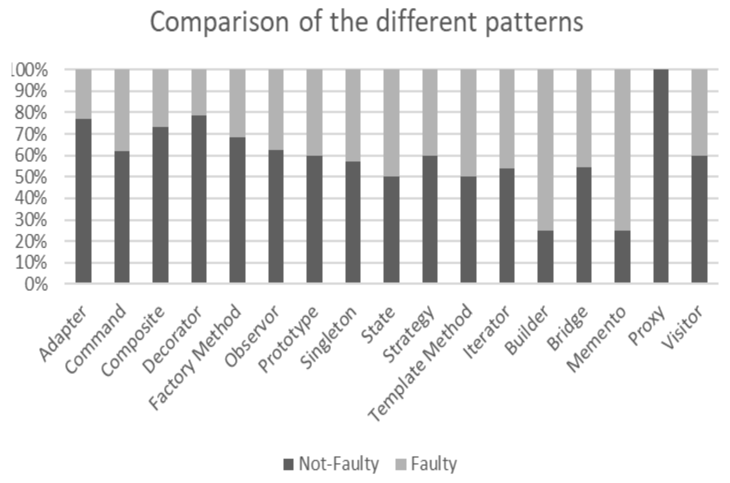

Fig. 5 Comparison of the fault-proneness of the different patterns

concrete-component classes are less fault-prone than the non-participant classes. Therefore, for these pairs of roles, we reject the null hypothesis of Hypothesis 4 and accept its alternative hypothesis. For the bridge and proxy patterns, there is no significant difference in fault-proneness among their roles. Therefore, for these two patterns, we accept the null hypotheses of Hypothesis 4.

4.3.3 Behavioural design patterns

The p-values associated with evaluating the differences in the fault-proneness of all the behavioral design patterns are insignificant at the pattern level and at the role level, as shown in Table 8. These values suggest that there are no significant differences in the fault-proneness of the classes that participate in each design pattern and the non-participant classes in that pattern. Therefore, we accept the null hypothesis of Hypothesis 3 for all the behavioral design patterns. In addition, the results suggest that there are no significant differences in the fault-proneness among the different roles of each pattern. Therefore, we accept the null hypothesis of Hypothesis 4 as well for these cases.

4.4 Design patterns comparison

In this section, we compare the difference in fault-proneness among the different design patterns. We found that, among 136 pairwise tests, only three pairs of patterns show significant difference: Decorator vs. Builder, Adapter vs. Builder, and Composite vs. Builder. In Table 9, we presented the results associated with these pairs. We can see in Fig. 5 that the classes associated with the builder design pattern are more fault-prone than the classes that participate in the decorator, the adapter, and the composite design patterns.

The previous few sub-sections presented the results associated with evaluating the difference in fault-proneness in the pattern level and in the role level. The obtained results suggest that the use of 4 design patterns improve the quality of software systems by reducing faults. These patterns are: adapter, composite, decorator, and factory method. The first three patterns are structural patterns. The obtained improvement might be due to the improvement in the structure of the software system as we mentioned in section 4.2. The improvement associated with the fourth pattern, the factory method, might be due to the context in which the factory method pattern works. The factory method pattern eases the creation of objects by defining an interface for creating objects but the instantiation of the class is let to the subclasses. The situation was different with the builder pattern, the builder pattern classes were more fault-prone than the non-participant classes and also than the classes that participate in several patterns: the adapter, the composite, and the decorator design patterns. we think the reason for that might stem from the context of the builder pattern. The builder pattern work with the construction of complex objects. This complexity might be the reason that increases the faults of the builder classes. The roles that show significant difference were mostly concrete roles. These roles implement functionality which makes them subject to change which is different from the abstract roles that define interfaces only. This might be the reason for the concrete classes to be more fault-prone than the other roles. However, this was not general conclusion since that some of the concrete classes were less fault-prone than the other roles and less than the non-participant classes.

4.5 Threats to validity

Construct validity. The threat to the construct validity of this experiment is associated with the fault collection. We know that discovering all faults in a system seems impossible. There might be some other faults that have not been discovered in the subject systems. However, in our case, the subject systems are popular and widely used open-source systems. They have been also used in several empirical studies in the literature. We thus believe that the data sets that were collected from them are reliable.

Internal validity. Internal validity is the degree to which the observed effects depend only on the intended experimental variables. The major threat to the internal validity of this work stems from the developer background. In fact, we are not sure whether the developers of these systems are well trained to work with design patterns. This could be a serious threat to the validity of the work in case we were studying a cause-and-effect relationship. In such a case, we would be required to control every variable. Since we are not studying a cause-and-effect relationship, we do not need to control every variable. In fact, we cannot control every variable. For example, the level of experience of the developers of the subject systems, this variable can influence the faults that can occur later in the systems. However, it is difficult, if not impossible, to control such variable. In our case, we are investigating whether there is an association between fault-proneness (the dependent variable) and design patterns (independent variables). That is not a cause-and-effect relationship. This kind of studies can serve as a precursor for deeper investigations such as controlled experiments.

External validity. The generalizability of this work is another concern. The ability to generalize the results of such a study-an investigation of the relationship between the design patterns and fault-proneness in object-oriented systems-requires the consideration of many factors. First, systems from different programming languages and/or of different sizes and complexities should be considered. In our study, the subject systems were all written in Java and they are small-medium size systems. Second, we used open-source systems in our experiment. We did not consider commercial systems. To increase the generalizability of the work, commercial systems should be considered as well. However, the obtained results can be considered one step on the road, given the scarcity of pattern data.

5. Conclusions

In this study, we measure and compare the fault proneness of design patterns at four granularity levels: the design, category, pattern, and role levels. We found that the participant classes in the design patterns are less fault-prone than the non-participant classes at the design level. At the category level, we found that the classes that participate in the structural and behavioral categories are less fault-prone than the non-participant classes in these categories and the classes that participate in the creational design patterns. At the pattern level, we found that the classes that participate in the factory method, adapter, composite, and decorator patterns are less fault-prone than the classes that do not participate in these patterns. In the case of the builder design pattern, the classes that participate in it are more fault-prone than the non-participant classes. All of these five patterns are either creational or structural patterns. None of the behavioral patterns shows an association with fault-proneness.

We also found that only these five patterns (i.e., builder, factory method, adapter, composite, and decorator) show significant differences in fault-proneness among their roles. In examining their roles, we found that the concrete-builder role is significantly more fault-prone than the non-participant classes. We think this is the case because the role includes detailed implementations of the methods of the abstract class. For the factory method, the classes that participate in the concrete-product role tend to be significantly less fault-prone than those that participate in the concrete-creator role and the non-participant classes. For the roles of the adapter design pattern, the classes that participate in the adapter role tend to be significantly less fault-prone than those that participate in the adaptee or the client roles. In addition, we found that the classes that participate in the adapter role tend to be significantly less fault-prone than the non-participant classes as well. For the composite pattern, the classes that participate in the leaf role tend to be significantly less fault-prone than the non-participant classes. Finally, the classes that participate in the concrete-decorator and the concrete-component roles in the decorator design pattern are significantly less fault-prone than the non-participant classes.

We recommend the use of design patterns in software design with respect to fault-proneness. However, the builder pattern has a negative association with fault-proneness, which suggests that it should be applied carefully. For patterns that do not show a significant association with fault-proneness, we recommend their application as well since these patterns have other benefits, such as improving programmers' productivity and promoting best practices, and at the same time, they do not seem to be negatively associated with faults.

This work can be extended further by including more patterns and more systems. In addition to that, the work can be extended by including systems developed with different programming languages. Moreover, more quality attributes can be addressed in the future. Design patterns developed with other paradigms, such as aspect-oriented patterns, can be investigated in future works as well.

Acknowledgments

The authors would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM) under Research Grant IN121056. We also would like to thank the anonymous reviewers for their insightful comments and suggestions.

References

Afacan, T. (2011), "State design pattern implementation of a DSP processor: A case study of TMS5416C", *Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems (SIES).*

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, A.S. (1977), *A Pattern Language*, Oxford University Press, New York, U.S.A.

Ali, M. and Elish, M. (2013), "A comparative literature survey of design patterns impact on software quality", *Proceedings of the 4th International Conference on Information Science and Applications (ICISA).*

Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S. and Avgeriou, P. (2015), "The effect of GoF design patterns on stability: A case study", *IEEE Trans. Softw. Eng.*, **41**(8), 781-802.

Ampatzoglou, A., Kritikos, A., Arvanitou, E.M., Gortzis, A., Chatziasimidis, F. and Stamelos, I. (2011), "An empirical investigation on the impact of design pattern application on computer game defects", *Proceedings of the 15th International Academic MindTrek Conference*, Tampere, Finland.

Arcelli Fontana, F., Maggioni, S. and Raibulet, C. (2011), "Understanding the relevance of micro-structures for design patterns detection", *J. Syst. Softw.*, **84**(12), 2334-2347.

Arcelli Fontana, F. and Zanoni, M. (2011), "A tool for design pattern detection and software architecture reconstruction", *Informat. Sci.*, **181**(7), 1306-1324.

Aversano, L., Canfora, G., Cerulo, L., Grosso, C.D. and Penta, M.D. (2007), "An empirical study on the evolution of design patterns", *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia.

Bernardi, M.L., Cimitile, M. and Di Lucca, G.A. (2013), "A model-driven graph-matching approach for design pattern detection", *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE).*

Bieman, J.M., Straw, G., Wang, H., Munger, P.W. and Alexander, R.T. (2003), "Design patterns and change proneness: An examination of five evolving systems", *Proceedings of the 9th International Software Metrics Symposium*.

Boehm, B.W. and Papaccio, P.N. (1988), "Understanding and controlling software costs", *IEEE Trans. Softw. Eng.*, **14**(10), 1462-1477.

Bosu, M.F. and MacDonell, S.G. (2013), "A taxonomy of data quality challenges in empirical software engineering", *Proceedings of the Software Engineering Conference (ASWEC).*

De Lucia, A., Deufemia, V., Gravino, C. and Risi, M. (2009), "Behavioral pattern identification through visual language parsing and code instrumentation", *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*.

De Lucia, A., Deufemia, V., Gravino, C. and Risi, M. (2010), "Improving behavioral design pattern detection through model checking", *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR).*

Dietrich, J. and Elgar, C. (2007), "Towards a web of patterns", *Web Semant.: Sci. Serv. Agent. World Wide Web*, **5**(2), 108-116.

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

Garzás, J. and Piattini, M. (2009), "Do rules and patterns affect design maintainability?", *J. Comput. Sci.*

Technol., **24**(2), 262-272.

Gatrell, M. and Counsell, S. (2011), “Design patterns and fault-proneness a study of commercial C# software”, *Proceedings of the 5th International Conference on Research Challenges in Information Science (RCIS)*.

Gatrell, M., Counsell, S. and Hall, T. (2009), “Design patterns and change proneness: A replication using proprietary C# software”, *Proceedings of the 16th Working Conference on Reverse Engineering*.

Guéhéneuc, Y.G. (2005), “Ptidej: Promoting patterns with patterns”, *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*, Glasgow, U.K.

Guéhéneuc, Y.G. (2007), “P-mart: Pattern-like micro architecture repository”, *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*.

Guéhéneuc, Y.G., Guyomarc’H, J.Y. and Sahraoui, H. (2010), “Improving design-pattern identification: A new approach and an exploratory study”, *Softw. Qualit. Contr*., **18**(1), 145-174.

Guéhéneuc, Y.G. and Antoniol, G. (2008), “DeMIMA: A multilayered approach for design pattern identification”, *IEEE Trans. Softw. Eng*., **34**(5), 667-684.

Hegedűs, P., Bán, D., Ferenc, R. and Gyimóthy, T. (2012), M*yth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability*, Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity, Springer Berlin Heidelberg, **340**, 138-145.

Jaafar, F., Guéhéneuc, Y.G., Hamel, S., Khomh, F. and Zulkernine, M. (2016), “Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults”, *Empir. Softw. Eng*., **21**(3), 896-931.

Jing, D., Lad, D.S. and Yajing, Z. (2007), “DP-Miner: Design pattern discovery using matrix”, *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*.

Juristo, N. and Vegas, S. (2011), “Design patterns in software maintenance: An experiment replication at UPM-experiences with the RESER’11 joint replication project”, *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research (RESER)*.

Krein, J.L., Pratt, L.J., Swenson, A.B., MacLean, A.C., Knutson, C.D. and Eggett, D.L. (2011), “Design patterns in software maintenance: An experiment replication at brigham young university”, *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research (RESER)*.

Kruskal, W.H. and Wallis, W.A. (1952), “Use of ranks in one-criterion variance analysis”, *J. Am. Stat. Assoc*., **47**, 583-621.

Lucia, A.D., Deufemia, V., Gravino, C. and Risi, M. (2009), “Design pattern recovery through visual language parsing and source code analysis”, *J. Syst. Softw*., **82**(7), 1177-1193.

Mann, H.B. and Whitney, D.R. (1947), *On a Test of Whether One of Two Random Variables is Stochastically Larger Than the Other*, Institute of Mathematical Statistics.

Mayvan, B.B., Rasoolzadegan, A. and Yazdi, Z.G. (2017), “The state of the art on design patterns: A systematic mapping of the literature”, *J. Syst. Softw*., **127**, 93-118.

Nanthaamornphong, A. and Carver, J.C. (2011), “Design patterns in software maintenance: An experiment replication at university of alabama”, *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research (RESER)*.

Niere, J., Schafer, W., Wadsack, J.P., Wendehals, L. and Welsh, J. (2002), “Towards pattern-based design recovery”, *Proceedings of the 24th International Conference on Software Engineering*.

Nija, S. and Olsson, R.A. (2006), “Reverse engineering of design patterns from java source code”, *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*.

Pham, H. (2001), *Software Reliability*, Wiley Encyclopedia of Electrical and Electronics Engineering, John Wiley & Sons, Inc.

Porter, A.A. and Selby, R.W. (1990), “Empirically guided software development using metric-based classification trees”, *Softw. IEEE*, **7**(2), 46-54.

Prechelt, L. and Liesenberg, M. (2011), “Design patterns in software maintenance: An experiment replication at freie university; Berlin”, *Proceedings of the 2nd International Workshop on Replication in Empirical Software Engineering Research (RESER)*.

Prechelt, L., Unger, B., Tichy, W.F., Brossler, P. and Votta, L.G. (2001), “A controlled experiment in maintenance: Comparing design patterns to simpler solutions", *IEEE Trans. Softw. Eng.*, **27**(12), 1134-1144.

Rudzki, J. (2005), "How design patterns affect application performance-a case of a multi-tier J2EE application", *Proceedings of the 4th international conference on Scientific Engineering of Distributed Java Applications*, Luxembourg-Kirchberg, Luxembourg.

Scanniello, G., Gravino, C., Risi, M., Tortora, G. and Dodero, G. (2015), "Documenting design-pattern instances: A family of experiments on source-code comprehensibility", *ACM Trans. Softw. Eng. Meth.*, **24**(3), 1-35.

SciTools (2014), http://www.scitools.com/download/.

Smith, J.M. and Stotts, D. (2003), *SPQR: Flexible Automated Design Pattern Extraction from Source Code*.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis, S.T. (2006), "Design pattern detection using similarity scoring", *IEEE Trans. Softw. Eng.*, **32**(11), 896-909.

Vokac, M. (2004), "Defect frequency and design patterns: An empirical study of industrial code", *IEEE Trans. Softw. Eng.*, **30**(12), 904-917.

Vokac, M., Tichy, W., Sjoberg, D., Arisholm, E. and Aldrin, M. (2004), "A controlled experiment comparing the maintainability of programs designed with and without design patterns-a replication in a real programming environment", *Empir. Softw. Eng.*, **9**(3), 149-195.

Wohlin, C. (2013), "Empirical software engineering research with industry: Top 10 challenges", *Proceedings of the 1st International Workshop on Conducting Empirical Studies in Industry*.

TK