The Effect of GoF Design Patterns on Stability: A Case Study

Apostolos Ampatzoglou, Alexander Chatzigeorgiou, *Member, IEEE*,
Sofia Charalampidou, and Paris Avgeriou, *Senior Member, IEEE*

Abstract—Stability refers to a software system's resistance to the "ripple effect", i.e., propagation of changes. In this paper, we investigate the stability of classes that participate in instances/occurrences of GoF design patterns. We examine whether the stability of such classes is affected by (a) the pattern type, (b) the role that the class plays in the pattern, (c) the number of pattern occurrences in which the class participates, and (d) the application domain. To this end, we conducted a case study on about 65.000 Java open-source classes, where we performed change impact analysis on classes that participate in zero, one (single pattern), or more than one (coupled) pattern occurrences. The results suggest that, the application of design patterns can provide the expected "shielding" of certain pattern-participating classes against changes, depending on their role in the pattern. Moreover, classes that participate in coupled pattern occurrences appear to be the least stable. The results can be used for assessing the benefits and liabilities of the use of patterns and for testing and refactoring prioritization, because less stable classes are expected to require more effort while testing, and urge for refactoring activities that would make them more resistant to change propagation.

Index Terms—Design Tools and Techniques, Object-oriented programming, Metrics/Measurement

1 Introduction

The term pattern, in the field of software engineering, refers to solutions to commonly occurring problems; software patterns have been written for different development phases (e.g., requirements, design, architecture) as well as application domains (e.g., embedded systems, enterprise applications). Some of the most used patterns are object-oriented design patterns [19], architectural patterns [14] and analysis patterns [18]. The first ones ([19]) were introduced in the mid-90s by Gamma, Helm, Johnson, and Vlissides (known as Gang of Four or GoF), where solutions to 23 common object-oriented problems were documented. These patterns are also known as the GoF design patterns.

Since their inception, the GoF design patterns have attracted large attention from the software engineering research community, as revealed by two recent secondary studies, conducted by Ampatzoglou et al. [6] and Zhang and Budgen [53]. In these secondary studies, more than 130 scientific papers related to research on GoF design patterns have been identified. The mapping study of Ampatzoglou et al. [6] revealed that one of the most important research topics on GoF design patterns is their effect on software quality characteristics (the rest research topics were: pattern formalizations, pattern detection, patterns application, and other topics [6]). One particular quality characteristic that both secondary studies emphasize is maintainability. According to both [6] and [53], research on the effect of GoF design patterns on software maintainability is highly active, but still deserves more investigation as many research questions remain unanswered. In this article, we adopt the ISO-9126 definition for maintainability as the *"software quality characteristic concerning the effort needed to make specified modifications to an already implemented system"*. ISO-9126 decomposes maintainability into four characteristics [24]:

- analyzability;
- changeability;
- stability;
- testability.

We focus on software *stability*[1] (and its opposite, instability), which according to ISO 9126 *"characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes"*. Concerning design patterns' effect on stability, until now, most studies have concentrated on pattern *change proneness*, i.e., the number of actual changes to pattern-participating classes, without differentiating between changes from new requirements, changes due to debugging activities, and changes that propagate from changes in other classes. Instability is different to change proneness as follows:

- change proneness is a measurement of all changes that occur to a class (e.g., new requirements, debugging, change propagation, etc.) [25], whereas stability only refers to the last type of change (propagation of other changes).

- *A. Ampatzoglou, S. Charalampidou, and P. Avgeriou are with the Institute of Mathematics and Computer Science, University of Groningen, Groningen, Netherlands. E-mail: {a.ampatzoglou, s.charalampidou}@rug.nl, paris@cs. rug.nl.*
- *A. Chatzigeorgiou is with the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. E-mail: achat@uom.gr.*

Manuscript received 11 Apr. 2014; revised 6 Mar. 2015; accepted 16 Mar. 2015. Date of publication 23 Mar. 2015; date of current version 26 Aug. 2015.
Recommended for acceptance by A. Tanter.
For information on obtaining reprints of this article, please send e-mail to: reprints.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2414917

[1] Instability is used in this paper as the opposite of stability, i.e., the probability of a system to change, due to changes occurring in different parts of the system.

- change proneness is usually calculated from the actual changes that occur in a class (a posteriori analysis), whereas stability can be calculated a priori.

Although instability and change proneness are closely related concepts that can be characterized as two sides of the same coin, there may be cases in which they are not correlated. For example, a class heavily depending on other classes would be highly instable; however, if this class does not actually change, then its change proneness would be low.

Therefore, in this study, instability is defined and measured at class level *as the degree to which a class is subject to change, due to changes in other, related classes, considering the probability of such classes to change as equal to a certain value*. This value is obtained considering a constant value for its internal change probability (due to reasons other than change propagation) as well as its dependencies on other classes. The exact value of the constant internal probability of change for a class does not influence the ranking of classes according to their instability. It will only affect the range of the absolute value of instability (for more details see Section 3.2). Therefore, the co-change of classes that implement the same requirement, which is not occurring because of their dependency, but because of their overlapping class contracts [35] is outside the aforementioned definition of instability. Similarly, the above definition of instability excludes all changes that occur to classes due to changing or additional requirements, and bug-fixing activities (as most definitions of co-change in the literature).

The reason that we focus on stability is that it has been advocated as one of the major benefits of GoF design patterns [19]: design patterns are expected to "shield" some participating classes from ripple effects, i.e., changes propagated to them due to changes occurring in the rest of the system. For example, in the *Façade* design pattern, the class playing the role of *Façade* should prevent the propagation of changes from clients to subsystem and vice-versa. We examine this "shielding" effect from the perspective of the design pattern structure, rather than the change frequency of the surrounding classes of the design pattern. Considering the change frequency of the surrounding classes would invalidate the findings of our study because the stability of the examined patterns classes would be subject to the history of changes in each system (see Section 3.2).

Therefore, we investigate if the claim that GoF patterns support the stability of certain pattern-participating classes holds in practice. In particular, our aim is to investigate how the instability of classes that participate in a GoF design pattern is influenced by four different factors, i.e., pattern type, class role, pattern coupling, and application domain. These factors have also been examined in the literature (see Section 2.3). The reasons why these factors are expected to be influential with respect to class instability are as follows:

- *Type of the GoF design pattern*. The type of the pattern is expected to influence the instability of the classes that participate in it, because the particular structure of each pattern is expected to provide "shielding" to different classes.
- *Role of the class inside the GoF design pattern.* The role that a class plays in the pattern is expected to lead to different levels of instability since different roles have different dependencies to the rest of the system.
- *Intersection of several GoF patterns on a single class* (termed pattern coupling in [34]). In the literature [27], [34], it is suggested that coupled pattern occurrences exhibit a different effect on several source code metrics. Thus, we investigate if a different effect holds for the instability of classes that participate in more than one pattern occurrences.
- *Application domain*. The application domain of a software system is expected to influence the way GoF design patterns are implemented. According to [1] and [48], quality differs significantly among application domains. Therefore, based on these differences, we assume the existence of: (a) differences in levels of instability, (b) differences in the way that both pattern and non-pattern parts of the system are implemented w.r.t. the use of object-oriented characteristics (e.g., encapsulation, inheritance, polymorphism etc.), and (c) potential differences on the amount of design pattern occurrences that would be identified in each domain. All the aforementioned assumptions are expected to differentiate our results per application domain.

To provide empirical evidence on the relation between pattern instability and those four factors, we conducted a multi-case study on about 65,000 classes of 537 open-source software (OSS) projects by performing change impact analysis (see Section 2.1). The reason for performing change impact analysis is to investigate all possible dependencies, through which a change can propagate from one class to another and the probability of such an event, i.e., class instability. Comparing the scope, the goals, and the research method of this study to the previous work on this subject (presented in Section 2), the main contributions of this study (elaborated in Section 2.4) are that:

- It investigates the effect of GoF design patterns on the *estimated* instability of classes, i.e., the probability of a class to change according to changes that occurred in other classes of the system, rather than actual changes occurring in the classes. Studying class instability gives a different perspective on the effect of design patterns on software maintainability, because it relates to the design structure of the pattern, rather than its context (surrounding classes). Additionally, instability, as a design measure and assuming constant internal probabilities of change, can be calculated early (in a pre-deployment phase), while, change proneness is a post-deployment measure. Therefore, instability indicates design spots that might suffer from changeability issues, which can be mitigated before software deployment.
- It is a *large-scale* empirical study. Until now, the largest study on design patterns and change proneness was conducted on three OSS (see Section 2.5).
- It investigates the effect of *coupled* patterns on stability. This is the first study that investigates the aforementioned phenomenon (see Section 2.5).

In the next section, we present related work on change impact analysis, on the effect of GoF design patterns on maintainability and stability, on pattern coupling, as well as an overview of the main contributions of this work with respect to related work. In Section 3, we present the tools used in the case study data collection phase. In Section 4, we present the case study design according to the guidelines provided by Runeson et al. [41]. In Section 5, we report the findings of the case study, which are discussed against each research question in Section 6. Threats to validity are discussed in Section 7. Finally, conclusions and future work are presented in Section 8.

2 Related Work

In this section, we provide an overview of previous research efforts related to the scope of this paper. More specifically, in Section 2.1, we introduce related work in the field of change impact analysis and in Sections 2.2 and 2.3, we discuss research findings on the effect of GoF design patterns on system maintainability and stability. In Section 2.4, we discuss research that has been performed on how design patterns interact and possible implications of this interaction. Finally, in Section 2.5, we summarize key results of studies that have assessed the effect of GoF design patterns on stability, introduced in detail in Section 2.3, and compare those studies against our study.

2.1 Change Impact Analysis

Change impact analysis deals with identifying and quantifying the effects caused by changes in one part of a system on other parts of the same system. According to the first law of software evolution stated by Lehman and Belady [31], namely ‘Continuing Change’, software systems must be continually adapted lest they become obsolete and therefore change impact analysis plays an important role in software development and maintenance. Before the actual application of changes, change impact analysis can be valuable for program comprehension and effort estimation ([13], [22]) whereas, after changes have been applied, it can be used to prioritize test cases and reveal relations among components [40]. The term impact analysis has been used for the first time by Horowitz and Williamson [23] in the mid-80s. Recently, Li et al. [32] have presented a survey of 23 code-based change impact analysis techniques. Change impact analysis techniques can be classified in two broad categories [32]: (a) traceability-based, where the goal is to identify the potential consequences of a change by relating different types of software artifacts (e.g., requirements with source code) and (b) dependency-based analysis, where dependencies among program entities (usually at the code level) are identified and used to assess change impact. The approach and the tool that have been used in this study to assess the stability of pattern- and non-pattern-participating classes belong to the second category since the dependencies among classes in object-oriented systems are used to identify potential change propagation.

2.2 Design Patterns and Maintainability

According to two recent mapping studies on the research state of the art on GoF design patterns, [6] and [53], software maintainability appears to be one of the key quality concerns of researchers that investigate the use of GoF design patterns. More specifically, according to Ampatzoglou et al., 40% (14 out of 35) of the studies on the effect of GoF design patterns on software quality attributes investigate the effect of GoF design patterns on software maintainability [6] whereas, according to Zhang and Budgen, GoF design patterns offer a framework for maintainability and future research efforts should be more focused on maintainability [53].

Two of the most well-known controlled experiments on the effect of GoF design patterns on software maintainability have been performed by Prechelt et al. and Vokac et al., in 2001 and 2004 respectively ([38], [49]). The aim of both studies was to compare the maintainability of systems with and without design patterns. In [38], the patterns considered were Abstract Factory, Observer, Decorator, Composite, and Visitor, while the participants of the experiment were professional software engineers. The results of the experiment suggest that it is usually preferable to apply a design pattern rather than a simpler solution (more details on the examined simpler solutions can be found in papers [38], [49]). In a later replication of the experiment by Vokac et al. [49], who used the same patterns and similar subject groups, the authors increased experimental realism because participants used a real programming environment instead of pen and paper. The results suggest that design patterns are not all beneficial or harmful with respect to maintenance and that the decision of applying a GoF design pattern or a simpler solution is best answered by the designer’s common sense.

Jeanmart et al. [26] performed an experiment with student participants that aimed at evaluating the understandability and the modifiability of Visitor design pattern instances. The experiment used three open-source projects as objects (including canonical and non-canonical representations of the Visitor pattern) and various comprehension and modification tasks as evaluation criteria. Their results suggest that the effort needed for modification tasks is reduced in cases where the canonical representation of the Visitor pattern is used and when the subjects have a good understanding of UML notations.

Ampatzoglou et al. [4] have attempted to provide an objective way of selecting between the application of patterns and alternative design solution, with respect to software maintenance. They proposed an analytical method that uses a set of maintainability predictors [47] and mathematically formalized their metric scores as functions of the number of pattern-participating classes. Applying that method on Bridge and Abstract Factory design patterns, they provided several cut-off points, i.e., number of pattern participating classes thresholds that, when surpassed, make the solutions become more maintainable than the alternative solutions, and vice versa. Both the study of Ampatzoglou et al. [4] and Jeanmart et al. [26] point out the existence of certain conditions, i.e., number of classes and design pattern representation/knowledge of UML notations respectively, that can be used as predictors to decide in which cases the design pattern solution is more maintainable.

Several other research efforts have empirically evaluated the use of design patterns, with respect to software maintainability. Specifically, Khomh and Guéhéneuc per-formed a survey with software engineers with significant experience on GoF design patterns and asked them to evaluate each pattern with respect to eight quality characteristics. The quality characteristic that was related to maintainability was expandability, i.e., the degree to which the design of a system can be extended. The results suggested that 19 out of the 23 GoF design patterns are evaluated as beneficial with respect to maintainability, whereas only four as harmful, i.e., Singleton, Flyweight, Proxy, and Memento [28].

Ampatzoglou and Chatzigeorgiou evaluated the maintainability of State, Strategy, and Bridge design pattern occurrences [3]. The case study was performed on two open-source computer games and provided a comparison between a pattern and a non-pattern version, with respect to complexity, coupling, cohesion, and size metrics. The results suggested that all identified GoF design pattern occurrences improved cohesion, coupling, and complexity of the systems but, as a side effect, increased the size of the systems, both in terms of lines of code and number of classes [3]. In a similar context, Kouskouras and Chatzigeorgiou evaluated the use of an architectural pattern, i.e., namely the Registry pattern adopted from [42], with respect to maintainability, by comparing it to a simple OO implementation, without the use of a pattern, as well as an alternative that combines the pattern with an AOP implementation. The results suggested that using the pattern offers a more maintainable design than the non-pattern version, while the AOP solution was optimal because it retained all beneficial pattern characteristics and limited coupling of the pattern inside the aspect [30].

Finally, Martin et al. investigated the relation between design patterns and the Open Closed Principle [33], through experimentation. More specifically, the authors checked real instances of the State design pattern to examine if the code that should be encapsulated within a particular design is actually using the encapsulation mechanisms of the pattern. The results of the performed experiment suggests that there is only a 20% chance of achieving conformance to the Open Closed Principle if the State design pattern is not used [37]. Assuming that conformance to the Open Closed Principle is the desired way of extending a system, i.e., a way of maintaining the system, the results suggested that there is only a 20% chance for a system without a State design pattern to be maintained in the desired Object-Oriented way, i.e., by adding subclasses, rather than modifying existing code. The result that adding subclasses is the most common way of maintaining a pattern instance during its evolution is supported by the same author, in [36].

2.3 Design Patterns and Stability

In addition to the ISO-9126 definition provided earlier, Yau and Collofello ([50], [51]) define software stability as resistance to propagation of changes (ripple effect) that the software would have when it is modified, which is also known as modular continuity [35]. Although the goal of this study is to evaluate the effect of GoF design patterns on stability, we do not to exclude from this discussion studies related to GoF design patterns and change proneness because: (a) work on patterns and stability is limited and (b) because results on change proneness and results on stability are related in the sense that instability is a subset of change proneness

Some of the first studies on the effect of design patterns on class change proneness were produced by Bieman et al. First, in 2001, the authors conducted an industrial case study that aimed at investigating correlation among code changes, reusability, design patterns, and class size. The results of the study suggest that the number of changes is highly correlated to class size and that classes that play roles in patterns or that are reused through inheritance are more change prone than others [9]. In a replication of the case study, in 2003, the same authors used three professional and two open-source projects, with the same research objectives. The results of the second study do not fully agree with those of the prior case study. The relationships between class size, design patterns participation and change proneness are still valid but appear weaker [10]. In 2009, Gatrell et al. replicated the work of Bieman [9], [10] on proprietary C# applications, by taking the same GoF design patterns into account. The main difference, apart from the programming language, was the metric used for measuring changes. Gatrell used a change-per-class measurement, whereas Bieman used a change-per-operation measurement. However, the results of the replication validated that classes that participate in GoF pattern occurrences are more change prone than classes which do not [20].

Di Penta et al. [15] investigated possible correlations among class change proneness, the role that a class holds in a pattern, and the kind of change that occurs. The design patterns under study are Abstract Factory, Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, and Template Method. They studied three open-source projects. The results of the paper are intuitive for the majority of the roles that a class can play in a design pattern instance. For example, classes playing the *Abstract Factory* role (Abstract Factory pattern) and the *Product* role (Command pattern) change less frequently than the concrete ones. Another example is the Command pattern, where classes playing the role of *Receiver* change more frequently than classes playing the role of *Command*. Furthermore, it is suggested that design activities should take into consideration the roles that a class can play, because interface roles' change proneness can make other parts of the system less robust to changes. Building on [15], Aversano et al. [7] investigated the evolution of GoF design patterns from the perspective of real changes that occur on pattern occurrences, across different releases. More specifically, the authors replicated the research questions of Di Penta et al. [15] and built on them by investigating the changes on pattern client and pattern target classes [7]. The results of the study suggest that pattern occurrences that are used for application purposes are changing more frequently and that different types of changes have a different effect on co-changing classes. Furthermore, Elish has qualitatively investigated the effect of structural GoF design patterns on stability [16] and describe through examples the way changes propagate among GoF design pattern participating classes. The illustrative examples suggest that the studied patterns (i.e., Adapter, Bridge, Composite and Façade) have a positive effect on stability of class diagrams.

TABLE 1
Research State of the Art on the Effect of GoF Design Patterns on Stability and Change Proneness

Study	Patterns Assessed	Stability Metric	Pattern Coupling	Validation Method	Per Pattern	Per Role
[7]	FM, Pr, Si, Ad, Co, De, Ob, Sta, Str, TM, Vi	CF, CFT, LoCCoC	No	Case study (3 OSS)	Yes	No
[9]	Si, FM, Prx, It	ACC ACO	No	Case study (39 versions of 1 commercial)	No	No
[10]	Ad, Bu, FM, It, Prx, Si, Sta, Str, Vi	ACC ACO	No	Case study (3 commercial, 2 OSS)	No	No
[15]	AF, Cmd, Ad, Co, De, FM, Ob, Pr, Si, Sta, Str, TM, Vi	CF CFT	No	Case study (3 OSS)	Yes	Yes
[16]	Ad, Br, Co, Fa	N/A	No	Descriptive Evaluation	Yes	No
[20]	Ad, Bu, FM, It, Prx, Si, Sta, Str, Vi	ACC	No	Case study (1 commercial)	Yes	No
Our study	AF, Pr, Si, Ad, Co, De, Ob, Sta, Str, TM, Vi, Pr	Ins	Yes	Case Study (537 OSS)	Yes	Yes

Design Patterns Abbreviations. *Factory Method (FM), Prototype (Pr), Singleton (Si), Adapter (Ad), Composite (Co), Decorator (De), Observer (Ob), State (Sta), Strategy (Str), Template Method (TM), Visitor (Vi), Proxy (Prx), Iterator (It), Builder (Bu), Abstract Factory (AF), Command (Cmd), Bridge (Br), Façade (Fa).*
Metrics Abbreviations. *Actual Change per Class (ACC), Not Available (N/A), Change Frequency (CF), Change Frequency Type (CFT), Actual Changes per Operation (ACO), Lines of Code Changed in Other Classes (LoCCoC), Instability (Ins).*

Finally, as indirect related work, we have identified several studies on the effect of anti-patterns and code smells on change-proneness, that will be used in our discussion section. Although this presentation is not exhaustive, we provide an overview of the studies that we have used. Firstly, Khomh et al. [29] investigate the impact of code smells on change-proneness by performing a case study on two OSS projects. The results that are relevant to ours are those that are related to specific class roles, such as abstract classes and subclasses. Secondly, Romano et al. [39], investigated the effect of anti-patterns on actual source code changes. More specifically, they indicate that different anti-patterns have different effect on change proneness (also underline the most change prone ones), and that specific anti-patterns lead to specific types of changes.

2.4 Design Patterns Coupling

The term ‘design pattern coupling’ has been introduced in 2001 by McNatt and Bieman [34]. Two or more design pattern occurrences are considered coupled when they share at least one pattern participating class [34].

Concerning the effect of coupled GoF design patterns on quality characteristics, we have been able to identify, only one related study [27]. In this study, Khomh and Guéhéneuc, have identified coupled GoF design pattern instances from five open-source projects and calculated several well-known structural quality metrics, such as Cyclomatic Complexity, Lack of Cohesion of Methods, coupling between objects (CBO), etc. The results revealed quite a different behavior of classes that participate in zero, one, or two and more roles in GoF design pattern occurrences. More specifically, the results suggest that classes that play two and more roles in a design pattern are more complex, more coupled, and less coherent than classes playing one or zero roles in GoF design patterns. The study reported on some demographic results, on the frequency of encountering GoF design pattern coupling. More specifically, the study found that JHotDraw contains only 5.81% of classes that play only one role while 24.45% play two roles in GoF pattern occurrences [27].

2.5 Overview

In this section, we summarize the key characteristics of studies (elaborated in Section 2.3) that have assessed the effect of GoF design patterns on stability or change proneness to discuss the main contribution of our study with respect to related work. The key characteristics of research that deals with patterns and stability or change proneness are summarized in Table 1. In the last line of the table, we present the features of our work.

Comparing the scope, the goals, and the research method of this study to the previous work, this study is:

- the largest-scale empirical study investigating the effect of GoF design patterns occurrences on stability.
- the first large-scale empirical study investigating the effect of GoF design pattern occurrences on any maintainability sub-characteristic, including, but not limited to stability.
- the first empirical study that deals with the effect of coupled GoF design patterns on stability.

3 Used Tools

In this section, we discuss background information needed to understand the tools that we used for GoF pattern detection and for calculating the probability of a class to change.

3.1 Design Pattern Detection

We employed two different pattern detection tools (SSA, by Tsantalis et al.) [45] and (PINOT, by Shi and Olson) [43], both capable of automatically identifying pattern occurrences (of the GoF catalogue) in a given Java project. Both tools can be downloaded from the web.[2]

The tool proposed by Tsantalis et al. [45] identifies pattern occurrences based on a similarity scoring algorithm (SSA), even if the patterns are variations to the standard forms in which they have been originally described. As an example, the approach can identify an occurrence of the Strategy pattern even if there is an intermediate inheritance level between the *Strategy* role (abstract class or interface) and the *Concrete Strategy* subclass role. The underlying detection algorithm is based on a generalization of the link analysis algorithms proposed by Blondel et al. [12].

Pattern inference and recovery tool (PINOT) is a pattern detection approach [43] that can identify occurrences from all structural and behavioral patterns in the GoF catalogue. Detection places emphasis not only on the structural aspects of patterns (derived from inter-class relationships) but also acknowledges the need to consider their behavioral aspect. Once inter-class analysis has been performed to narrow down the search space to particular methods, further static behavioral analysis is applied to each candidate method's body in terms of control flow and data flow.

Consequently, although both tools are performing static analysis, the results of the tools are not expected to be identical, in the sense that:

- SSA investigates methods calls, whereas PINOT does not;
- SSA investigates object creation, whereas PINOT does not;
- PINOT investigates control flow and data flow, whereas SSA does not;
- PINOT identifies pattern occurrences only in their original versions. SSA identifies deviations, as well.

According to an independent study on design pattern detection tools, by Binun and Kniesel [11], the recall rate (i.e., *the percentage of existing patterns that are identified by the tool*) of the SSA tool ranges from 24 to 52% (40.8 percent in average), while the recall rate for PINOT ranges from 13 to 50% (27.2% in average). Additionally, the precision rate (i.e., *the percentage of the identified patterns that is correct*) for the SSA tool ranges from 51 to 80 percent (66.0% in average) and from 9 to 78% (30.6% in average) for the PINOT tool. Yet, the evaluation was performed only on a limited amount of GoF design pattern types (i.e., Composite, Observer, Decorator, Chain of Responsibility, and Proxy) and, therefore, cannot be generalized to all design pattern occurrences that the tools identify. Finally, among the tools discussed in [11], the similarity scoring tool and PINOT are the only ones that can analyze projects regardless of their size.

To increase the degree of confidence on the employed tools and to exclude from the analysis patterns for which the results are not sufficiently accurate, we have manually inspected a number of design pattern occurrences, as presented in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2015.2414917. We have selected to inspect one pattern occurrence of each type, recovered from each application domain (so in total approximately eight pattern occurrences per application domain, i.e., a grand total 119 pattern occurrences). The process was as follows:

a) One pattern occurrence of each type has been randomly picked for every application domain (i.e., approximately eight design pattern occurrences for each pattern type);
b) The first and the second author independently reviewed the retrieved patterns, by providing a "√"or an "X";
c) All pattern occurrences in which the results were contradictive have been discussed by the two authors, this procedure could lead in a change in the evaluation of the reviewer;
d) Every pattern occurrence that included even one "X" was characterized as false positive, accompanied with the reasoning for such a decision.

The results of the inspection led to the following corrective actions:

- Merge State and Strategy occurrences, because they are not easy to differentiate, even manually.
- Merge Façade and Mediator occurrences, because many Mediator occurrences have been manually identified as Façade occurrences.
- Remove Flyweight and Chain of Responsibility occurrences, because the number of false-positives was high.

These actions are expected to reduce the number of false positive pattern occurrences, because pattern types with low precision levels have been either removed from the analysis or merged with similar patterns.

As a final step on the process of selecting and using design pattern detection tools, we faced the decision on whether we should consider the union or the intersection of the two tools. In this study, as the final set of explored patterns, we use the union of the results of the two tools, for the following reasons:

- *Increased number of investigated GoF design pattern types.* The SSA tool identifies occurrences from 11 GoF design patterns types, whereas PINOT from 13 types (nine in common and six unique). Therefore, considering the intersection of the results would lead to a dataset involving a reduced number of GoF design pattern types (i.e., 9 patterns, compared to the 13 pattern types, in the case of union).
- *Increasing recall.* Based on the low number of recall of both tools, we can deduct that both tools "miss" a significant number of pattern occurrences. By joining the results of the tools we aim at including additional true-positive occurences (decrease false negatives), that will increase recall.

However, as a side-effect of this decision, we acknowledge the possible increase of false-positive (decrease of precision). We discuss this as a threat to the validity of this study.

[2] http://java.uom.gr/~nikos/pattern-detection.html
http://www.cs.ucdavis.edu/~shini/research/pinot

3.2 Class Instability

Predicting whether a given software module will change in a future version is an ambitious goal because any actual decision to perform changes to a class is subject to numerous factors. The probability that a certain class will change in the future is affected not only by the likelihood of modifying the class itself but also from possible changes in other classes that might propagate to it. These so-called ripple effects [21] (or change propagation) are the result of dependencies or 'axes of change' (the term 'axis of change' is used as in [46]) among classes through which a change in a class (such as the change in a method signature—i.e., method name, types of parameters and return type) can affect other classes enforcing them to be modified.

The tool[3] that has been employed in this study [46] analyzes the axes of change in which each class is involved and calculates the instability incurred by each axis of change. The accuracy of these probabilistic estimates can be improved by using past data to calculate the probability of change for each class due to modifications to the class itself (internal probability), as well as the percentage of changes that actually have propagated from other classes (propagation factor). As an example, for a class A having a dependency on another class B due to the existence of a reference ($axisB$), the probability of A being changed due to a change in class B is obtained as $\mathrm{P}(A{:}axisB) = \mathrm{P}(A\,|\,B)\bullet\mathrm{P}(B)$. $\mathrm{P}(A\,|\,B)$ is the conditional probability of a change in class A with respect to a change in class B and represents the possibility of propagating a change from one class to the other while P (B) refers to the internal probability of changing class B. A class might be involved in several dependencies and, because even one change will be a reason for editing the code, the probability in which we are interested is the joint probability of all events.

Regarding the internal probability of change, a constant value has been used for all classes. In that sense the results of this study do not reflect the actual distribution of internal changes, because classes are expected to change with different frequencies. Because of this decision, the extracted probabilities reflect only the extent to which a class is subject to future changes because of propagation of changes due to the underlying system design, i.e., due to the dependencies that it has on other classes in terms of inheritance, reference or name dependencies. Consequently, the obtained values are consistent with the notion of stability, which according to the ISO 9126 quality model [24] captures the capability of a software product to avoid unexpected effects from modifications of the software (otherwise, if we had not used a constant value for internal probability of change, it would calculate change proneness and not instability.

As an example, suppose the same instance of a design pattern used in two different circumstances, i.e., (a) in a "design hotspot" where the classes that emit changes to the pattern are *changing very frequently* (see Fig. 1a), (b) in a design spot where the classes that emit changes to the pattern are *not changing very frequently* (see Fig. 1b). In this example, if we had not used a fixed internal probability of change for the classes that communicate with the pattern instance (classes in the pattern 'neighborhood'), the same design pattern instance would have been characterized as stable in the case of Fig. 1b, and as instable in the case of Fig. 1a. However, the structure and connectivity of the pattern instance to the rest of the system is the same. Therefore, we believe that considering the actual frequency of changes in our study would invalidate the investigation of pattern stability from a structural perspective, since in such a case, stability would also be affected by the design spot, in which the pattern is used.

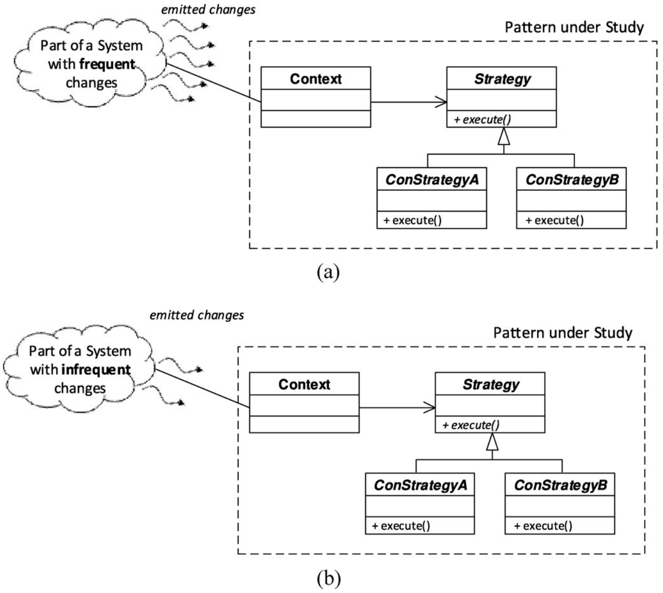


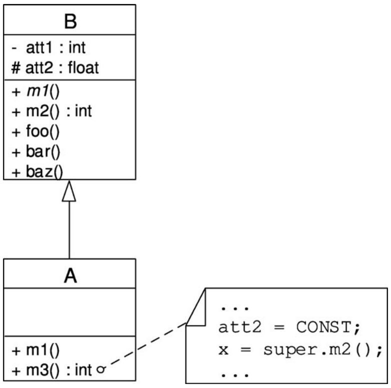
Fig. 1. (a) Design pattern instance placed in a frequently-changing spot of the design, (b) design pattern instance placed in an infrequently-changing spot of the design.

Concerning the propagation factor of changes among dependent classes, we preferred not to use a constant value, because the change propagation factor should ideally reflect as closely as possible the underlying design and the effect of design patterns. For this reason, we used a ripple effect measure (REM), which attempts to quantify the probability of a change occurring in class B to be propagated to a dependent class A, as discussed in Section 3.3.

3.3 Ripple Effects Measurement

In general, there are two types of axis of change, along which a change can propagate: i.e., generalization and association relationships. To quantify the propagation factor, we attempt to estimate the percentage of the accessible interface of a class, which might emit changes to a dependent class. In case of an association, this estimate can be obtained as the ratio of distinct method calls from A to B, over the number of public methods in class B. In case of generalization, there are three possible reasons for change propagation: (a) super method invocation (use of super), (b) access of protected fields, and (c) override or implementation of abstract methods of the superclass. All these sources should be normalized over the total number of accessible members in the

[3] Old tool: http://java.uom.gr/nikos/probabilistic-evaluation.html new tool: http://iwi.eldoc.ub.rug.nl/root/2014/ClassInstability/
We provide a link to both the old and the new version of the tool: (a) as an acknowledgement to the tool that we used a starting point for reuse and (b) as a reference for readers that might be interested in comparing the results of the two tools.



```
...
att2 = CONST;
x = super.m2();
...
```

Fig. 2. Sample code for illustrating the calculation of REM.

superclass. According to these observations, REM can be calculated as follows:

$$\text{REM} = \frac{NDMC + NOP + NPrA}{NOM + NA}. \quad (1)$$

NDMC: Number of distinct method calls from class *A* to class B (super class method invocations for the case of generalization)

NOP: Number of polymorphic methods in class *B* (valid only for generalization)

NPrA: Number of protected attributes in class *B* (valid only for generalization)

NOM: Number of methods in class *B*

NA: Number of attributes in class *B* (valid only for generalization)

The aforementioned measure has been incorporated in the employed tool, by re-writing the functionality related to the calculation of class probability of change.[4] Specifically, for every dependency that is identified by the tool, we calculate REM and set it as the propagation factor between the depended classes. For example, consider the sample design of Fig. 2, where class *A* extends class *B*.

In the general case, class *A* can change if a change occurs in *B* in the following cases:

- if it overrides a method, and the signature (method name, types of parameters and return type) of this method changes.
- if it calls a method of the superclass and the signature of the superclass method changes.
- if it uses a protected attribute, and this attribute changes name.

On the other hand, the following changes in *B* do not propagate in *A*:

- changes in the body of any function.
- changes in the signature of methods that *A* does not call or override.
- changes in private attributes.

In the above example there are three types of change which might propagate from superclass *B* to subclass *A*: (a) change in `att2`, (b) change in `m1()`, and (c) change in `m2()`. Changing the type, the name, or deleting `att2` will lead to a compile error wherever `att2` is accessed. Changing the signature of `m2()` would lead to a compile error in the corresponding invocation. Finally, changing the signature of `m1()` would lead to a compile error in the place where `m1()` is overridden, because `m1()` is declared abstract in the superclass. Thus, the estimate for the propagation factor can be calculated as:

$$\text{REM} = \frac{RFC + NOP + NPrA}{NOM + NA} = \frac{1+1+1}{5+2} = 0.42.$$

Although REM is not a proper probability value, it captures the degree of interdependence between two classes, and thus provides a relative estimate for the propagation factor in each case.

3.4 Discussion on the REM

In this section we discuss some key strengths and limitations of the previously defined measurement (REM). Specifically, we discuss: (a) differences of REM to existing metrics, (b) ability of REM to differentiate between pattern and non-pattern versions of the system, and (c) REM as a change proneness measure.

One of the first tasks that we have performed while designing this study was the identification of an existing software metric that would be adequate for quantifying the instability of a class. Intuitively, instability can be associated to coupling metrics, i.e., metrics that quantify the extent to which classes are interconnected.[4] After going through the definition of the most popular coupling metrics we identified that all of them suffer from at least one of the following limitations for measuring stability:

- they quantify only the number of dependencies between classes, and not the intensity of the coupling - e.g. coupling between objects, afferent coupling (AfC), efferent coupling (EfC), etc.
- they quantify the intensity of a class dependency, but use a count of how many times a method is called inside another method as a measure - e.g., message passing coupling (MPC). This is not desirable because even one method call can lead to a change propagation.
- they use attribute-related coupling, only by counting the number of fields that are declared through an aggregation relationship - e.g., measure of aggregation (MOA). This is not desirable, since for classes in the same hierarchy, changes can propagate also through `protected` fields.

Therefore, none of the already existing metrics was able of quantifying all the identified ways that a class could emit changes to another (see Section 3.3). Furthermore, to validate our aforementioned qualitative evaluation, we performed a small scale quantitative evaluation of REM (on two open-source projects). The results suggested that REM is more highly correlated to change propagation than any of the aforementioned existing coupling metrics.

In addition to that, when comparing design-patterned versus non-design-patterned spots of the system design, the differences in the nature of the expected changes to them

[4] Other structural quality attributes like complexity or cohesion have not been considered, because they quantify internal characteristics of a class (e.g., similarity of methods/attributes or number of decision statements in a method), and not its interconnection to other classes.

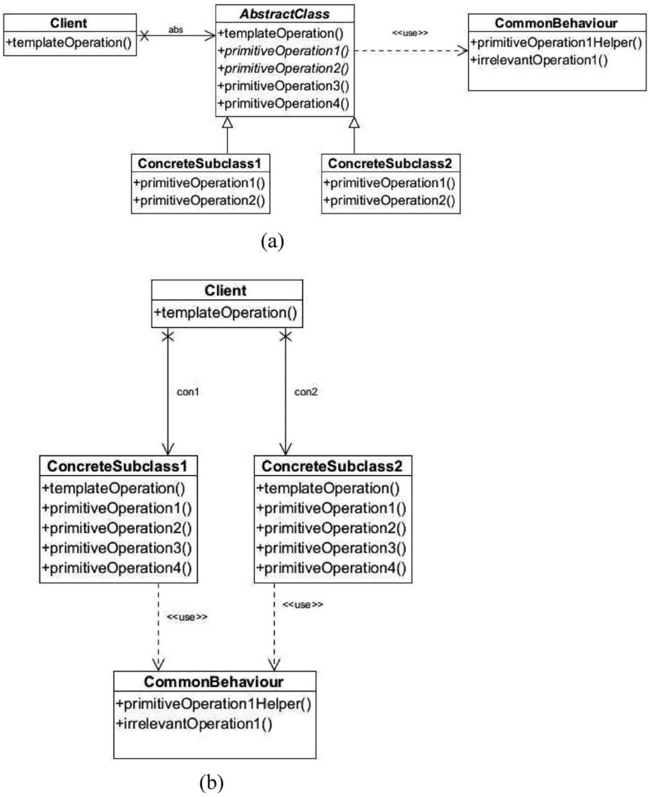


Fig. 3. (a) Illustrative template method instance, (b) illustrative template method alternative.

should be identified. The basic issue that discriminates between these two spots is that the goal of many design patterns is to make the design resilient against certain kinds of change that are expected to be frequent. Those kinds of change will never propagate, yet many (if not most) of the changes will be of such type. According to [36], the most common way of maintaining a design pattern instance is by adding subclasses in the class hierarchies of the patterns (in cases when they are applicable). Also, this way of maintaining a system is acknowledged as the desired way, based on the Open-Closed Principle [33]. Therefore, such a discrimination should not be neglected by the way that REM is calculated. Based on the definition of REM, the difference between propagating changes over inheritance versus propagating changes over associations is captured, leading to the desired differentiation.

For example, consider the following two cases (the first with a template method instance, see Fig. 3a, and the second an equivalent non-pattern solution, see Fig. 3b). We note that in order for the examples to be realistic we do not consider that the pattern instance is completely self-sufficient, but some information is encapsulated in other classes as well. For the design in Fig. 3a (Template Method pattern) the calculation of REM is as follows:

- `Client`: 0.2, it depends on one out of five methods of `AbstractClass` (`templateOperation`)
- `AbstractClass`: 0.5, it depends on 1 out of 2 methods of `CommonBehaviour` (primitiveOperation1Helper)
- `ConcreteSubclasses`: 0.4, they depend on 2 out of 5 methods of `AbstractClass` (i.e., `primitiveOperation1` and `primitiveOperation2`)

Whereas, for the design in Fig. 3b (Template Method alternative) the calculation of REM is as follows:

- `Client`: $0.2 + 0.2 - 0.2^*0.2 = 0.36$, it depends on 1 out of 5 methods of both `ConcreteSubclasses` (`templateOperation`)
- `ConcreteSubclasses`: 0.5 they depend on 1 out of 2 methods of `CommonBehaviour` (`primitiveOperation1Helper`).

During evolution, a typical expected change in the design is the addition of `ConcreteSubclasses`, which in the non-pattern version appear to have larger REM values than the pattern version. Also, as `ConcreteSubclasses` are added, the REM of the `Client` increases as well. Thus, the proposed metric discriminates between design-patterned spots from non-patterned spots (based on the dependencies they are involved into), even with regard to the particular changes for which the patterns have been designed.

Finally, we acknowledge that the proposed metric is incapable of quantifying class change proneness, because it is purely syntactic and does not take into account class contracts, changes due to modified requirements, and bug fixing activities. To propose such a metric one would have to consider the semantics of functions, but even then it would be impossible to obtain an accurate estimate of change propagation probability, without analyzing the history of actual changes. However, addressing this need would invalidate the findings of our study as the stability of the examined patterns would be subject to the history of changes of the surrounding classes and not only to their dependencies.

4 Case Study Design

The objective of this case study is to investigate the stability of classes that participate in GoF design pattern occurrences. To achieve this goal, we compare the stability of classes participating in zero, one, or more design pattern occurrences, through a multi-case study. The case study has been designed and reported according to the template suggested by Runeson et al. [41]. The next sub-sections contain the four parts of the design, i.e., Objectives and Research Questions, Case Selection and Units of Analysis, Data Collection and Pre-Processing, and Data Analysis.

4.1 Objectives and Research Questions

The goal of the study is described using the Goal-Question-Metrics (GQM) approach [8]:

"*Analyze open source projects for the purpose of evaluating design pattern participating classes with respect to their stability, i.e., their probability to change due to changes occurring on classes directly or indirectly associated with them from the point of view of software developers, in the context of open-source Java software projects*".

According to our goal and the four factors that we explained in the Introduction section (pattern type, class role, pattern coupling, and application domain), we have derived three research questions that will guide the case study design and the reporting of the results:

RQ$_1$: Is the number of pattern occurrences, in which a class participates, correlated to the stability of the class?

RQ$_{1.1}$: Is there a difference in the stability of classes that participate and classes that do not participate in design patterns?

RQ$_{1.2}$: Is there a difference in the stability of classes that participate in zero, one, or more than one design pattern occurrences?

RQ$_{1.3}$: Is there a difference in the stability of classes that participate in zero, one, or more than one design pattern occurrences, across different application domains?

RQ$_2$: Is the type of the pattern, in which a class participates, correlated to the stability of the class?

RQ$_{2.1}$: Is the type of the single pattern, in which a class participates, correlated to the stability of the class?

RQ$_{2.2}$: Is the type of the coupled patterns, in which a class participates, correlated to the stability of the class?

RQ$_3$: Is the role that a class plays in a single pattern, correlated to the stability of the class?

Although RQ$_{1.1}$ could be answered through the investigation of RQ$_{1.2}$, we have preferred to state it as a separate research question because all previous studies have only answered RQ$_{1.1}$ and we can directly compare our results with those of previous studies. The metrics used to answer these research questions are discussed in Section 4.3.

4.2 Case Selection and Units of Analysis

According to Yin, for every case study, researchers must determine the context, the cases, and the units of analysis [52]. In this study, the context is open-source software and the cases/units of analysis are open source system classes. We note that this case study is holistic, because for each case, one unit of analysis is extracted.

To gather as many cases as possible, we have decided to use a software engineering repository that documents design pattern occurrences (using the design pattern detection tools mentioned in Section 3.1); the repository, named *percerons.com*, was created by one of the authors [5]. The aforementioned repository was initially created in 2009 as a catalogue of design pattern occurrences and a search engine to provide access to them. In the current version, the repository shares data on 537 OSS projects. In order to guarantee, as far as possible, the data validity, we performed the pattern occurrence validation process and the corrective actions (see Section 3.1), before data extraction.

On the completion of this process we obtained 64,941 units of analysis. From these classes, 10,413 participated in exactly one design pattern occurrence 2,716 participated in more than one design pattern occurrences and 51,812 did not participate in any design pattern occurrence.

4.3 Data Collection and Pre-Processing

The dataset that has been used in this study consists of 64,941 rows, one row for each class of the considered systems. For every class, we recorded nine variables:

[V1] *Software system*: The name of the OSS project from where we extracted the data.

[V2] *Application domain*: The application domain of the software system (as defined in Sourceforge).

TABLE 2
Dataset Description

Pattern Participation	Class Count	%
No Pattern	51,812	79.8
Single Pattern	10,413	16,0
Coupled Pattern	2,716	4,2
Total	64,941	100.0

[V3] *Class name*: The name of the class under study.

[V4] *Type of pattern*: The name/names of the GoF design patterns that a class participates in (e.g., for single patterns: State, for coupled patterns: Strategy and Visitor)

[V5] *Names of roles*: The name/names of the role/roles that a class [V3] plays in GoF design pattern/patterns mentioned in [V4] (e.g., for single patterns: Concrete Product, for coupled patterns: Concrete Strategy and Concrete Element)

[V6] *Instability*: The probability of class [V3] to change due to change propagation, as provided by the tool described in Section 3.2. For the rest of the paper, this value will be referred to as instability.

[V7] *Count of pattern occurrences*: A numeric representation of [V4], i.e., number of pattern occurrences.

[V8] *Pattern participation*: A Boolean representation of [V4]. It is set to true for classes that participate in at least one pattern and to false for classes that participate in no pattern occurrences.

[V9] *Coupled pattern participant*: An additional representation of variable [V4]. It has three values: no pattern, single pattern, or coupled pattern. We use this variable to distinguish coupled from single patterns.

[V8] and [V9] are variables that are derived from [V7]. These variables have been created to ease the analysis of the dataset because Boolean and ordinal representations offer additional means for statistical analysis. In Section 4.4, we map the aforementioned variables to the research questions where they were used.

The produced dataset can be categorized with respect to GoF design patterns participation as shown in Table 2. The results of Table 2 suggest that approximately 20% of system classes participate in at least one GoF design pattern occurrence. This is in accordance to the outcome of a previous study, by Khomh and Guéhéneuc, who suggested that the number of classes that participate in at least one design pattern is between 4-30% [27]. However, by comparing the percentage of classes playing exactly one (in [27] it is reported to be between 4-30 percent) or more than one role (in [27] it is reported to be between 12-26 percent), we can observe a differentiation, that is probably due to the used pattern detection tools, and due to the fact that in [27] the authors selected to investigate six selected well-known OSS projects, whereas in our study we investigated 537 projects, including both reputed and less-known ones.

In Table 3, we present the application domains of the OSS projects, the number of projects classes, and the number of pattern participating classes in each domain.

The application domains have been recorded during data extraction according to the categorization in

TABLE 3
GoF Design Patterns Repository Demographics

Application Domain	Project Count	Class Count	Pattern Participation (%)
Audio and Video	50	4,301	31,3%
Business and Enterprise	50	5,768	28,8%
Communications	59	4,305	28,6%
Development	119	16,273	13,3%
Games	135	12,970	20,9%
Graphics	53	7,715	25,9%
Home and Education	31	3,177	21,9%
Science and Engineering	40	10,432	12,4%
Total	537	64.941	20,2%

TABLE 4
Single GoF Design Patterns Participants

GoF Design Pattern	Class Count
Adapter*	2,894
Template Method	2,388
Singleton	2,384
Façade-Mediator	1,280
Strategy-State	550
Factory Method	476
Prototype	161
Proxy	89
Abstract Factory	73
Decorator	57
Composite	32
Observer	24
Visitor	5
Total	10,413

We note that both tools report occurrences of Object Adapter

Sourceforge, which is the OSS repository that was used for mining open-source software projects in 2009. We believe that mapping OSS projects with the application domains that their owners have selected to classify them is a safe option for this kind of characterization. The data in Table 3 suggest that around 20% of classes in OSS systems participate in GoF patterns, a result which is in accordance with [27, in which Khomh and Guéhéneuc suggest that approximately 30% of JHotDraw (i.e., one of the most frequent and pattern intensive OSS examples for GoF pattern research) classes participate in patterns. In Fig. 4, we present the distribution of patterns across domains, across investigated pattern types.

In Table 4, we present the number of classes that participate in single design pattern occurrences that have been retrieved during our analysis, whereas in Table 5, we present the number of classes that participate in the most common combinations of GoF design pattern occurrences that occur in coupled patterns. The sum of coupled pattern occurrences does not match the one presented in Table 2, because there are 180 more coupled pattern types (with lower occurrence frequency) that are omitted from Table 5.

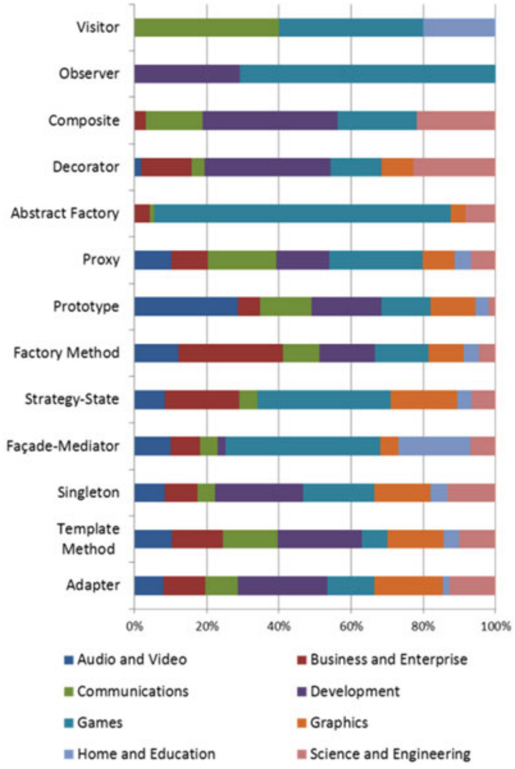


Fig. 4. Distribution of pattern occurrences across application domains.

In Table 6, we present the number of occurrences for each pattern role, for single pattern occurrences. Similar to Table 5, single pattern roles with a low number of occurrences are omitted. Finally, in Table 7, we present a synthesized representation of roles across patterns. The rationale behind this data synthesis is based on the existence of similar roles that are found in different patterns. Namely, every role can be classified as:

- client,
- abstract class/interface,
- concrete subclass,
- aggregate/container in a "whole-part" relationship, or the dependent class in a "simple association" (for simplicity, further referenced as aggregate)
- component in a "whole-part" relationship or the independent class in a "simple association" (for simplicity, further referenced as component), and
- other type, with either more than one associations, e.g., both inheritance and aggregation (composite/decorator) or no association (singleton).

TABLE 5
Coupled GoF Design Patterns Participants

Coupled GoF Design Patterns	Class Count
2 Façade-Mediator	314
Adapter, Template Method	250
Singleton, Adapter	133
Adapter, Strategy-State	106
Adapter, Façade-Mediator	84
Template Method, Prototype	78
2 Adapters	67
Singleton, Façade-Mediator	61
2 Template Methods	52
Adapter, Factory Method	44
Total	1,189

* *180 more combinations of patterns with occurrences that involve less than 44 participating classes*

TABLE 6
Single Patterns Roles

GoF Design Pattern	Class Count
Singleton	2,384
Concrete Class [Template Method]	1,883
Adapter	1,451
Adaptee	1,443
Hidden Type-Colleague	840
Abstract Class [Template Method]	505
Mediator-Façade	440
Concrete Factory [Factory Method]	363
Concrete Strategy-State	336
Context [Strategy-State]	119
Creator [Factory Method]	113
Concrete Prototype	102
Strategy-State	95
Concrete Factory [Abstract Factory]	66
Proxy	40
Real Subject [Proxy]	40
Client [Prototype]	37
Concrete Decorator	32
Leaf [Composite]	22
Prototype	22
Concrete Observer	16
Component [Decorator]	13
Decorator	11
Total	10,373

* *20 more roles with less than 10 occurrences*

In Table 7, we observe that the number of clients is low, namely 156. This fact is due to a limitation of pattern detection tools to identify classes that play this role in all types of patterns (only for State-Strategy, Prototype, Observer).

4.4 Data Analysis

To explore our dataset for answering the research questions described in Section 4.1, we applied descriptive statistics and hypothesis testing, as shown in Table 8. From Table 8, we observe that all variables are used in the investigation of at least one research question, except for variables [V1] and [V3]. Variables [V1] and [V3] are used for tracking/verification purposes, i.e., to identify systems and classes that are involved in design pattern occurrences so as to manually assess the validity of the pattern detection tools.

Spearman correlation is used as measurement of correlation between numerical and ordinal variables. Values of Spearman Correlation Coefficient that are near unity (1.0) suggest that the values are highly correlated. In addition, although scatter plots are normally used for correlation analysis (RQ$_1$), a heat map has been used because both variables are ordinal. The 95% CI Bars present the mean value of a numerical variable and its 95% confidence interval. Error bars can also be used to visually compare the mean values of two or more groups and get preliminary indications on the existence of statistically significant differences.

TABLE 7
Roles across GoF Design Patterns

GoF Design Pattern	Class Count
Abstract Class / Interface	779
Concrete Subclass	2,864
Aggregate / Container	1,891
Component	2,283
Client	156
Other	2,440
Total	10,413

TABLE 8
Data Analysis per Research Question

RQs	Variables	Data Analysis
RQ$_1$	[V6] numerical [V7] ordinal	Spearman Correlation Line Chart and Heat Map
RQ$_{1.1}$	[V6] numerical [V8] binary	95% Confidence Interval Error Bars Independent Sample t-test
RQ$_{1.2}$	[V6] numerical [V9] ordinal	95% Confidence Interval Error Bars Independent Sample t-test Hochberg's GT2 Post Hoc test
RQ$_{1.3}$	[V2] categorical [V6] numerical [V9] ordinal	95% Confidence Interval Error Bars Independent Sample t-test Hochberg's GT2 Post Hoc test
RQ$_2$	[V4] categorical [V6] numerical	95% Confidence Interval Error Bars Analysis of Variance (ANOVA) Hochberg's GT2 Post Hoc test
RQ$_3$	[V4] categorical [V6] numerical	95% Confidence Interval Error Bars Analysis of Variance (ANOVA) Hochberg's GT2 Post Hoc test
RQ$_4$	[V5] categorical [V6] numerical	95% Confidence Interval Error Bars Analysis of Variance (ANOVA)

To investigate the existence of statistically significant differences in the mean values of numerical variables among groups, we have used two kinds of tests: (a) independent sample t-tests for comparing two groups of variables and (b) analysis of variance (ANOVA) for comparing the mean values of more than two groups. In the case of ANOVA, the test only reveals the existence of some differences among groups but does not point out the groups that differ. To have a more precise understanding of the relationships among certain groups, we performed Hochberg's GT2 Post Hoc tests, which is for samples whose internal groups are not equal in terms of population [17] and [44].

5 Results

In this section, we present the results of the case study, organized per research question. All results and comparison to related work are discussed in Section 6.

5.1 RQ$_1$: Number of Pattern Occurrences

To investigate if the number of design pattern occurrences in which a class participates is correlated to the instability of a class, we performed a Spearman correlation test. The two variables appear to be almost not correlated at all (Correlation Coefficient: 0.18, sig: 0.00). Despite the weak correlation, the results suggest that *as the number of design patterns in which a class participates increases, the less stable the class becomes* (see the heat map and embedded line chart in Fig. 5). In addition to that, one can observe that the instability of classes that participate in exactly one pattern occurrence is slightly smaller

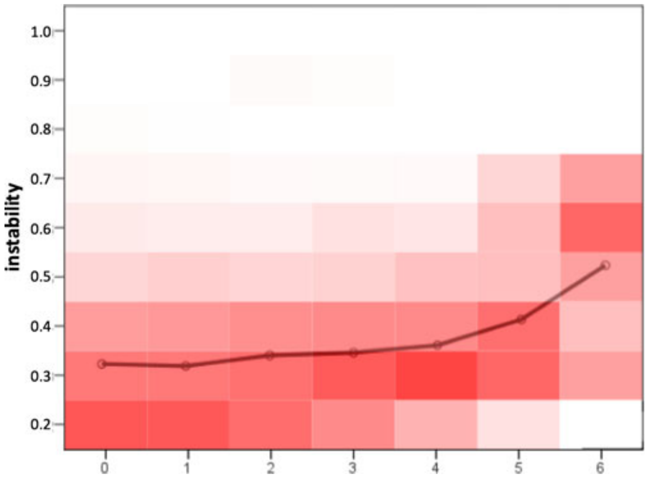


Fig. 5. Pattern occurrences vs. instability (Heat Map, plus line chart representing average instability values).

(approx. 3 percent) than the average instability of classes that do not participate in any pattern.

> *The instability of classes is weakly correlated to the number of patterns that a class participates in (the correlation is statistically significant). However, as the number of pattern occurrences increases, the instability increases as well.*

Next, to further investigate the relationships between the count of pattern occurrences in which a class participates and its instability, we explore three null hypotheses:

$H_{0(a)}$ The mean instability of classes that participate in at least one design pattern occurrence equals the mean instability of classes that do not participate in any design pattern occurrence.

$H_{0(b)}$ The mean instability of classes that participate in zero, one and more pattern occurrences is equal.

$H_{0(c)}$ The mean instability of classes that participate in zero, one and more GoF design pattern occurrences is equal, regardless of the application domain.

5.1.1 $RQ_{1.1}$: Participation in At Least One, or Zero Occurrences

In Fig. 6, we observe that classes that participate in at least one design pattern occurrence are slightly less stable than classes that do not participate in any GoF design pattern occurrence (mean$_{\text{at least one design pattern}}$: 0.341 and mean$_{\text{zero patterns}}$: 0.339). This result is mainly caused by the fact that in the *"at least one pattern participant"* category, we synthesize the average instability of classes that participate in at least one design pattern occurrences. Therefore, the results are not contradictory to those of Fig. 5; however, although the results in Fig. 5 indicate that classes that participate in exactly one design pattern are slightly more stable than classes that do not participate in any pattern, the instability increases significantly when the number of patterns in which a class participates is higher.

The results of the corresponding independent sample t-test (sig: 0.41, lower confidence interval of difference: –0.004 and higher confidence interval of difference: 0.001), cannot lead to the rejection of the aforementioned hypothesis $H_{0(a)}$.

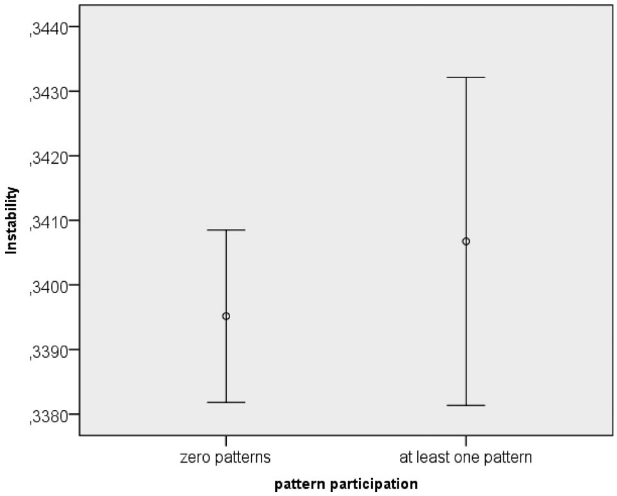


Fig. 6. Pattern participation vs. instability (error bars).

Thus, the mean instability of classes that participate in at least one design pattern occurrence does not differ with statistical significance from the mean instability of classes that do not participate in any design pattern occurrence.

> *The instability of classes that do not participate in design pattern occurrences is not statistically significantly different, from the instability of classes that participate in design pattern occurrences*

5.1.2 $RQ_{1.2}$: Participation in Zero, One, or More Occurrences

The results illustrated in Fig. 7 suggest that although the mean instability of classes that participate in exactly one GoF design pattern is slightly lower than the mean instability of classes that do not participate in any design pattern occurrence, there is an overlap in the 95% intervals of their mean values. This fact indicates that the difference in the corresponding mean values might not be statistically significant.

To more thoroughly examine hypothesis $H_{0(b)}$, i.e., whether the mean instability of classes that participate in zero, one, and more design pattern occurrences is equal, we performed an analysis of variance (ANOVA).The results of

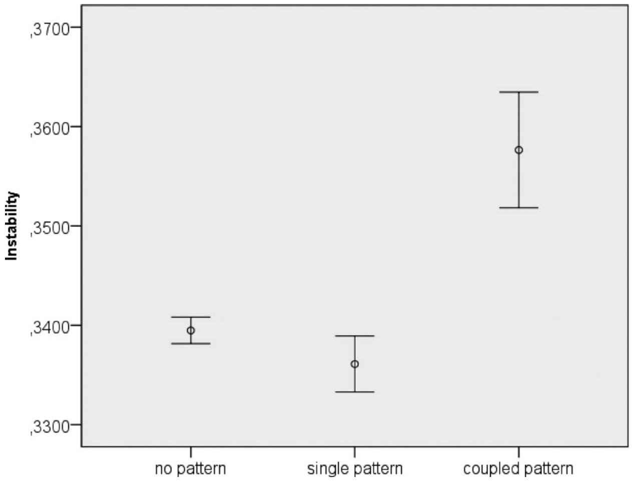


Fig. 7. Pattern coupling vs. instability (error bars).

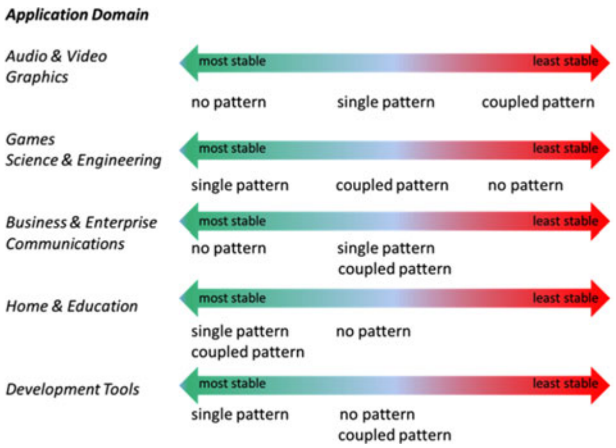


Fig. 8. Differences among various application domains.

ANOVA suggest that the three groups, i.e., *no pattern, single pattern, and coupled pattern present statistically significant differences* (F: 20.19 and sig: 0.00), but the Post-Hoc test of Hochberg's GT2 suggests that *the differences are not significant between all groups, but only between no pattern and coupled pattern* (sig: 0.03, lower confidence interval of difference: –0.02 and higher confidence interval of difference: –0.01), and *single pattern and coupled pattern* (sig: 0.03, lower confidence interval of difference: –0.03 and higher confidence interval of difference: –0.01).

> *The instability of classes that participate in more than one design pattern occurrence is statistically significantly higher, from the instability of classes that participate in exactly one design pattern occurrence and of classes that do not participate in any design pattern occurrences*

5.1.3 *RQ$_{1.3}$: Application Domains*

Finally, concerning the mean instability of classes that participate in zero, one, and more GoF design pattern occurrences, across different application domains, the results reveal five different clusters of application domains, that exhibit different effect of GoF design pattern participation to class instability as follows (for 95% CI Error Bars see Fig. 13 in Appendix B, available in the online supplemental material). The main findings are summarized in Fig. 8.

From Fig. 8, it becomes clear that the reported results on RQ1.2 (i.e., the relationship of the participation of a class in zero, one or more pattern occurrences and class instability) vary, depending on the examined application domain. For example, in Graphics, Audio and Video applications, the most stable classes do not participate in GoF design pattern occurrences, whereas the most instable ones, participate in more than one design pattern occurrence. On the other hand, in Games, Scientific and Engineering applications, the most stable classes participate in exactly one design pattern occurrence, whereas the most instable ones, do not participate in any pattern.

> *The instability of classes that participate in zero, one or more design pattern occurrences is statistically significantly different across different application domains*

5.2 RQ$_2$: Design Pattern Type

In this section, we present the results concerning RQ$_2$, i.e., the effect of the design pattern type on class stability. More specifically, in Section 5.2.1, we present results on single design pattern occurrences whereas in Section 5.2.2 on coupled design patterns.

5.2.1 *RQ$_{2.1}$: Single Design Pattern*

To investigate if the effect of patterns on class instability is equal across all studied GoF design patterns, we have set and explored the following null hypothesis:

H$_{0(d)}$ The instability of a class that participates in a GoF design pattern occurrence is equal, across the studied GoF design pattern types.

Investigating the hypothesis, through an error bar on the 95% confidence interval (CI) of instability (see Fig. 9) suggests that there are differences in the mean values of instability across different types of GoF design patterns. The confidence interval for each design pattern is obtained considering the values of instability of individual classes

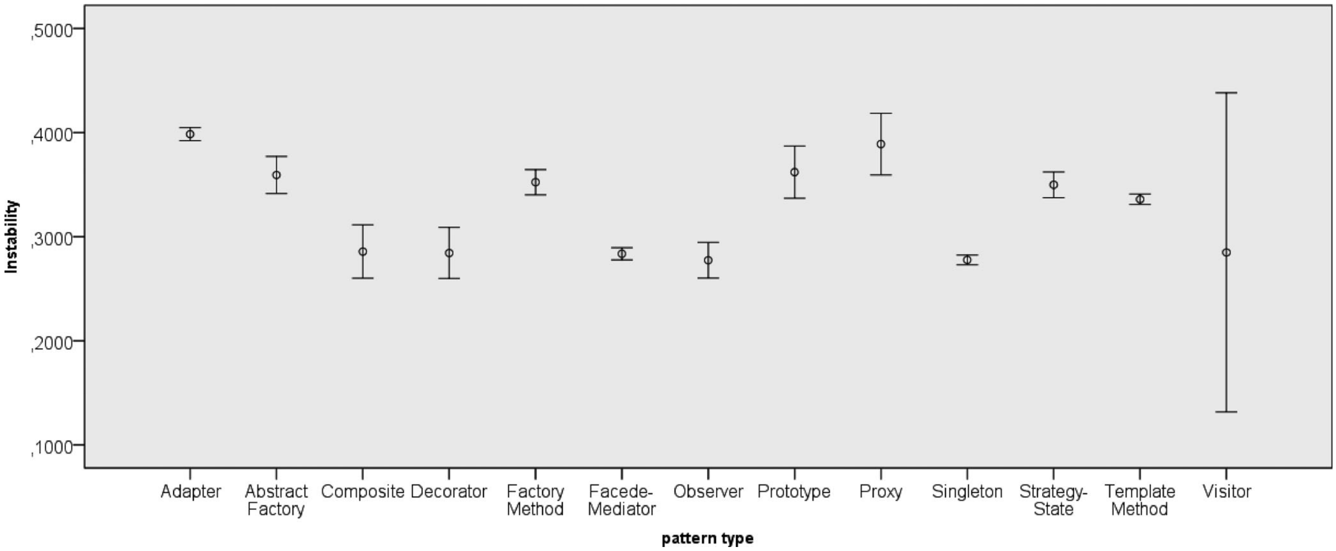


Fig. 9. Single pattern type vs. instability (error bars).

TABLE 9
Post-Hoc Results (Single Pattern Type vs. Instability)

	Adapter	Abstract Factory	Composite	Decorator	Façade-Mediator	Factory Method	Observer	Prototype	Proxy	Singleton	Strategy-State
Abstract Factory	0,848										
Composite	0,001**	0,773									
Decorator	0,000**	0,224	1,000								
Façade -Mediator	0,000**	0,000**	1,000	1,000							
Factory Method	0,000**	1,000	0,647	0,052	0,000**						
Observer	0,002**	0,735	1,000	1,000	1,000	0,653					
Prototype	0,103	1,000	0,411	0,030*	0,000**	1,000	0,425				
Proxy	1,000	1,000	0,025*	0,001**	0,000**	0,850	0,034*	1,000			
Singleton	0,000**	0,000**	1,000	1,000	1,000	0,000**	1,000	0,000**	0,000**		
Strategy – State	0,000**	1,000	0,779	0,091	0,000**	1,000	0,771	1,000	0,617	0,000**	
Template Method	0,000**	1,000	0,996	0,497	0,000**	0,873	0,991	0,907	0,024*	0,000**	0,998

participating in each pattern. In most of the cases, there are limited or no overlaps of the 95% CI bars. The mean instability of a pattern is calculated as the average instability of classes that participate in it. Although in Fig. 9, the mean instability for each pattern is depicted (the dot in each line), the main emphasis of the diagram is on the 95percent CI bars. Focusing on the mean values poses a threat to the validity of the results because the number of pattern participating classes in each pattern occurrence is related to the type of the pattern (e.g., Singleton in its most common form involves only one class, whereas other patterns like State or Strategy can involve a large number of concrete subclasses), as we discuss in Section 7.

The ANOVA test indicates that the studied groups, i.e., GoF design pattern types, present statistically significant differences in terms of their mean values of instability (F: 108.56 and sig: 0.00). The results of the Hochberg's GT2 Post-Hoc tests are presented in Table 9. Patterns that are not included in Table 9, do not differ from any other pattern, possibly because of the small number of occurrences in the dataset. In each cell of Table 9, we present the significance value of the Hochberg's GT2 test (i.e., the extent to which the difference in the mean instability of one pattern [row] from the mean instability of another pattern [column] is statistically significant). A difference between two pattern types is statistically significant if the corresponding value is less than 0.01(annotated with a double asterisk in Table 9, whereas single asterisks denote a statistical significance at a 0.05 level).

> *The instability of classes that participate in exactly one design pattern occurrence is statistically significantly different across different types of GoF design patterns.*

5.2.2 $RQ_{2.2}$: Coupled Design Patterns

The results of this section concern only the coupled pattern occurrences of Table 5, because of the large numbers of possible combinations of GoF design pattern occurrences. To investigate $RQ_{2.2}$, we have stated and investigated the following null hypothesis:

$\mathbf{H_{0(e)}}$ The instability of a class that participates in more than one GoF design pattern occurrences is equal, across the studied combinations of GoF design pattern occurrences.

The 95% Confidence Interval Error Bars (see Fig. 10) suggest that classes that participate in most types of coupled

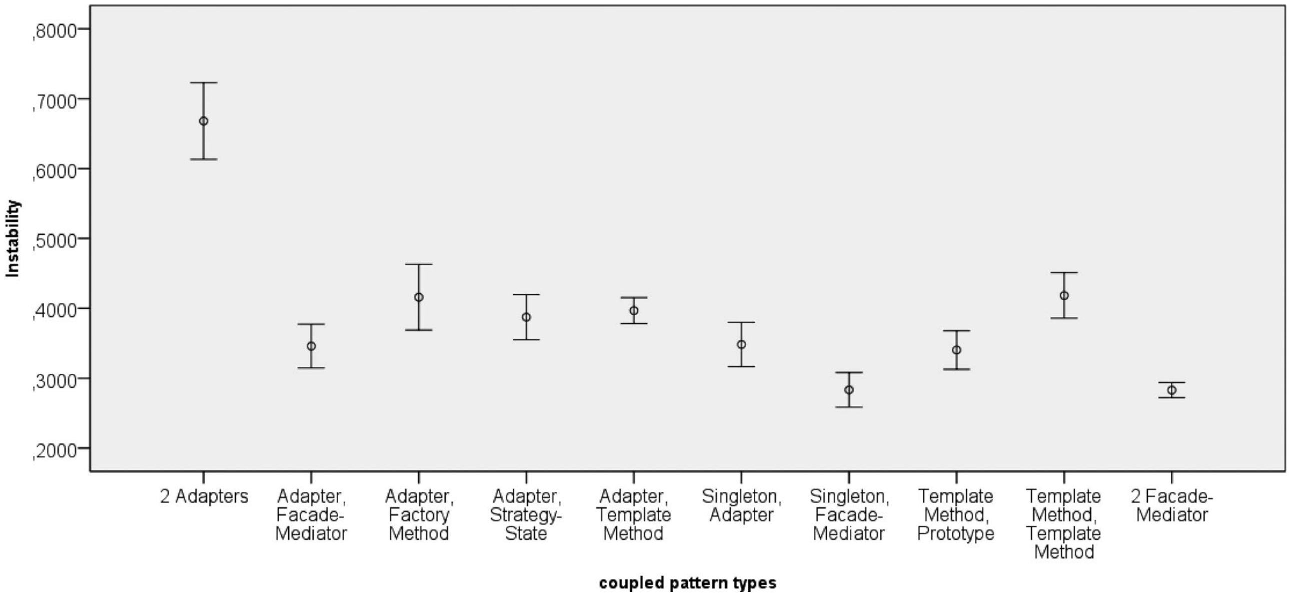


Fig. 10. Coupled pattern types vs. instability (error bars).

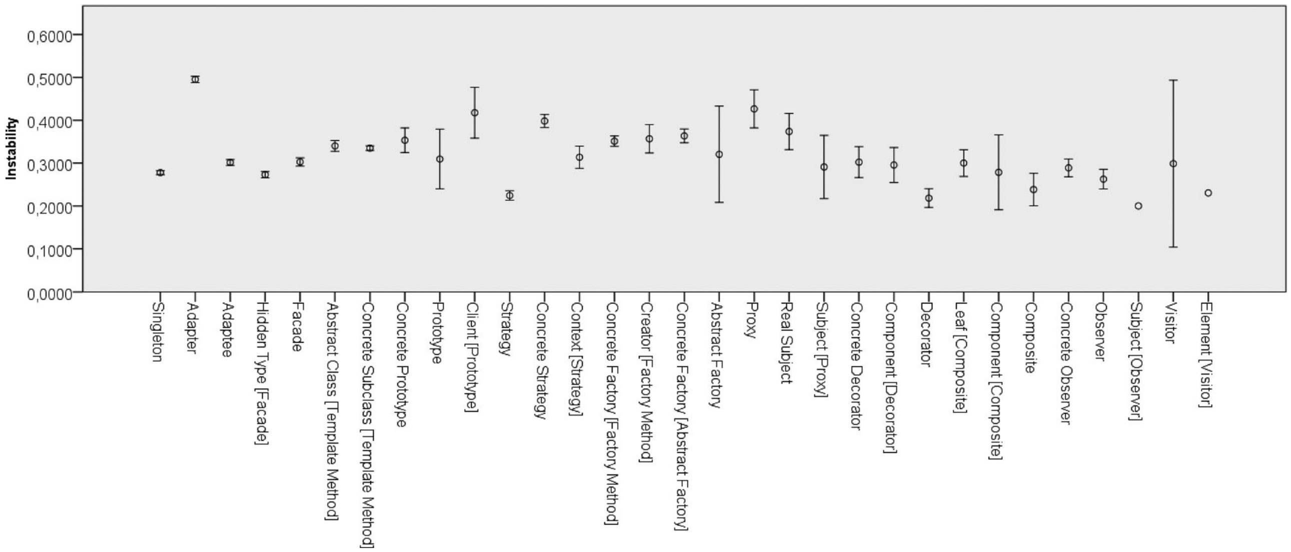


Fig. 11. Pattern roles vs. instability (error bars).

design pattern occurrences exhibit similar levels of instability because in most of them there are overlaps on their 95% confidence interval error bars. However, the ANOVA test suggests that mean values of instability differ across groups, i.e., different types of coupled patterns (F: 53.66 and sig: 0.00). The Hochberg's GT2 Post-Hoc analysis pointed out the most stable and most unstable pattern couplings, as follows:

Most Unstable Couplings :

- 2 Adapters

Most Stable Couplings :

- 2 Façade-Mediator
- Façade-Mediator, Singleton

The instability of classes that participate in more than one design pattern occurrence is statistically significantly different across different types of coupled GoF design patterns.

5.3 RQ$_3$: Design Pattern Roles

To investigate RQ$_3$ we performed the same analysis, i.e., error bars, ANOVA and Hochberg's GT2 Post-Hoc tests, on the groups formed by the primary pattern roles and synthesized pattern roles, defined in Tables 6 and 7:

H$_{0(f)}$ The instability of a class that participates in a single GoF design pattern occurrence is the same, regardless of the role that the class plays in the GoF design pattern occurrence.

Concerning roles of specific GoF design patterns (see error bars on Fig. 11), the ANOVA test suggests that the mean values of instability across different pattern roles have statistically significant differences (F: 118.52 and sig: 0.00). Thus, different roles are subject to different change propagation from the classes not participating in the pattern, because some roles are "shielded" inside the pattern occurrence while others are not. The results of the Hochberg's GT2 Post-Hoc tests suggest that there are certain motifs on the roles that differ from others. These differences can be mainly explained by the purpose of the role in the GoF design patterns, as described in Table 7. The findings illustrated in Fig. 11 are summarized in Table 10.

As observed in association/aggregation-based patterns (e.g., Adapter and Façade), the class playing the Aggregate role is more unstable than the class playing the Component role. On the other hand, in inheritance-based patterns (e.g., Strategy and Observer), the concrete subclasses are more unstable than abstract classes (see discussion on Section 6.3). The only exception are Template Method occurrences. The results of ANOVA validate that the mean value of instability is different across groups (F: 458.32 and sig: 0.00).

To further investigate the relationship between the role that a class plays in a pattern and its instability, we differentiate between instability caused by:

A) Dependencies among classes within the same pattern (pattern-internal dependencies);

B) Dependencies between a pattern-participating class and a class "outside" the pattern (pattern-external dependencies).

The distinction between the two types of instability is important because instability of type (b) can be used for assessing the shielding from the "*outside world*" that a role offers to the rest of the pattern-participating classes, whereas instability of type (a) is representing the instability caused by the structure of the pattern itself. To this end, in Table 11, we present the average number of dependencies (both internal and external) and the average REM per dependency, for each pattern role. We list only pattern roles that can be

TABLE 10
Instability of GoF Design Patterns Participant Roles

GoF Design Pattern	Roles Instability Comparison
Adapter	Aggregate > Component
Façade-Mediator	Aggregate > Component
Strategy-State	Subclass > Superclass
Factory Method	Subclass > Superclass
Prototype	Subclass > Superclass
Proxy	Subclass > Superclass
Abstract Factory	Subclass > Superclass
Observer	Subclass > Superclass
Template Method	Superclass > Subclass

TABLE 11
Dependencies and REM for GoF Design Pattern Roles

GoF Design Pattern	Roles	AVG (Dependencies)		AVG (REM)	
		Internal	External	Internal	External
Adapter	Aggregate	1.00	5.52	0.13	0.18
	Component	0.00	2.80	0.00	0.22
Façade-Mediator	Aggregate	2.86*	2.25	0.15	0.19
	Component	1.00	1.55	0.24	0.15
Strategy- State	Subclass	1.00	2.40	0.38	0.34
	Superclass	0.00	0.97	0.00	0.21
Factory Method	Subclass	1.00	2.61	0.19	0.14
	Superclass	0.00	2.81	0.00	0.20
Prototype	Subclass	1.00	2.65	0.27	0.21
	Superclass	0.00	2.53	0.00	0.20
Proxy	Subclass	1.50	3.13	0.25	0.20
	Superclass	0.00	1.50	0.00	0.35
Abstract Factory	Subclass	1.00	2.83	0.15	0.12
	Superclass	0.00	2.42	0.00	0.10
Observer	Subclass	1.00	1.85	0.31	0.24
	Superclass	0.50	1.55	0.11	0.16
Template Method	Subclass	1.00	2.11	0.28	0.18
	Superclass	0.00	2.71	0.00	0.17

* *The average number of Hidden Types / Colleagues are not related to the structure of the pattern, but is empirically retrieved.*

classified either as aggregate/component (when they participate in part-whole relations or a simple association) or as subclass/superclass (when they participate in generalization relationships). The results of Table 11 suggest that:

- *Pattern-external dependencies* are higher in number than *pattern-internal dependencies* and therefore more important concerning the instability of the pattern participating classes;
- The only patterns with more than one *pattern-internal dependency* per role are Proxy and Façade-Mediator;
- In *association/aggregation-based patterns,* pattern participating classes are more tightly coupled (higher REM) to *pattern-external* classes than pattern internal classes;
- In *inheritance-based patterns,* pattern participating classes are more tightly coupled (higher REM) to *pattern-internal* classes (i.e., their superclass) than pattern external;
- The average number of dependencies and REM, regardless of the type of dependency (external or internal), is in agreement with the results on the Instability of GoF Design Pattern participating classes presented in Table 10;
- The roles that are *most "shielded"* behind an aggregate class or a superclass are the *Hidden Type* (Façade-Mediator), the *Concrete Subclass* (Template Method), and the *Concrete Observer* (Observer).

> *The instability of classes that participate in design pattern occurrences is statistically significantly different across different GoF design patterns roles*

6 Discussion

In this section, we discuss the findings of our case study organized per research question and in comparison to the previous work. We remind that all evidence reported from the literature on the stability of design patterns, in fact deal with change proneness, i.e., the actual changes that occur in a class, merging changes from new requirements, debugging, and change propagation, whereas our study focuses only on the impact of change propagation.

6.1 Number of Pattern Occurrences

The state-of-the-art on the change proneness of GoF design pattern-participating classes suggests that classes that participate in GoF design patterns change more often than the classes that do not participate in design pattern occurrences [9], [10], [20]. These results are validated from our case study (see Fig. 5), if we do not differentiate between single and coupled design pattern occurrences. Such a distinction has not been made in any previous studies. The two studies [27], [34] that investigate the effect of design pattern coupling on quality attributes are not related to instability or change proneness.

By distinguishing between single and coupled design pattern occurrences, we observe that classes that participate in a single design pattern are, on average, slightly more stable than classes that are not pattern participants. This is an intuitive result, because design patterns provide decoupling, which stops changes from being propagated to the classes that participate in the design pattern. On the contrary, classes that participate in more than one design pattern occurrences are clearly losing this advantage (see Figs. 5 and 7). This result is also intuitive because the more responsibilities a class is assigned, the more unstable it becomes in terms of propagated changes. This observation is due to the fact that a class with more responsibilities must communicate with more classes, increasing the number of external dependencies, thus rendering it more "vulnerable" to change propagation. The results are in accordance to those of Khomh and Guéhéneuc, that suggest that quality characteristics, such as coupling, cohesion, and complexity appear to be worse in classes that participate in more than two design pattern occurrences, rather than classes that participate in one or zero design pattern occurrences [27].

However, for the first observation (single pattern-participating classes are more stable than the non-pattern participating classes) to be statistically significant, we must take an additional parameter into consideration: application domain. The abovementioned claims are statistically significant for the application domains of *Games, Home and Education Applications, Development Tools,* and *Science and Engineering Applications*. This fact suggests that design pattern occurrences in these domains are loosely coupled to the rest of the system and thus become more resistant to propagation of changes. On the contrary, in *Communication Tools, Business and Enterprise, Audio and Video,* and *Graphics* applications, classes that participate in GoF design pattern occurrences are more prone to change propagation. Therefore, developers of such applications should be aware of the increased probability of changes propagating to pattern participating classes from the rest of the system. The results of our study are in accordance to those of Vasquez et al. [48], that suggest that other indirect quality indicators (such as anti-patterns or code smells) vary among different application domains as well.

6.2 Design Pattern Type

A different perspective for further investigating the effect of GoF design patterns on stability is not to examine the set of patterns as a whole, but independently, according to the type of each design pattern occurrence. To the best of our knowledge, the only research results that are comparable to ours are the results of Aversano et al. [7]. However, Aversano et al. have not separately investigated single from coupled design pattern occurrences, while they measured change proneness by summing up changes due to new requirements, debugging, and change propagation. Because Aversano et al. [7] have not studied the same set of patterns, we only compare results on the common subset. Our results suggest that similarly to anti-patterns [39], each design pattern has a different effect on change-proneness.

According to the results of Fig. 9 and the accompanying ANOVA test, the most stable classes can be found in Singleton, Façade-Mediator, Observer, Composite, and Decorator occurrences. Classes that participate in Proxy and Adapter occurrences are more unstable than other pattern-participating and non-pattern-participating classes. These results are similar to those of Aversano et al., in terms of Singleton, Adapter, Composite and Decorator, but are not completely similar for Observer.[5] A possible reason is that Aversano et al. [7] investigated change proneness incurred by the addition, deletion. or modification of Observer occurrences, whereas in our case the addition or deletion of Concrete Observers or Subjects does not lead to the propagation of any change: adding or deleting a subclass in an hierarchy does not change the dependencies and the REM value of the other classes in the system (Fig. 3a).

The fact that the Singleton design pattern occurrence is more stable than other design pattern occurrences can be explained by the fact that it consists of a single class, which does not have to carry additional dependencies to implement the pattern. This lack of dependencies limits the classes through which it can receive change requests and therefore its instability. Concerning Façade-Mediator, their low levels of instability might be caused by the *clean* separation they provide between subsystems. When using *Façade* or *Mediator*, the communication between *Colleagues* is synchronized by a single class, limiting the number of dependencies between them, and therefore their instability. Finally, the fact that the instabilities of Composite and Decorator are similar is intuitively correct, because these patterns share a common structure.

Among the *inheritance-based patterns* Proxy is the most unstable, because it is the only one in which one subclass is dependent on the other, increasing the number of internal dependencies (see Table 11). Concerning *association/aggregation-based patterns,* Adapter, which provides a means for reusing the functionality of a class, is highly unstable, especially in the role of the Adapter that holds a large number of external dependencies (see Table 11).

Another interesting result with respect to design pattern type instability is that the three creational patterns that have been investigated, i.e., Factory Method, Abstract Factory, and Prototype exhibit similar levels of instability. This fact implies that all creational patterns are implemented in a similar way in terms of dependencies and that their structural differences are not strongly affecting their stability.

The results of investigating the stability of coupled design patterns (see Fig. 10) suggest that coupling: (a) two Façade/Mediator or (b) Façade/Mediator with Singleton occurrences does not have a substantial negative effect on stability. Combining two Adapter occurrences should be avoided, because this pattern combination exhibits high levels of instability for the involved classes. These results cannot be compared to any other results in the literature, because this is the first time that design pattern coupling is being explored with respect to stability. However, we can observe that the effect of single patterns on instability is propagated to the coupled pattern occurrences. For example, one of the most stable single patterns, i.e., Façade-Mediator is part of the most stable coupled pattern occurrences. In particular, when combining two occurrences of stable design patterns, e.g., Façade-Mediator (instability ≈ 0.28), the produced coupled pattern is slightly more unstable than the single pattern occurrences. However, the coupled pattern remains the most stable among coupled pattern occurrences (instability ≈ 0.30). On the other hand, Adapter, which is one of the patterns with the most unstable single pattern occurrences, is part of the two most unstable coupled pattern types.

6.3 Design Pattern Roles

Finally, when taking into account the roles that a class can play in GoF design pattern occurrences, we observed that the instability of the role depends on the mechanism that the pattern uses for relating classes, i.e., association or aggregation (e.g., Adapter, Façade, etc.) or inheritance (Strategy, Observer, etc.) and the type of pattern. From the

[5] The Proxy, Façade, and Mediator design pattern are not examined in Aversano et al. [6]

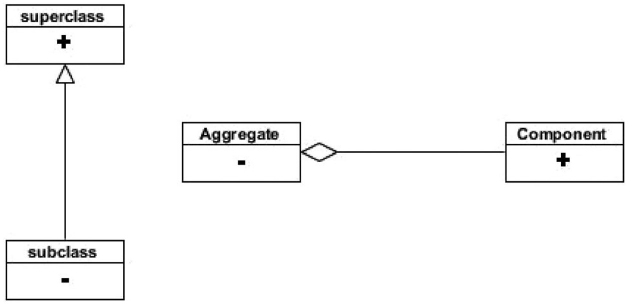


Fig. 12. Stability of pattern roles w.r.t. class relation.

results of this study, we observe that the most stable design pattern roles are the `Strategy` and the `Hidden Type` (*Façade-Mediator*). Both results are intuitively correct, because:

- *Strategy* classes are in many cases purely abstract classes that hold a limited number of dependencies to other classes (0.97 in general -they usually hold 0 or 1 dependencies). Therefore, they are not heavily dependent on the public interface of other classes and their role is not prone to change propagation;
- *Hidden Type* classes are expected to be "*shielded*" behind Façade or Mediator classes. The original intention of these classes (i.e., Façade and Mediator) is to handle the communication among Hidden Type classes, thereby decoupling them. This decoupling offers shielding against change propagation from classes outside the pattern.

The results of the study suggest that in a class hierarchy, the superclass is more stable than the subclass, whereas in an association or a "part-whole" relationship the component (or independent) class is more stable than the aggregate (or dependent) class, as shown in Fig. 12. This result might appear to be unexpected because the *Aggregate* and the *Superclass* are the channel through which the pattern is communicating with the rest of the system. However, a closer investigation of how the mechanisms of inheritance and aggregation or association work explains these results.

Comparing the stability of the two roles in a hierarchy, *Subclasses* inherits any dependency of the *Superclass.* So, if the *Superclass* depends on many other "pattern-external" classes, these dependencies are also added in the dependency list of the *Subclass*. In addition, exactly because *Subclasses* supply concrete functionality, they usually must collaborate with other classes, further increasing the number of external dependencies. Finally, they also hold a strong relationship with their superclass. Therefore, the dependencies of a *Subclass* are by definition more than those of a *Superclass*. The fact that change proneness increases as the depth of inheritance of a class becomes higher has also been observed by Bieman et al. [10]. Additionally, the aforementioned results agree with those of Khomh et al. [29], which suggest that abstract classes are less change prone than children classes.

On the contrary, in object association/aggregation the *Aggregate* role is less stable, because: (a) it communicates with all the *Components* of the sub-system (which act as suppliers of concrete functionality) and (b) it handles the communication between different sub-systems, whereas the dependencies of *Components* (which are shielded behind the interface of the *Aggregate* and usually perform limited interaction outside their boundaries) are at most equal to the number of all *Components* in the same sub-system. Therefore, the number of dependencies of the classes playing the *Component* roles is less or equal to the number of dependencies of classes playing the role of the *Aggregate*.

7 Threats to Validity

In this section, we present and discuss construct, reliability, external, and internal validity threats for this study [41]. Construct validity reflects to what extent the phenomenon under study really represents what is investigated according to the research questions. The reliability of the case study is related to whether the data is collected and the analysis is conducted in a way that can be replicated with the same results. External validity deals with possible threats while generalizing the findings derived from the sample to population. Finally, internal validity is related to identification of confounding factors, i.e., factors other than the independent variables that might influence the value of the dependent variable.

7.1 Construct Validity

The threat to construct validity is related to the accuracy of the tools and approach used to assess class instability and to detect design pattern occurrences. This is a construct validity in the sense that inaccurate results might lead to measuring a different phenomenon than the one that we originally intended to investigate.

The selected algorithm for calculating class instability considers, as explained in Section 3.2, the dependencies in the system's structure through which changes can propagate from one class to another [46]. The calculation of the possibility of future changes is by nature an ambitious goal that cannot be achieved with high levels of accuracy, considering the numerous factors that might affect the decision of a designer to modify a classes, so this would be an important threat to validity if our study's objective would be to estimate change proneness. Likewise, setting a constant value for the internal probability of change for system classes would also be a threat for accurately estimating change proneness. However, the goal of this study is to assess instability rather than the actual change proneness, which is a straightforward procedure that is accurate, in the sense that it is solely based on class dependency analysis. Therefore, there is no real threat to validity from estimating instability.

In addition to that, concerning the accuracy of the selected design pattern detection tool, one possible threat is related to the possibility of considering false positives in our study or of neglecting true occurrences (false negatives) [45]. To mitigate the threat regarding false positives, two of the authors performed manual validation of design pattern occurrences. Based on the results the authors performed several enhancement actions, as presented in Section 3.1. Finally, we believe that the precision rates in our dataset is higher than the ones reported in [11], for two reasons:

a) Based on our manual validation, we observed that many false-positives were identified due to misplacement of a pattern occurrence between similar patterns (e.g., Façade as Mediator, State as Strategy and vice-versa). This threat is completely mitigated in our study because we report results on such patterns as one.

a) The patterns that have been used in [11] are quite complex in their structure and therefore the chance of observing a misclassification on them is higher than for simpler patterns (about 70-75 percent) [11]. Also occurrences of simpler patterns is accurate by these tools in approximately 80% of the cases [11]. Following this observation, and the frequency of occurrence of these patterns in our dataset, we believe that the precision in our dataset is higher.

The impact of false negatives in our study (due to limited recall rate in certain patterns) is alleviated by the fact that the examined data set contains already a vast number of occurrences. Even if true occurrences have remained undetected, the dataset is sufficiently large to enable the investigation of the instability of pattern-participating classes, at least for a number of patterns. In any case, as a mitigation action for this threat we preferred to use the union of the results of the used pattern detection tool, instead of using the intersection.

7.2 Reliability

To mitigate threats to reliability, two different researchers were involved in the data collection and one double-checked the results of the other. Furthermore, one researcher double-checked the results of the data analysis performed by another researcher. All primitive data can be reproduced by using the *percerons.com* online repository or the tools mentioned in Section 3.

7.3 External Validity

Concerning external validity, we have identified three possible threats to the validity of our results. Firstly, all the investigated systems are written in Java and there is a possibility that the results would be different for other object-oriented languages and other patterns. Secondly, we have examined fifteen (15) out of the twenty three (23) design patterns described by Gamma et al. [19], thus the results cannot be generalized to the rest of the GoF design patterns, since their stability may differ. Finally, the results of the study cannot be generalized to “special” implementations of design patterns instances in which there are no static relationships among classes playing different roles. For example, in the *reflective implementation of the Visitor design pattern*, the accept method uses reflection to choose the appropriate method to call on a Visitor. However, these are certainly exceptional cases and therefore we believe they pose a minor threat to the validity of the results.

7.4 Internal Validity

Finally, we consider pattern participation as an instability factor, i.e., we examine how the roles that classes have due to their participation in patterns lead to instability. However, a class may have other responsibilities outside the pattern, which may also result in dependencies and thus cause instability. This may potentially be a confounding factor and therefore constitutes an internal threat to validity [41], in the sense that factors other than the independent variables (pattern participation) affect the value of the dependent variable (instability). To exclude other possible factors of instability, we should compare two versions of the same class, one designed with a GoF pattern and one designed with an alternative solution. The two systems would offer the same functionality, with the only change being the application of the pattern itself. In such a case, it would be possible to investigate the effect of design patterns isolated from the other change factors. However, such cases are extremely difficult to find in existing real-world examples or even to implement artificially on a large scale. Furthermore, GoF design patterns have been associated with a large number of design alternatives, which can substitute the role of the pattern [2]. Therefore, even if we set up such an experiment, it would only be possible to compare the GoF design patterns to a limited number of specific design alternatives. Thus, both conducting a case study or a controlled experiment have their own limitations and none of the two would actually mitigate this risk.

8 Conclusion

This study investigated the effect of GoF design patterns on class stability. To achieve this goal, we conducted a multi-case study on about 65.000 open-source Java classes to explore the probability of a class to change, due to propagation of changes that occurred in other classes. To assess the stability of GoF design patterns, we examined classes that participate in zero, one, or more design pattern occurrences. To the best of our knowledge, this is the largest case study on the effect of GoF design patterns on stability and the only study that reveals the different levels of stability between classes that participate in one or more pattern occurrences.

The results of the case study indicate that classes that play exactly one role in a GoF design pattern are more stable than classes that play zero or more than one role in GoF design pattern occurrences. However, the level of statistical significance of that claim varies across different application domains. The results also suggest that different GoF design patterns provide different levels of stability to the classes that participate in them. For example, Singleton, Façade-Mediator, Observer, Composite, and Decorator occurrences seem to consist of classes that are more resistant to changes propagating from other classes. Finally, the role that a class plays in a design pattern is also an indicator of its resistance to propagation of changes. We observed that the use of association/aggregation for establishing object communication, classes that play the *Aggregate* role are less stable than classes that play the *Component* role. On the other hand, in design patterns that involve inheritance, public *Superclasses* are more stable than *Subclasses*.

The aforementioned results are valuable to practitioners, because they provide indications for testing and refactoring prioritization. First, concerning testability, classes that are less resistant to change propagation should be checked for defects more often and more exhaustively, because they are expected to be more defect-prone. Second, concerning refactorings, due to the harmful effects of instability, classes that are less resistant to change propagation should be refactored to more stable designs.

Additionally, we strongly believe that GoF design patterns are not uniformly impacted by all possible sources of change, such as propagation from other classes, accommodation of new requirements, and removal of defects. Treating all potential sources of change as a common type might lead to coarse-grain conclusions, because a particular design pattern might be beneficial in preventing one type of change and less helpful in shielding from other types of changes. Thus, as a line of future research one could investigate the susceptibility of GoF design pattern-participating classes to change with respect to factors other than the propagation of change, such as modifications due to corrective or adaptive maintenance. This could be performed by contrasting instability and change proneness taking into account the history of changes of system classes. Specifically, while inspecting past data, one would have to consider not only method signature changes, but also changes in the contract of classes, which might emit changes. By distinguishing among different types of actual changes, it would be possible to investigate whether design pattern roles offer selective shielding, in terms of types of change (e.g., from new requirements, from propagated changes, or from bug fixing).

Furthermore, the methodology that has been described in this paper analyzes the proneness of classes that participate in patterns to change, due to changes occurring in other classes. The analysis can also be performed at a more fine-grained level, that is, by examining the susceptibility of individual methods to change, which would require the analysis of dependencies between methods.

ACKNOWLEDGMENTS

This research was cofinanced by the European Union (European Social Fund-ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)-Research Funding Program: Thalis-Athens University of Economics and Business-SOFTWARE ENGINEERING RESEARCH PLATFORM. The authors would like to thank the Associate Editor Prof. Éric Tanter, and the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. Apostolos Ampatzoglou is the corresponding author.

REFERENCES

[1] A. Ampatzoglou, A. Gkortzis, S. Charalampidou, and P. Avgeriou, "An Embedded multiple-case study on OSS design quality assessment across domains," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2013, pp. 255–258.

[2] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Design pattern alternatives: What to do when a GoF pattern fails," in *Proc. 17th PanHellenic Conf. Informat.*, Sep. 2013, pp. 122–127.

[3] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object oriented design patterns in game development," *Inf. Softw. Technol.*, vol. 49, no. 5, pp. 445–454, May 2007.

[4] A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, "A methodology to assess the impact of design patterns on software quality," *Inf. Softw. Technol.*, vol. 54, no. 4, pp. 331–346, Apr. 2012.

[5] A. Ampatzoglou, O. Michou, and I. Stamelos, "Building and mining a repository of design pattern instances: Practical and research benefits," *Entertainment Comput.*, vol. 4, no. 2, pp. 131–142, Apr. 2013.

[6] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Research state of the art on GoF design patterns: A mapping study," *J. Syst. Softw.*, vol. 86, no. 2, pp. 1945–1964, Jul. 2013.

[7] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Sep. 2007, pp. 385–394.

[8] V. R. Basili, G. Caldiera, and H. D. Rombach, "Goal Question Metric paradigm," in *Encyclopedia of Software Engineering*, West Sussex, U.K.: Wiley, 1994, pp. 528–532.

[9] J. M. Bieman, D. Jain, and H. J. Yang, "OO design patterns, design structure, and program changes: An industrial case study," in *Proc. 17th Int. Conf. Softw. Maintenance*, Nov. 2001, pp. 580–589.

[10] 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 *Proc. 9th Int. Softw. Metrics Symp.*, Sep. 2003, pp. 40–49.

[11] A. Binun and G. Kniesel, "Witnessing Patterns: A data fusion approach to design pattern detection," Inst. of Comput. Sci. III, Univ. Bonn, Bonn, Germany, Tech. Rep. IAI-TR-2009–02, Jan. 2009.

[12] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. A. Van Dooren, "Measure of similarity between graph vertices: Applications to synonym extraction and web searching," *SIAM Rev.*, vol. 46, no. 4, pp. 647–666, 2004.

[13] S. A. Bohner, "Impact analysis in the software change process: A year 2000 perspective," in *Proc. Int. Conf. Softw. Maintenance*, Nov. 1996, pp. 42–51.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. West Sussex, U.K.: Wiley, 1996.

[15] M. Di Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of relationships between design pattern roles and class change proneness," in *Proc. 24th Int. Conf. Softw. Maintenance*, Sep./Oct. 2008, pp. 217–226.

[16] M. Elish, "Do structural design patterns promote design stability?" in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf.*, Sep. 2006, pp. 215–220.

[17] A. Field, *Discovering Statistics Using SPSS*, 3rd ed. Newbury Park, CA, USA: SAGE, 2005.

[18] M. Fowler, *Analysis Patterns: Reusable Object Models*. Reading, MA, USA: Addison-Wesley, Oct. 1996.

[19] E. Gamma, R. Helms, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1995.

[20] M. Gatrell, S. Counsell, and T. Hall, "Design patterns and change proneness: A replication using proprietary C# software," in *Proc. 16th Working Conf. Reverse Eng.*, Oct. 2009, pp. 160–164.

[21] F. M. Haney, "Module connection analysis: A tool for scheduling of software debugging activities," in *Proc. AFIPS Fall Joint Comput. Conf.*, Dec. 1972, pp. 173–179.

[22] S. Hassaine, F. Boughanmi, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "A seismology-inspired approach to study change propagation," in *Proc. 27th Int. Conf. Softw. Maintenance*, Sep. 2011, pp. 25–30.

[23] E. Horowitz and R. C. Williamson, "SODOS: A software documentation support environment—Its definition," *IEEE Trans. Softw. Eng.*, vol. 12, no. 8, pp. 849–859, Aug. 1986.

[24] Information Technology: Software Product Evaluation, Quality Characteristics and Guidelines for Their Use, Int. Organisation for Standardization, ISO 9126, 1992.

[25] F. Jaafar, Y.-G. Guéhéneuc, S. Hammel, and G. Antoniol, "Detecting asynchrony and dephase change patterns by mining software repositories," *J. Softw.: Evol. Process.*, vol. 26, no. 1, pp. 77–106, Jan. 2014.

[26] S. Jeanmart, Y.-G. Guéhéneuc, H. Sahraoui, and N. Habra, "A study of the impact of the Visitor design pattern on program comprehension and maintenance tasks," in *Proc. 3rd Int. Symp. Empirical Softw. Eng. Meas.*, Lake Buena Vista, FLA, USA, Oct. 2009, pp. 69–78.

[27] F. Khomh and Y.-G. Guéhéneuc, "Playing roles in design patterns: An empirical descriptive and analytic study," in *Proc. 25th Int. Conf. Softw. Maintenance*, Sep. 2009, pp. 83–92.

[28] F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *Proc. 12th Eur. Conf. Softw. Maintenance Reeng.*, Athens, Greece, Apr. 2008, pp. 274–278.

[29] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact if code smells on software change-proneness," in *Proc. 16th Working Conf. Reverse Eng.*, Lille, France, Oct. 2009, pp. 75–84.

[30] K. Kouskouras, A. Chatzigeorgiou, and G. Stephanides, "Facilitating software extension with design patterns and aspect-oriented programming," *J. Syst. Softw.*, vol. 81, no. 10, pp. 1725–1737, Oct. 2008.

[31] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*. London, U.K.: Academic, 1985.

[32] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Softw. Testing, Verification Rel.*, vol. 23, no. 8, pp. 613–646, Dec. 2013.

[33] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[34] W. B. McNatt and J. M. Bieman. "Coupling of design patterns: Common practices and their benefits," in *Proc. 25th Int. Comput. Softw. Appl. Conf. Invigorating Softw. Develop.*, Oct. 2001, pp. 574–579.

[35] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1997.

[36] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu, "Toward effective deployment of design patterns for software extension: A case study," in *Proc. 4th Workshop Softw. Qualit*, Shanghai, China, May 2006, pp. 51–56.

[37] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu, "Do maintainers utilize deployed design patterns effectively?" in *Proc. 29th Int. Conf. Softw. Eng.*, May 2007, pp. 168–177.

[38] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta, "A controlled experiment in maintenance comparing design patterns to simpler solutions," *IEEE Trans. Softw. Eng.*, vol. 27, no. 12, pp. 1134–1144, Dec. 2001.

[39] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," in *Proc. 19th Working Conf. Reverse Eng.*, Oct. 2012, pp. 437–446.

[40] P. Rovegard, L. Angelis, and C. Wohlin, "An empirical study on views of importance of change impact analysis issues," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 516–530, Apr. 2008.

[41] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. West Sussex, U.K.: Wiley, 2012.

[42] P. Sommerlad and M. Rüedi, "Do-it-yourself reflection," in *Proc. 3rd Eur. Conf. Pattern Lang. Programm. Comput.*, 1998.

[43] N. Shi and R. Olson, "Reverse engineering of design patterns from Java source code," in *Proc. 21st Int. Conf. Automated Softw. Eng.*, Sep. 2006, pp. 123–134.

[44] SPSS Inc, "SPSS 16.0 Data Preparation," *SPSS Manual*, retrieved on-line at, Sep. 2013.

[45] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006.

[46] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 601–614, Jul. 2005.

[47] A. Van Koten and A. R. Gray, "An application of Bayesian network for predicting object-oriented software maintainability," *Inf. Softw. Technol.*, vol. 48, no. 1, pp. 59–67, Jan. 2006.

[48] M. L. Vasquez, S. Klock, C. McMillan, A. Sabane, D. Poshyvanyk and Y.-G. Guéhéneuc, "Domain matters: Bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps," in *Proc. 22nd Int. Conf. Program Comprehension*, Jun. 2014, pp. 232–243.

[49] M. Vokác, W. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns: A replication in a real programming environment," *Empirical Softw. Eng.*, vol. 9, no. 3, pp. 149–195, Sep. 2004.

[50] S. Yau and J. Collofello, "Some stability measures for software maintenance," *IEEE Trans. Softw. Eng.*, vol. 6, no. 6, pp. 545–552, Nov. 1980.

[51] S. Yau and J. Collofello, "Design stability measures for software maintenance," *IEEE Trans. Softw. Eng.*, vol. 11, no. 9, pp. 849–856, Sep. 1985.

[52] R. K. Yin, Case Study Research: Design and Methods, 3rd ed. Los Angeles, California, USA: Sage, 2003.

[53] C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1213–1231, Sep./Oct. 2012.

Apostolos Ampatzoglou received the BSc degree in information systems in 2003, the MSc degree in computer systems in 2005, and the PhD degree in software engineering by the Aristotle University of Thessaloniki in 2012. He is an assistant professor in the Johann Bernoulli Institute for Mathematics and Computer Science of the University of Groningen, The Netherlands, where he carries out research and teaching in the area of software engineering. His current research interests are focused on reverse engineering, software maintainability, software quality management, open source software engineering and software design. He has published more than 25 articles in international journals and conferences. He is/was involved in more than 10 R&D ICT projects, with funding from national and international organizations.

Alexander Chatzigeorgiou received the Diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. He is an associate professor of software engineering in the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. From 1997 to 1999, he was with Intracom S.A., Greece, as a telecommunications software designer. Since 2007, he has also been a member of the teaching staff at the Hellenic Open University. His research interests include object-oriented design, software maintenance, and software evolution analysis. He is a member of the IEEE and the Technical Chamber of Greece.

Sofia Charalampidou received the BSc degree in information technology from the Technological Institute of Thessaloniki, Greece, and the MSc degree in software engineering from the Chalmers University of Technology, Sweden. She is currently working toward the PhD degree at the University of Groningen, The Netherlands, in the group of Software Engineering and Architecture. Her research interests include software design, maintenance, and metrics.

Paris Avgeriou is a professor of software engineering in the Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, The Netherlands where he has led the Software Engineering research group since September 2006. Before joining Groningen, he was a postdoctoral fellow of the European Research Consortium for Informatics and Mathematics (ERCIM). He has participated in a number of national and European research projects directly related to the European industry of Software-intensive systems. He has coorganized several international conferences and workshops (mainly at the International Conference on Software Engineering-ICSE). He sits on the editorial board of *Springer Transactions on Pattern Languages of Programming (TPLOP)*. He has edited special issues in *IEEE Software*, *Elsevier Journal of Systems and Software* and *Springer TPLOP*. He has published more than 130 peer-reviewed articles in international journals, conference proceedings and books. His research interests lie in the area of software architecture, with strong emphasis on architecture modeling, knowledge, evolution, patterns and link to requirements. He is a senior member of the IEEE.