Information and Software Technology 105 (2019) 1–16

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes?

Daniel Feitosa[a,*], Apostolos Ampatzoglou[a], Paris Avgeriou[a], Alexander Chatzigeorgiou[b], Elisa.Y. Nakagawa[c]

[a] Department of Mathematics and Computer Science, University of Groningen, The Netherlands
[b] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
[c] Department of Computer Systems, University of São Paulo, Brazil

ARTICLE INFO

*Keywords:*
Software architecture
GoF patterns
Design
Quality analysis
Evaluation

ABSTRACT

*Context:* GoF patterns have been extensively studied with respect to the benefit they provide as problem-solving, communication and quality improvement mechanisms. The latter has been mostly investigated through empirical studies, but some aspects of quality (esp. run-time ones) are still under-investigated.
*Objective:* In this paper, we study if the presence of patterns enforces the conformance to good coding practices. To achieve this goal, we explore the relationship between the presence of GoF design patterns and violations of good practices related to source code correctness, performance and security, via static analysis.
*Method:* Specifically, we exploit static analysis so as to investigate whether the number of violations of good coding practices identified on classes is related to: (a) their participation in pattern occurrences, (b) the pattern category, (c) the pattern in which they participate, and (d) their role within the pattern occurrence. To answer these questions, we performed a case study on approximately 13,000 classes retrieved from five open-source projects.
*Results:* The obtained results suggest that classes not participating in patterns are more probable to violate good coding practices for correctness, performance and security. In a more fine-grained level of analysis, by focusing on specific patterns, we observed that patterns with more complex structure (e.g., Decorator) and pattern roles that are more change-prone (e.g., Subclasses) are more likely to be associated with a higher number of violations (up to 50 times more violations).
*Conclusion:* This finding implies that investing in a well-thought architecture based on best practices, such as patterns, is often accompanied with cleaner code with fewer violations.

## 1. Introduction

Design patterns have been introduced in the software engineering literature by Gamma et al. [1] (known as the Gang of Four (GoF)—Gamma, Helm, Johnson, and Vlissides), aiming to provide common solutions to recurring problems, while designing object-oriented (OO) systems. The GoF catalogue includes 23 patterns, organized into three categories (structural, behavioral, and creational), based on their purpose[1] [1]. Since their inception, GoF patterns have been widely explored by both researchers and practitioners, and are currently considered as a common practice for software development. In addition to their original purpose of solving OO design problems [2], their effect on quality attributes (QAs) has also been widely investigated, according to two mapping studies by Ampatzoglou et al. [3] and Mayvan et al. [4]. However, the current state of the research has two main limitations:

- ***Limited number of studies related to run-time qualities***. In particular, on the one hand, several empirical studies have explored the impact of GoF design patterns on design-time QAs such as modifiability and reusability (see for example [4]). On the other hand, research on the effect of GoF patterns on run-time QAs, such as security and performance, is fairly limited [4]. Although GoF patterns are not originally intended to serve any run-time QA in particular, some indirect effect, either positive or negative, is to be

* Corresponding author.
*E-mail address:* d.feitosa@rug.nl (D. Feitosa).

[1] According to Gamma et al., there is a second GoF patterns categorization based on their scope (i.e., class vs. object). However, this categorization was not considered in this study.

https://doi.org/10.1016/j.infsof.2018.07.014
Received 30 December 2017; Received in revised form 26 July 2018; Accepted 30 July 2018
Available online 06 September 2018
0950-5849/ © 2018 Elsevier B.V. All rights reserved.

expected. For instance, developers often use GoF design patterns as communication mechanisms, which facilitate the understanding of each other's code. As a consequence, code smells, such as Message Chains and Middle Man, can be avoided and, thus, performance improved, since the number of method calls is decreased.

- ***Run-time qualities have been explored mostly through dynamic analysis***. Until now, the limited work done on studying the effect of GoF patterns on run-time QAs was performed mostly by using dynamic analysis, i.e., by exploring the observed effects during the execution of a system. For example, researchers have used profilers for extracting memory usage or energy consumption data to investigate performance (e.g., Litke et al. [5] and Sahin et al. [6]). An alternative to dynamic analysis, for investigating the same phenomenon, is the employment of static analysis. Static analysis is an established method for performing code quality analysis [7,8,9,10,11], mostly because it is based upon an artifact that is always available to quality engineers (i.e., source code). There are also a few research efforts that employed static analysis for assessing run-time qualities, e.g., [12] for assessing reliability with GERT, [13] for assessing performance with FindBugs, [14,15,16] for assessing security, etc. With static analysis, one would be able to explore the underlying relationship between GoF patterns and run-time QAs without executing the systems in which they are instantiated. We believe that statically detecting violations of good coding practices that affect run-time QAs is complementary to dynamic analysis, since it promotes the assessment of different artefacts (e.g., violations instead of profilers output) and by using a different approach (static instead of dynamic analysis). We note that, even if the introduction of design patterns might not directly aim at removing violations, it may lead to a 'cleaner' and well-designed architecture, accompanied by better coding practices. Nevertheless, not all parts of the architecture can benefit from the introduction of patterns, as the pattern goal might be irrelevant to the functionality/design of that part. Design patterns are not a panacea as they cannot solve all design problems and all design problems cannot be solved by a design pattern.

Motivated by the aforementioned limitations, in this study, we exploit static analysis to explore whether the application of GoF patterns can be associated to the existence of violations of good coding practices related to three run-time QAs, namely, correctness, performance and security, as defined in the SQuaRE quality model [17]. We note that we consider correctness a run-time QA because, as performance and security, it is discernible at run-time [18]. We selected these QAs as they are highly relevant for both practitioners and researchers, with considerable literature addressing them. However, there is a lack of studies investigating them from the proposed perspective, i.e., by using static analysis to examine the effect of GoF patterns on them.

To estimate the effect of GoF patterns on the aforementioned qualities, we adopted the same approach used by Sahin et al. [6] and Gatrell and Counsell [19], i.e., we compare pattern-participating (PP) parts of the system against non-pattern-participating (NPP) parts. Similar to Ampatzoglou et al. [20] and Aversano et al. [21], our investigation is performed at class-level to standardize data collection and source code analysis. Specifically, by working on class-level we can discriminate between: PP classes – that participate in pattern occurrences – and NPP classes. Additionally, we further classify PP classes into: single-pattern-participating (SPP), i.e., those that participate in exactly one pattern occurrence; and coupled-pattern-participating (CPP), i.e., those that participate in more than one pattern occurrences. According to the literature (e.g., [20,22,23]), these two types of pattern participation can lead to diverse effects on QAs; therefore, we treat them separately in this study.

To explore the relationship of GoF patterns with the aforementioned QAs, we compare quality levels of classes (quantified by the number of violations) measured from four different perspectives, serving the sub-goals of this study:

***sg1.****Pattern participation*—by clustering classes according to their pattern participation, i.e., NPP, SPP and CPP. As previously mentioned, this perspective allows us to compare the number of violations concentrated in SPP and CPP elements against NPP elements.
***sg2.****Pattern category*—by clustering classes according to the pattern category in which they participate, i.e., creational, behavioral and structural. This perspective allows us to investigate whether there are differences in the relationship of GoF patterns of different categories on QAs.
***sg3.****Pattern*—by clustering classes according to the pattern in which they participate (e.g., Singleton, State, Strategy, etc.). This perspective allows finer grained observations of the relationship of applying GoF patterns and run-time QAs. It is rather common when investigating GoF patterns, as it represents the unit of the proposed solutions (i.e., the patterns) and can inform designers of both benefits and disadvantages of their usage.
***sg4.****Pattern role*—by clustering classes according to the role they play in the pattern occurrence (e.g., Concrete State, Concrete Prototype, etc.). This perspective represents the finest-grained analysis that can be performed under the considered level (i.e., class-level). It allows us to investigate if the number of violations in classes is related to specific roles or to the joint effect of all roles.

We note that the last three perspectives involve SPP classes only. This decision is based on the fact that for coupled pattern occurrences it is not possible to separate the individual influence of each pattern. Moreover, based on literature, coupled design pattern occurrences have a different effect on QAs compared to single occurrences [20,22,23].

Summarizing the above, the main contribution of our work is that *it explores the link between patterns and aspects of quality that are not evident as problems yet*. For example, classes that do not participate in design pattern instances may be more prone to the existence of performance issues (e.g., unnecessary data boxing[2] and unboxing, allocation of an object only to get its class, or inefficient use of collections). If the number of concentrated violations becomes high, it may result in a perceivable decrease of quality regarding performance. Therefore, the early identification of such issues, and their potential link to some GoF patterns, is considered important.

Another contribution is that our work increases the validity of the empirical results on the subject in terms of data source and methodological triangulation [24]. In other words, we can reach a safer conclusion by gathering data from different sources and using different methods. Approaching a problem from different perspectives is especially important from an empirical software engineering viewpoint in the sense that every method poses different threats to validity (e.g., the use of profilers provides an overhead to program execution that is difficult to filter out). Additionally, some use cases may never be executed during dynamic analysis, leading to the omission of the underlying violations, but they will show up in static analysis, as it covers the whole codebase. Therefore, if studies using different methods reach similar conclusions, the results can be more uniformly interpreted.

The remainder of this paper is organized as follows: related work is presented in Section 2, along with a discussion of the main points of differentiation of this study. In Section 3, we present the case study design, whereas its results are presented in Section 4, followed by a discussion of the findings in Section 5. Finally, we report on threats to validity and actions taken to mitigate them in Section 6, and draw the conclusions in Section 7.

[2] Boxing and unboxing refers to encapsulating data from one type into another, causing the value to be wrapped, leading in turn to an extra hop in order to access the value (by unboxing it).

## 2. Related work

In this section, we present related work that discusses the relationship between the application of design patterns and run-time QAs. We clarify that we only present studies that consider GoF design patterns; therefore, we excluded studies that use different patterns, e.g., architectural [25] or security [26] patterns. This section is organized into four sub-sections: First, we present the related work, grouped by the QAs addressed in this study, i.e., correctness (see Section 2.1), performance (see Section 2.2) and security (see Section 2.3). Next, we summarize this section and present the main points of advancement of our work (Section 2.4). Most of the related work discussed in this section were retrieved from a mapping study on GoF design patterns, by Ampatzoglou et al. [3], and on a literature survey on the impact of patterns on quality, by Ali and Elish [27].

### 2.1. Design patterns and correctness

Vokac [28] analyzed the correlation of five design patterns and correctness in a large commercial product (written in C++, with ~500KLOC). The product was investigated over weekly snapshots of the source code, during a period of three years. For each snapshot, the correctness of pattern-participating classes was measured in terms of number of defects, as collected from the issue tracking system. The results of the study suggest that Factory, Observer, Singleton, and Template Method patterns are correlated to higher defect frequency in source code. Additionally, Singleton and Observer seem to be often used in complex parts of the project (i.e., with more code, and higher defect frequency).

Ampatzoglou et al. [29] investigated the correlation between 12 design patterns and correctness. For that, they performed a case study involving 94 software projects in the game application domain. In this study, information was collected regarding bug tracking and pattern instances from each version of every software. During the analysis, each pattern was analyzed separately in order to identify correlations between the number of defects and pattern instances. The results of the study suggest that specific design patterns are related to higher defect frequency, although the presence of pattern occurrences (without examining each pattern separately) seems not to be correlated with such a frequency.

Gatrell and Counsell [19] investigated the effect of 11 design patterns on correctness by analyzing a commercial project written in C# (with ~266KLOC). For that, PP classes were manually collected and compared against NPP classes over a two-year period, correlating them with the fault history provided by the source control system, aiming at finding fault-prone classes. The results of the study suggest that PP classes are more fault-prone than NPP classes, as well as that this is related to both a higher number and the size of changes in NPP classes. Additionally, the authors characterized Adapter, Template Method and Singleton as the most fault-prone patterns.

Aversano et al. [21] investigated the relationship between correctness of pattern participants and the scattering degree of concerns[3] that communicate with them. For that, occurrences of 12 design patterns were extracted from several snapshots of three open-source projects, and the correctness was measured in terms of code defects. The results of this study suggest that patterns that induce crosscutting concerns (i.e., implemented across several classes spread along the system [30]) are correlated to a higher number of defects in their participants.

### 2.2. Design patterns and performance

Afacan [31] investigated the effect of the State design pattern on performance of a Digital Signal Processor (DSP). The author compared three implementations for a state machine: in C, C++, and C++ using the State design pattern. For that, performance was measured in terms of execution time (in clock cycles, and μs), and required memory (in 16-bit words). The results suggest that usage of the State design pattern has a negative effect on the performance of a system. However, the author also reports that the gain in architectural aspects is worth the expectedly small loss in performance.

Rudzki [32] investigated the effect of design patterns on performance. For that, two design patterns (Facade and Command) were compared as alternative solutions to each other. These patterns were used for implementing two different solutions for accessing services of business layer from a sample Java application. Their performance was measured in different deployment configurations, using four metrics: throughput, response time, number of correctly served requests, and number of requests. The results of the study suggest that, in general, Facade provided a better performance than Command. However, some results were hard to interpret due to noise in the measured values.

Chantarasathaporn and Srisa-an [33] proposed a pattern instantiation for the Factory pattern [1]. This variant consists of an energy conscious implementation of the pattern by using C# language. In order to create an instantiation for power limited systems, the authors evaluated several options that varied in terms of component structure (i.e., class or struct) and type (i.e., static or non-static). The energy consumption was measured using four metrics (obtained via profiler): User Processor Time (UPT), Privileged Processor Time (PPT), Total Processor Time (TPT) and Memory used by the specific software process. The results of the study suggest that the modified Factory Method consumes around 11% less CPU time than the regular implementation.

Sahin et al. [6] investigated the energy consumption of design patterns. For that, they considered 15 design patterns, five from each of the categories proposed by Gamma et al. [1], measuring the difference in energy consumption between two versions of the same software (before and after applying the pattern). For measuring the energy consumption, the authors used a tool created by them, which is also introduced in their work. The results of this study show that: (a) design patterns can increase or decrease the energy usage; (b) the impact in energy consumption is not necessarily similar for pattern within the same category; and (c) energy usage is unlikely to be predicted by considering design-level artefacts only.

Finally, Litke et al. [5] investigated changes in the energy consumption due to the application of three different design patterns. For that, they used a profiler for measuring: memory accesses to the instruction memory; memory accesses to the data memory; and dissipated energy within the processor core. The results of the study show that the application of design patterns does not necessarily imply a change of energy consumption.

### 2.3. Design patterns and security

To the best of our knowledge, there is a lack of empirical studies investigating security aspects of GoF design patterns; however, we were able to identify one descriptive study. Ferraz et al. [34] relate the 12 common types of security requirements proposed by Firesmith [35] to the GoF pattern categories [1]. The authors suggest that using an initial set of GoF patterns might substantially reduce the effort required to fulfill security requirements in the future. However, no empirical analysis was performed to evaluate the proposal.

A possible explanation on the lack of related work on the relationship between security and GoF patterns is the fact that GoF patterns were not originally intended to serve security requirements [36]; hence the existence of specialized solutions known as security patterns [37]. Thus, we are not interested in investigating whether the use of GoF patterns promotes security, but on the contrary, if the application of GoF patterns leads to violations of security good practices.

[3] According to Aversano et al., it is how spread, among classes, is the implementation of a concern.

**Table 1**
Overview of related work.

| #ref | Objectives | | | Empirical setting | | | | Ability to compare results | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | QA | Patterns | Approach | Validation | Projects | Level | Classes | PP vs. NPP | Granularity | SPP ≠ CPP |
| [19] | Correctness | 11[a] | Dynamic | Case study | 1 | Class | 7,439 | Yes | Pattern | No |
| [21] | Correctness | 12[b] | Dynamic | Case study | 3 | Class | ~10,000 | No | Pattern | No |
| [28] | Correctness | 5[c] | Dynamic | Case study | 1 | Class | 1,550 | No | Pattern | Yes |
| [29] | Correctness | 12[b] | Dynamic | Case study | 94 | System | ~85,000 | No | Pattern | No |
| [5] | Performance | 3[d] | Dynamic | Example | 6 | Pattern instance | ~30 | Yes | Pattern | Yes* |
| [6] | Performance | 15[e] | Dynamic | Example | 15 | Pattern instance | ~250 | Yes | Category, pattern | Yes* |
| [31] | Performance | State | Dynamic | Case study | 1 | System | 8 | Yes | Pattern | No |
| [32] | Performance | Facade, Command | Dynamic | Case study | 1 | Pattern instance | – | No | Pattern | No |
| [33] | Performance | Factory Method | Dynamic | Example | 1 | Pattern instance | ~20 | No | Pattern | Yes* |
| [34] | Security | All | None | Theoretical | 0 | None | 0 | No | Category | No |
| This | Correctness, performance and security | 12[b] | Static | Case study | 5 | Class | 12,857 | Yes | Category, pattern, role | Yes |

* Only SPP components are considered.
[a] Adapter, Builder, Command, Creator, Factory, Template Method, Proxy, Singleton, State, Strategy, Visitor.
[b] Abstract Factory, Singleton, Composite, Adapter, Command, Observer, State, Strategy, Template Method, Decorator, Prototype and Proxy.
[c] Singleton, Template Method, Decorator, Observer, Factory.
[d] Factory Method, Adapter, Observer.
[e] Abstract Factory, Builder, Factory Method, Prototype, Singleton, Bridge, Composite, Decorator, Flyweight, Proxy, Command, Mediator, Observer, Strategy, Visitor.

### 2.4. Overview of related work

The main differences of our study compared to the related work are summarized in Table 1. In particular, we compare the studies with respect to three aspects:

(a) ***Objectives***: The conceptual elements of the work, i.e., studied QAs; studied GoF patterns; and type of approach measuring QAs.
(b) ***Empirical setting***: The empirical setup of the studies, i.e., type of validation; number of used projects; level of measurement (i.e., unit from which the QA was assessed); and the number of assessed classes (a dash indicates that it was not possible to find or estimate the number of classes).
(c) ***Ability to compare results***: The elements of the analysis that are comparable to our study, i.e., whether or not PP components are compared against NPP components; granularity of the pattern investigation—i.e., category, pattern and role—whether or not there is a distinction between SPP and CPP.

Based on this overview, the advancements of this study compared to the state-of-the-art research are:

- *It investigates three run-time QAs using **static analysis***, providing evidence on their potential relationship to the application of design patterns;
- *It identifies **similarities and differences** between the results obtained by static analysis and those obtained by dynamic analysis*, increasing the validity of evidence on the subject, as well as adding to the current state of the art on analysis of run-time QAs;
- It is, to the best of our knowledge, the first study that provides ***empirical evidence*** on the *relationship between the use of GoF patterns and security*.

## 3. Case study design

This section describes the case study protocol, which was designed according to the guidelines of Runeson et al. [38] and is reported based on the Linear Analytic Structure [38].

### 3.1. Objectives and research questions

The goal of this study is described using the Goal-Question-Metrics (GQM) approach [39], as follows: "**analyze** software projects **for the purpose of** evaluating GoF design patterns **with respect to** their potential relationship with run-time quality attributes, **from the point of view of** software developers **in the context of** open source systems". Based on the goal of this study, we defined the following research questions (RQ):

**RQ$_1$**: *To what extent do run-time QAs differ between non-pattern-participating (NPP), single-pattern-participating (SPP), and coupled-pattern-participating (CPP) classes?*

***RQ$_{1.1}$***: To what extent do the aforementioned groups of classes differ regarding correctness?
***RQ$_{1.2}$***: To what extent do the aforementioned groups of classes differ regarding performance?
***RQ$_{1.3}$***: To what extent do the aforementioned groups of classes differ regarding security?

RQ$_1$ aims at exploring whether the application of GoF design patterns is related to the levels of run-time QAs. This question is important to investigate, in the sense that certain GoF patterns are using "expensive" or sometimes complex OO mechanisms, e.g., polymorphism or extensive message passing, that can potentially harm run-time QAs in favor of improving design-time ones. Additionally, while performing such an investigation, it is important to treat the two types of pattern participation (SPP and CPP) separately, because CPP classes are a special case of pattern-participation.

**RQ$_2$**: *Is the relationship between GoF patterns and run-time QAs different across categories of design patterns?*

***RQ$_{2.1}$***: Is there a difference in the levels of correctness among classes participating in patterns of different categories?
***RQ$_{2.2}$***: Is there a difference in the levels of performance among classes participating in patterns of different categories?
***RQ$_{2.3}$***: Is there a difference in the levels of security among classes participating in patterns of different categories?

$RQ_2$ aims at investigating if the different purposes that patterns serve (i.e., create objects, handle system behavior, and organize source code structure) lead to a different relation between GoF pattern application and the levels of run-time qualities. A similar question was considered in other studies such as Sahin et al. [6]. To explore this research question, we focus only on SPP classes, clustering them by category (creational, behavioral and structural). We exclude CPP classes from this RQ, since the behavior of coupled pattern cannot be safely attributed to one of the patterns participating in it.

**$RQ_3$**: *Is the relationship between GoF patterns and run-time QAs, different across design patterns?*

***$RQ_{3.1}$***: Is there a difference in the levels of correctness among classes participating in different patterns?
***$RQ_{3.2}$***: Is there a difference in the levels of performance among classes participating in different patterns?
***$RQ_{3.3}$***: Is there a difference in the levels of security among classes participating in different patterns?

$RQ_3$ aims at identifying design patterns whose classes might be more prone to violating good coding practices. Therefore, alternative solutions (non-pattern or non-GoF) may be preferred when possible [40]. Many studies (e.g., [19,21,28]), have explored similar RQs. To answer this research question, we focus only on SPP classes, clustering them by design pattern.

**$RQ_4$**: *Is the relationship between GoF patterns and the levels of run-time QAs, different across design patterns roles?*

***$RQ_{4.1}$***: Is there a difference in the levels of correctness among classes playing different roles in GoF patterns?
***$RQ_{4.2}$***: Is there a difference in the levels of performance among classes playing different roles in GoF patterns?
***$RQ_{4.3}$***: Is there a difference in the levels of security among classes playing different roles in GoF patterns?

$RQ_4$ aims at investigating the different roles that classes can play within a certain pattern (e.g., the Subject in an Observer pattern instance) to identify those that are more prone to harm specific run-time QAs. Although roles have also been explored in other studies, e.g., [20,41], we decided not to investigate the roles individually, but rather consider the similar purposes they have (e.g., Container, Containee and Client), namely meta-roles (see Section 3.3.3). We made this decision since meta-roles may lead to more exploitable results as they encompass responsibilities that may lead to more relevant investigation [20]. To explore this research question, we focus only on SPP classes, clustering them by design pattern meta-role.

### 3.2. Case selection and unit of analysis

This study is a holistic multiple-case study, in which open-source software (OSS) projects are the subjects. As unit of analysis we refer to one class of a project over a certain period of time when the number and type of patterns, in which the class participates, is stable. Based on the above, as a stable pattern status period, we refer to a set of versions in which the class did not change its participation status (i.e., participating to a specific pattern, or not participating to any pattern). Therefore, it is important to note that, in order to avoid possible bias from outliers (e.g., an outstanding good release, or a very buggy version), we consider all available versions of each system, considering the measurement of each metric in a unit of analysis as the average of all versions in the stable pattern status period.

Concluding, we consider triplets of *<class_name, pattern_participation, versions-span>* as unit of analysis. For example, suppose that class C1 starts its lifespan as a participant in a Visitor pattern, until version 6; next, it does not participate in a pattern for three versions; and next it becomes a Strategy participant for three more versions. This class would provide us with three units of analysis, as follows:

- <C1, Adapter, Ver. 1-Ver. 6>
- <C1, no-pattern, Ver. 7-Ver. 9>
- <C1, Strategy, Ver. 10-Ver. 12>

In order to select appropriate cases (i.e., subjects) for our study, we considered OSS projects from SourceForge.[4] The projects used in our analysis were required: (a) to be written in Java, due to limitations of the used tools (see Section 3.3.3); (b) to have an adequate number of versions for evolution analysis; and (c) not to be considered as "toy examples". For that, we selected the five most popular projects from SourceForge that fit our requirements, and we collected all available versions for each one of them. The selected projects, accompanied by their size and duration information, are presented in Table 2.

**Table 2**
Projects considered in the case study.

| Project name | Starting year[a] | Size[b] | NoC[c] | NoV[d] |
|---|---|---|---|---|
| Bonita BPM | 2009 | 138K | 3,994 | 45 |
| Convertigo | 2011 | 79K | 1,779 | 40 |
| Eclipse Checkstyle | 2003 | 9K | 216 | 37 |
| Hibernate | 2001 | 162K | 3,374 | 126 |
| LogicalDOC | 2008 | 46K | 818 | 31 |

[a] Year of registration according to SourceForge.
[b] Size in lines of code (of the last version).
[c] NoC = Number of classes (of the last version).
[d] NoV = Number of versions.

### 3.3. Data collection and pre-processing

In order to answer the research questions stated in Section 3.1, we extracted three sets of variables from each class of each version of the five selected projects (to later compute the units of analysis - see Section 3.3.3), as follows:

- project and class identification information;
- pattern participation information; and
- estimates on the levels of QAs.

We clarify that the third set represents assessments of the studied QAs, and, therefore, when referring to the attributes, we are in practice referring to their assessments. For that, we selected metrics that, to the best of our knowledge, are able to quantify aspects of their levels of quality (see Section 3.3.2). An overview and more details on each variable are presented in Section 3.3.3. Details on the pattern detection and run-time QAs assessment are presented in the following sections.

#### 3.3.1. Detection of design patterns occurrences

Regarding pattern detection in each version of the projects, we used a tool developed by Tsantalis et al. [42]. This tool[5] uses a Similarity Scoring Algorithm (SSA) for detecting instances of 12 patterns, namely, Adapter/Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State/Strategy, Template Method, and Visitor. By reverse engineering the system under study, this tool isolates subsystems and explores the relationship between elements in each one of them, applying the proposed SSA to detect occurrences of the aforementioned patterns [42]. The tool was already evaluated in independent studies (e.g., by Kniesel and Binun [43], and Pettersson et al. [44]), which reported positively on its performance, precision and recall rates. In short, the recall of the tool averages around 70%, varying

[4] https://sourceforge.net/.
[5] http://users.encs.concordia.ca/~nikolaos/pattern_detection.html.

between 25% and 100% in the reported benchmarks. The precision is reported to be close to 100%, mostly due to structure-based detection approach. Moreover, we manually verified the precision of the tool by checking 50 random pattern instances for each GoF pattern that is detected by this tool (i.e., over 500 instances in total—standing for 32% of the total dataset in terms of pattern instances), which were all true positives. We decided to use this tool for the following reasons:

- it covers a fair amount of design patterns that can be detected;
- it has adequate performance, as reported in Tsantalis et al. [42], also when compared to similar tools [43,44]; and
- it facilitates the pattern detection process.

Although this tool is able to detect the aforementioned patterns, it does not extract all PP classes. According to Aversano et al. [21] classes are subdivided into two categories: (a) main PP classes, comprising those that provide the structure of the pattern solution (commonly abstract classes); and (b) extended PP classes, which are subclasses of the former that extend the functionality of the pattern solution. The SSA tool only detects the main PP classes. Therefore, as we require all PPs to be identified for our study, a second tool[6] named SSA+, developed by the authors, was used to identify and extract the extended PP classes. SSA+ takes as input the output of SSA, and is able to identify 10 extra roles, based on the information provided by SSA for each pattern occurrence. The extra roles are: *Concrete Creator* and *Product*, for Factory Method pattern; *Concrete Prototype*, for Prototype pattern; *Leaf*, for Composite pattern; *Concrete Decorator* and *Concrete Component*, for Decorator pattern; *Concrete Observer*, for Observer pattern; *Concrete State/Strategy*, for State/Strategy pattern; *Concrete Class*, for Template Method pattern; and *Subject*, for Proxy pattern.

The task performed by SSA+ is deterministic, as it identifies classes that comply with a set of rules (e.g., inherit from a main PP class), and, therefore, the final version of the tool should present no faulty results, as it was thoroughly tested. In order to further validate SSA+, we manually verified the output for numerous pattern occurrences (randomly selected) of each pattern for which we detect extra roles. In all cases, SSA+ found only true positives, and no class appeared to be missing. The tool was also used in another study, in which a similar verification procedure was performed [45]. Additionally, to validate our data collection, we verified the frequency of pattern occurrences in the entire dataset against the frequencies obtained by related work [20,23]. In Table 3, we present the distribution of classes among all participation types (NPP, SPP and CPP), for the five projects, as well as the summary of all projects. All frequencies are presented considering main roles only (detected via SSA), as well as all roles (SSA+, i.e., main and extra roles). Related work has considered main roles and the frequency reported in Table 3 (SSA) is accordance to theirs [20,23,29]. Finally, one can notice that, as expected, the frequency of PP classes considering all roles (SSA + ) is higher than considering main roles only (SSA).

**Table 3**
Frequency of pattern occurrences based on SSA and SSA+.

| Project name | Number of classes | Tool | NPP | SPP | CPP |
|---|---|---|---|---|---|
| Bonita BPM | 3994 | SSA | 74.0% | 19.2% | 6.8% |
| | | SSA+ | 54.0% | 35.3% | 10.8% |
| Convertigo | 1779 | SSA | 89.0% | 8.2% | 2.8% |
| | | SSA+ | 74.6% | 16.1% | 9.3% |
| Eclipse Checkstyle | 216 | SSA | 71.8% | 19.9% | 8.3% |
| | | SSA+ | 40.7% | 44.9% | 14.4% |
| Hibernate | 3374 | SSA | 63.5% | 25.2% | 11.4% |
| | | SSA+ | 51.0% | 26.6% | 22.4% |
| LogicalDOC | 818 | SSA | 85.0% | 9.5% | 5.5% |
| | | SSA+ | 80.6% | 13.0% | 6.5% |
| Total | 10,181 | SSA | 74.0% | 18.5% | 7.5% |
| | | SSA+ | 58.5% | 27.4% | 14.1% |

The frequency is w.r.t. the last version of each project.

### 3.3.2. Assessment of run-time quality attributes

To evaluate software projects with respect to their run-time QAs, we performed static analysis by collecting the amount of several different types of violations of good coding practices. For that, we used the tool FindBugs,[7] which detects such violations and provides warnings [46]. The tool was already evaluated in independent studies (e.g., by Hovemeyer and Pugh [46] and Ayewah et al. [47]), which reported an average precision of 66% and stated that the precision can be increased by measures such as filtering bug patterns and selecting confidence levels. We also evaluated the tool in a previous study and found that its precision for the three levels of confidence (i.e., low, medium, and high) are 26.67%, 60%, and 73.33%, respectively [48]. Thus, we advised discarding violations with low level of confidence.

It is possible that FindBugs introduces noise (i.e., false positives) to the data collection. However, we also found in other studies [8,9,10,13,46], as well as in our experiments, that the violations identified by FindBugs are valuable pointers to parts of the system that need to be maintained. Moreover, we note that although part of the violations detected by FindBugs may incur bugs in the system (therefore the nomenclature “bug pattern”), we do not make this assumption. We treat them simply as violations, which can be used as indicators of quality. Other studies explored this approach to estimate quality [13,49]. In particular, Khalid et al. [13] examined violations from FindBugs, correlating them to software quality as perceived by end-users. Their results suggest that violations can be used as quality indicators, as the two were closely related in the observed population. In short, we adopt the estimation of violations identified by static analysis not as an absolute number of real bugs or faults in the system at a given moment, but as a quality indicator that can warn developers and architects to investigate a given part of the system.

In this case study, we have chosen to use FindBugs because it provides:

- a collection of over 400 bug patterns;
- adequate precision when compared to similar tools [7,46,47], which reflects on the relevance of the offered bug patterns; and
- a grouping of these bug patterns in nine high-level categories[8] that can in turn be mapped into QAs, as presented below.

In this study, to evaluate run-time QAs, we considered the first five categories (in total 246 bug patterns), as they can be mapped to the three studied QAs: *correctness*[9] (*Correctness and Multithreaded Correctness* categories), *performance*[10] (*Performance* category), and *security*[11] (*Security* and *Malicious Code* categories). The levels of quality, in each OSS version, for the three aforementioned QAs are estimated by the quantity of:

[6] https://github.com/search-rug/ssap.
[7] http://findbugs.sourceforge.net/.

[8] The categories are: Security, Correctness, Multithreaded Correctness, Performance, Malicious Code, Bad Practice, Internationalization, Experimental and Dodgy Code.
[9] An example of a correctness bug pattern is: Signature declares use of unhashable class in hashed construct (http://findbugs.sourceforge.net/bugDescriptions.html#HE_SIGNATURE_DECLARES_HASHING_OF_UNHASHABLE_CLASS).
[10] An example of a performance bug pattern is: Method allocates an object, only to get the class object (http://findbugs.sourceforge.net/bugDescriptions.html#DM_NEW_FOR_GETCLASS).
[11] An example of a security bug pattern is: HTTP cookie formed from untrusted input (http://findbugs.sourceforge.net/bugDescriptions.html#HRS_REQUEST_PARAMETER_TO_COOKIE).

- ***New violations per lines of code (LoC)***: this metric partially expresses the likelihood of a class to harm the assessed QA;
- ***Removed violations* (i.e., *not detected in comparison to the previous version) per LoC***: this metric partially expresses the likelihood of a class to benefit the assessed QA. As this metric depends on the previous existence of violations, we represent it as the percentage of removed violations compared to the total amount; and
- ***Total violations* (i.e., *total number of detected violations) per LoC***: this metric takes into account the resulting effect of both adding and removing violations.

We clarify that, concerning correctness and security, the quantities are the sum of the two categories of bug patterns that each QA is comprised of. For example, security is measured by summing the numbers from both *Security* and *Malicious Code* categories. For all three QAs a lower number of Total and New violations reflect a higher level of quality, while it is the opposite for Removed violations (i.e., the level of quality is directly proportional to the number of Removed violations).

#### 3.3.3. Collection procedure and pre-processing

The data collection phase was a two-step process. First, we collected raw data of the QAs assessment variables for every class of every version using FindBugs, as well as design pattern related variables using the SSA and SSA+ tools. All tools work on Java binary code, so we fed them with a set of .class files and recorded the outcome. For FindBugs, we used the command line version 3.0.0, for automation purposes. We configured the tool with maximum effort (i.e., enabling analysis that increases precision), and reported violations with medium or high confidence level (to improve precision, as reported in a previous study [48]) and from all urgency priorities (i.e., from least to most harmful to the system). For SSA tool, we used the command line version 4.5, also for automation purposes.

Next, we derived additional information that was needed to collect every variable for the units of analysis. For that, two tasks were performed:

***t1.*** We compiled FindBugs' information for each project. The *bug detection report of all versions* was compiled into a **history of violations**, generated by FindBugs. The number of violations from each QA was obtained by counting the rule violations with medium and high confidence from Findbugs.[12] The three aforementioned metrics (New, Removed, and Total number of violations) were calculated based on these values.

***t2.*** We mapped pattern roles to their respective meta-roles, according to the map presented in Table 4. The meta-roles are assigned to roles based on the purpose of those roles. We considered the same seven meta-roles as in our earlier work (for more details see [20]): Client, Container (a container or aggregate in a "whole-part" relationship, or the dependent class in a "simple association"), Containee (a containee or component in a "whole-part" relationship or the independent class in a "simple association"), Superclass (or abstract class), Subclass, Compound (playing two or more of the aforementioned roles), and Singleton. For example, the Subject acts as a container in the Observer pattern.

Finally, a dataset was created with the information of all variables for each unit of analysis. This dataset was recorded as a table into a spreadsheet, in which each line corresponded to one class of one project in a certain version range. Summarizing, the full list of variables, together with their description, is presented in Table 5. The final dataset is created as described above to facilitate the identification of coupled patterns (by detecting duplication of classes in different patterns on the same version), and merging of tuples to build data subsets for answering each RQ (e.g., merging tuples of same pattern category and version, to answer $RQ_2$).

[12] We had analyzed and validated FindBugs, in a previous work [13], regarding its confidence levels, reporting that precision can be improved by excluding bugs with low confidence level.

**Table 4**
Mapping of pattern roles to meta-roles.

| Pattern type | Pattern role | Meta-role |
|---|---|---|
| Adapter/Command | Adaptee/Receiver | Containee |
| | Adapter/Concrete Command | Container |
| Composite | Component | Superclass |
| | Composite | Compound |
| | Leaf | Subclass |
| Decorator | Component | Superclass |
| | Concrete component | Subclass |
| | Concrete decorator | Subclass |
| | Decorator | Compound |
| Factory method | Concrete creator | Compound |
| | Creator | Superclass |
| | Product | Containee |
| Observer | Concrete observer | Subclass |
| | Observer | Compound |
| | Subject | Container |
| Prototype | Client | Client |
| | Concrete prototype | Subclass |
| | Prototype | Superclass |
| Proxy | Proxy | Compound |
| | RealSubject | Subclass |
| | Subject | Superclass |
| Singleton | Singleton | Singleton |
| State/Strategy | Concrete State/Strategy | Subclass |
| | Context | Client |
| | State/Strategy | Superclass |
| Template Method | Abstract class | Superclass |
| | Concrete class | Subclass |

**Table 5**
List of collected variables.

| Variable | Description | Tool |
|---|---|---|
| [V1] | Source project of the class | – |
| [V2] | Class full name (package + class name) | – |
| [V3] | Versions of the project considered in the unit of analysis | – |
| [V4] | Class type (i.e., NPP, SPP, or CPP) | SSA & SSA+ |
| [V5] | Name of the category (i.e., behavioral, creational, structural) containing the pattern in which the class participated. | |
| [V6] | Name of the pattern in which the class participated | |
| [V7] | Name of the meta-role that the class played in the pattern | |
| [V8-V10] | Violation metrics for correctness | FindBugs |
| [V11-V13] | Violation metrics for performance | |
| [V14-V16] | Violation metrics for security | |

### 3.4. Data analysis

During this phase, we analyzed the previously described variables (V1–V16) to investigate the relationship between the use of design patterns and the level of run-time QAs. We clarify that, unless specified, when referring to a relationship with a certain QA, we imply the scores of all metrics of the QA. The analysis of the collected data is split in four steps, each aiming at answering each of the four RQs. In each step, a data subset was derived from the final dataset (see Section 3.3.3), and further analyzed as follows:

***s1.*** *Verify relationship between NPP, SPP, and CPP classes for each QA.* In this step, we derived a data subset consisting of classes that are related to only one of the three participation types during their

entire version-span (within the collected data). By avoiding change of participation, we aimed at mitigating bias caused by the joint effect of multiple types of pattern participation. Therefore, independent sample *t*-tests are performed in order to investigate differences in the level of QAs among the types of participation. This step was divided into four sub-steps.

a. *Select relevant tuples*. We selected only classes that have been only NPP, SPP, or CPP during their lifetime (i.e., classes that did not change their participation).
b. *Remove duplicated entries*. As CPP classes appear in more than one pattern occurrence, it is necessary to avoid accounting for duplicates of classes. For example, a class that has initially been a Singleton and in a later version part of an Adapter occurrence would produce two units of analysis as a pattern participant, which have identical number of violations (because they regard the same class). These two units of analysis would be merged into a single CPP entry (to avoid duplicated entries).
c. *Compute units of analysis*. Tuples concerning the same class in the different versions are now united by calculating the average for each metric, resulting in the data subset to be analyzed.
d. *Calculate differences in levels of QAs*. We performed independent sample *t*-tests using pattern participation (V4) as a grouping variable, and the number of violations (V8-V16) as test variables. We clarify that, despite analyzing three groups (NPP, SPP, and CPP), we did not perform analysis of variance, since we intend to look into every pairwise comparison.

***s2.*** *Verify difference among design pattern categories for each metric of each QA*. In this step, we derived a data subset consisting of classes that were SPP during their entire version-span, for the same reasons described in step 1. In a similar four-sub-steps process, the units of analysis were initially selected and clustered into three clusters, one for each pattern category (structural, behavioral, and creational); afterwards, duplicates were removed, i.e., tuples concerning same class, version, and pattern category; then, the units of analysis were computed by merging tuples concerning same classes and different versions; finally, a one-way analysis of variance (ANOVA) was performed for each metric of each QA, i.e., [V8 - V16], using pattern category (V5) as grouping variable.

***s3.*** *Verify difference among design patterns for each metric of each QA*. In this step, we derived a data subset consisting of classes that were SPP during their entire version-span, for the same reasons described in step 1. Similar to step 2, we performed ANOVA for each metric of each QA, but using the pattern (V6) as grouping variable. Removal of duplicates and computation of units of analysis were also similarly performed (considering pattern in the procedure).

***s4.*** *Verify difference among design pattern meta-roles for each metric of each QA*. In this step we derived a data subset consisting of classes that were SPP during their entire version-span, for the same reasons described in step 1. Next, we clustered the dataset into seven groups, one for each meta-role (V7). Finally, we performed independent sample *t*-tests between pairs of meta-roles for each metric of each QA, i.e., [V8–V16].

Summarizing the procedure for answering the RQs, Table 6 presents the mapping between each RQ, the used variables, as well as the step of the analysis in which each RQ is answered, along with the used presentation methods.

## 4. Results

In this section, we present the results of the case study, highlighting the most important observations based on the acquired data. Each RQ is addressed separately, presenting an overview of the considered data subset, as well as the statistical analysis over the data (see Section 3.4). Before presenting the results, we clarify that the metric on Removed violations was not statistically evaluated in all RQs due to the low number of classes that had removed violations. However, this is not an important issue since the metric on Total number of violations also reflects the effects of the removed ones (see Section 3.3.2).

### 4.1. Comparison between SPP, PPC, and NPP classes ($RQ_1$)

The descriptive statistics (i.e., number of units of analysis, mean number of violations per 10 KLOC, and standard deviation) of the data subset built for answering $RQ_1$ (see step 1 of Section 3.4) are presented in Table 7. We clarify that due to the nature of the data (i.e., violations) it is expected that most of the classes in the dataset do not present violations. Thus, metrics such as median and mode would not be descriptive for our dataset and, for that reason, are not included in Table 7. For each metric, we highlight the type of pattern participation with the lowest amount of *New* and *Total* violations, e.g., for *New security* violations SPP classes had the lowest average number of violations (i.e., 0.72).

Based on the descriptive statistics, we created the radar chart depicted on Fig. 1 to better visualize and compare the three types of pattern participation. To create the charts, we normalized the mean for each metric (New violations, Removed violations, and Total number of violations) as the ratio over the best result. Therefore, the best result has score 1, and the other two scores are equal or less than 1. We note that for New and Total violations, the ratio is inverse, i.e., more violations are implied by scores closer to 0, and best results (i.e., fewer violations) are denoted with scores close to 1. Regarding Removed violations high scores imply best results (more violations are corrected), whereas low scores refer to cases in which only few violations are resolved. In each radar chart, we have created three lines: (a) a continuous red line for NPP classes; (b) a dotted blue line for SPP classes; and (c) a dashed green line for CPP classes. To interpret these charts one needs to check which type of pattern participation has a value equal to 1, and then compare to the rest. For instance, for New violations we can observe that SPP classes exhibit the best results for Security and Correctness, whereas CPP for Performance.

In order to verify the previously presented differences, we carried out statistical tests to compare all obtained means. For that, we performed independent *t*-tests between every two types of participation (i.e., SPP vs. NPP, SPP vs. CPP, and NPP vs. CPP) for each metric of each QA (V8–V16). In Table 8 we present the results of the tests that are statistically significant. For example, the difference between NPP and SPP for New security bugs presented in Fig. 1 (top on the left radar chart) is statistically significant. From the results, the following observations can be highlighted.

*Non-pattern-participating (NPP) classes are likely to underperform when compared against pattern-participating (PP) classes (both SPP and CPP)*. For nine out of the 18 possible comparisons (between NPP and the other two class types), NPP classes exhibit a statistically significant larger number of violations than classes participating in patterns. *At the same time, there is no strong statistical evidence of the difference between SPP and CPP classes*. However, it should be pointed out that one difference between SPP and CPP classes has been found to be statistically significant (i.e., Total amount of performance violations), providing an indication in favor of SPP classes.

### 4.2. Comparison between pattern categories $RQ_2$

To compare the different pattern categories (i.e., Behavioral, Creational and Structural), we considered the data subset as described in step 2 of the data analysis (see Section 3.4), comprising units of analysis limited to SPP classes. The descriptive statistics for this dataset are presented in Table 9. Similar to $RQ_1$, we present the number of units of analysis, mean number of violations per 10 KLOC, and standard deviation for each category (i.e., in this case, of each pattern category).

**Table 6**
Mapping of RQs to variables, steps, and presentation.

| Research Question | | Used Variables | Step | Presentation Method |
|---|---|---|---|---|
| **RQ1** (relationship between PP and NPP classes) | **RQ1.1** (regarding correctness) | [V2-V4]: unit of analysis<br>[V8-V10]: correctness metrics | 1 | Independent sample t-test |
| | **RQ1.2** (regarding performance) | [V2-V4]: unit of analysis<br>[V11-V13]: performance metrics | | |
| | **RQ1.3** (regarding security) | [V2-V4]: unit of analysis<br>[V14-V16]: security metrics | | |
| **RQ2** (difference among design pattern categories) | **RQ2.1** (regarding correctness) | [V5]: pattern category<br>[V8-V10]: correctness metrics | 2 | ANOVA |
| | **RQ2.2** (regarding performance) | [V5]: pattern category<br>[V11-V13]: performance metrics | | |
| | **RQ2.3** (regarding security) | [V5]: pattern category<br>[V14-V16]: security metrics | | |
| **RQ3** (difference among design patterns) | **RQ3.1** (regarding correctness) | [V6]: pattern<br>[V8-V10]: correctness metrics | 3 | ANOVA |
| | **RQ3.2** (regarding performance) | [V6]: pattern<br>[V11-V13]: performance metrics | | |
| | **RQ3.3** (regarding security) | [V6]: pattern<br>[V14-V16]: security metrics | | |
| **RQ4** (difference among design pattern meta-roles) | **RQ4.1** (regarding correctness) | [V7]: meta-role<br>[V8-V10]: correctness metrics | 4 | Independent sample t-test |
| | **RQ4.2** (regarding performance) | [V7]: meta-role<br>[V11-V13]: performance metrics | | |
| | **RQ4.3** (regarding security) | [V7]: meta-role<br>[V14-V16]: security metrics | | |

**Table 7**
Descriptive statistics of the data subset for $RQ_1$.

| VT | QA | Class Type | N | Mean | SD |
|---|---|---|---|---|---|
| New | Security | NPP | 7,961 | 1.39 | 16.30 |
| | | SPP | 3,249 | 0.72 | 8.58 |
| | | CPP | 1,647 | 0.86 | 11.50 |
| | Correctness | NPP | 7,961 | 0.56 | 11.50 |
| | | SPP | 3,249 | 0.18 | 2.82 |
| | | CPP | 1,632 | 0.45 | 9.99 |
| | Performance | NPP | 7,961 | 0.74 | 10.90 |
| | | SPP | 3,249 | 0.44 | 5.70 |
| | | CPP | 1,647 | 0.24 | 2.26 |
| Total | Security | NPP | 7,961 | 19.90 | 137.00 |
| | | SPP | 3,249 | 11.60 | 81.90 |
| | | CPP | 1,647 | 11.20 | 90.80 |
| | Correctness | NPP | 7,961 | 5.35 | 50.20 |
| | | SPP | 3249 | 2.92 | 35.10 |
| | | CPP | 1,643 | 3.65 | 29.30 |
| | Performance | NPP | 7,961 | 7.44 | 57.90 |
| | | SPP | 3,249 | 4.52 | 43.70 |
| | | CPP | 1,647 | 8.79 | 55.60 |
| Removed | Security | NPP | 387 | 2.94 | 18.60 |
| | | SPP | 131 | 19.10 | 147.00 |
| | | CPP | 86 | 13.20 | 69.20 |
| | Correctness | NPP | 197 | 13.20 | 47.10 |
| | | SPP | 59 | 9.22 | 37.70 |
| | | CPP | 87 | 2.34 | 7.90 |
| | Performance | NPP | 259 | 10.30 | 53.20 |
| | | SPP | 71 | 11.70 | 43.00 |
| | | CPP | 103 | 4.84 | 18.00 |

To better visualize the descriptive data, we created a radar chart to ease the comparison of mean values (see Fig. 2). Similar to Fig. 1, the best result has score 1, and the other two scores are equal or less than 1.

To verify the differences presented by the descriptive statistics, we performed a two-step statistical analysis for each metric of each QA. First, we carried out an analysis of variance (ANOVA) to identify the existence of differences between the three pattern categories for the given metric. If a difference was detected, then we applied a post-hoc test to recognize which pattern categories differentiate from each other. For the post-hoc test, we used the Bonferroni correction due to its ability to control Type I error (i.e., find a relationship that in fact does not exist), and its suitability for a small number of categories [50]. In Table 10, we present the tests that revealed a statistically significant difference. From the results, we highlight the following observations.

*Classes that participate in Creational patterns are likely to have fewer violations of security and performance coding practices than those participating in Structural and Behavioral patterns.* Based on the statistically significant differences presented in Table 10, we notice that comparisons of Creational patterns regarding security are mostly valid. With regards to performance violation, the only statistically significant different provides an indication that Creational patterns may be less prone to such violations. This is an expected result as Creational patterns tend to be simpler than patterns of the other two categories. Moreover, this finding corroborates with findings of related work.

### 4.3. Comparison between patterns ($RQ_3$)

For comparing the 12 patterns considered in this study, we focused on SPP classes as units of analysis. However, this time we clustered them by pattern, so as to be able to explore the differences between such types for each QA metric. Similar to the previous RQs we present the descriptive statistics of this data subset (see Table 11). This table shows the number of units of analysis for each pattern, as well as the mean and standard deviation for each QA metric. It is important to highlight that we excluded three patterns from the analysis (Composite, Observer and Proxy) due to the small number of available SPP classes (1, 15 and 11, respectively). Additionally, we did not consider results with mean violations equal to zero, although they are shown in Table 11 (for clarity purposes). Such a mean indicates that we identified no violations in the SPP classes and, although we expect this number to be small, we cannot predict it with enough confidence.

To better visualize the difference between the means and, most importantly, how the patterns are ordered, we plotted each metric in different rows of Fig. 3. Each row presents the mean number of violations, normalized by the best score, i.e., the best score received 1 (plotted on the rightmost side) and all others a ratio of this value factor (patterns with score 0 are not plotted). The patterns are identified by symbol (see legend of the figure), and each pattern category (e.g., Creational) has a different filling color and graphic pattern. It is important to highlight that the Adapter/Command patterns are detected

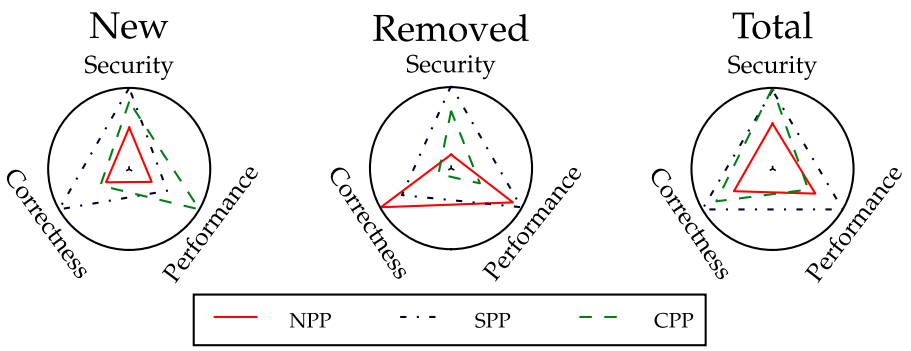


**Fig. 1.** Relationship between pattern participation type and QAs.

jointly by SSA+ (due to design similarities), and that they are from different categories (Structural and Behavioral respectively). This is considered in both charts and analysis of the results.

Similar to the previous RQ, we used ANOVA to verify the differences among the means. However, for the post-hoc test, we selected Games-Howell because the number of groups (i.e., patterns) was inadequate for using the Bonferroni correction. Table 12 presents the statistically significant results. From the results, we highlight the following observations.

*SPP classes of a Factory Method instance are less prone to violations.* This finding is supported by the facts that six comparisons involving Factory Method are statistically significant (see Table 12). This result is not surprising because this is a Creational pattern, which has previously shown to be the least vulnerable one. *Ordering of patterns tends to reflect the ordering of pattern categories* (see Section 4.2), as it would be expected. By looking at Fig. 3, one can see that Creational patterns are predominantly ranked among those with best scores, whereas Behavioral and Structural patterns interchangeably rank among those with worst scores. For example, Factory Method, which is a Creational pattern, achieves the highest scores, while Template Method (a Behavioral pattern) and Decorator (a Structural pattern) have the worst scores. However, no clear ordering appears within the patterns of a category (expect for Factory Method). This might be evidence that the amount of violations might be more related to the type of responsibility (identified by a pattern category) rather than to specific patterns.

### 4.4. Comparison between pattern roles ($RQ_4$)

As shown in Table 14, there are 30 pattern roles distributed among the 12 patterns considered in this study. To study these roles, we decided to consider the meta-roles (see Table 14) rather than the roles themselves, as we expect the amount of violations to be related to the type of responsibility a role has. This reduces the number of groups to be analyzed to the seven meta-roles presented in Table 14.

By focusing the investigation to the meta-roles that classes play in a pattern instance, we analyze the relationship between the types of roles these classes have. Therefore, for $RQ_4$ we investigated only the combinations of roles that participate within the same pattern (i.e., meta-roles that collaborate to provide a specific pattern solution). For example, in the instance of a Prototype pattern, classes of the following meta-roles are present: Client, Superclass and Subclass. The

**Table 9**
Descriptive statistics of the data subset for $RQ_2$.

| VT | QA | Pattern Category | N | Mean | SD |
|---|---|---|---|---|---|
| New | Security | Behavioral | 1,830 | 0.87 | 7.37 |
| | | Creational | 1,082 | 0.40 | 4.10 |
| | | Structural | 50 | 8.08 | 49.60 |
| | Correctness | Behavioral | 1,830 | 0.25 | 3.40 |
| | | Creational | 1,082 | 0.10 | 2.11 |
| | | Structural | 50 | 0.09 | 0.61 |
| | Performance | Behavioral | 1,830 | 0.63 | 6.93 |
| | | Creational | 1,082 | 0.23 | 3.70 |
| | | Structural | 50 | 0.30 | 1.60 |
| Total | Security | Behavioral | 1,830 | 14.90 | 92.30 |
| | | Creational | 1,082 | 7.96 | 67.70 |
| | | Structural | 50 | 42.20 | 161.00 |
| | Correctness | Behavioral | 1,830 | 4.90 | 51.60 |
| | | Creational | 1,082 | 1.17 | 15.80 |
| | | Structural | 50 | 3.30 | 23.30 |
| | Performance | Behavioral | 1,829 | 7.12 | 56.50 |
| | | Creational | 1,082 | 2.58 | 33.50 |
| | | Structural | 50 | 5.66 | 34.10 |

**Table 8**
Statistically significant results from the investigation of $RQ_1$.

| VT | QA | Test | Eq. of Variance | | Independent T-test | | |
|---|---|---|---|---|---|---|---|
| | | | F | Sig. | t | df | Sig(2-tailed) |
| New | Security | NPP vs. SPP | 19.13 | < 0.01 | 2.87 | 10,523.31 | < 0.01 |
| | Correctness | NPP vs. SPP | 12.96 | < 0.01 | 2.70 | 9,938.51 | < 0.01 |
| | Performance | NPP vs. SPP | 8.72 | < 0.01 | 1.93 | 10,576.89 | 0.05 |
| | | NPP vs. CPP | 13.92 | < 0.01 | 3.77 | 9,606.00 | < 0.01 |
| Total | Security | NPP vs. SPP | 40.61 | < 0.01 | 3.98 | 9,729.72 | < 0.01 |
| | | NPP vs. CPP | 23.97 | < 0.01 | 3.20 | 3,404.78 | < 0.01 |
| | Correctness | NPP vs. SPP | 24.81 | < 0.01 | 2.91 | 8,510.88 | < 0.01 |
| | Performance | NPP vs. SPP | 25.71 | < 0.01 | 2.91 | 7,912.10 | < 0.01 |
| | | SPP vs. CPP | 30.02 | < 0.01 | -2.72 | 2,703.67 | < 0.01 |
| Removed | Correctness | NPP vs. SPP | 23.97 | < 0.01 | 3.15 | 219.77 | < 0.01 |

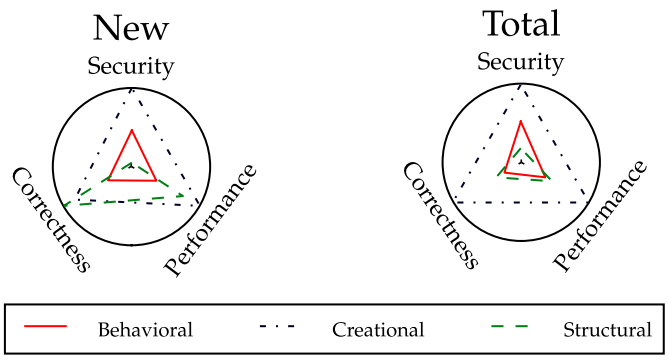


**Fig. 2.** Relationship between pattern categories and QAs.

**Table 10**
Statistically significant results from the investigation of $RQ_2$.

| VT | QA | ANOVA | | Post-hoc test (Games-Howell) | |
|---|---|---|---|---|---|
| | | F | Sig. | Test | Sig. |
| New | Security | 5.92 | < 0.01 | Adapter/Command vs. Prototype | 0.04 |
| | | | | Factory Method vs. Template Method | 0.03 |
| | | | | Singleton vs. State/Strategy | 0.059 |
| Total | Security | 3.72 | < 0.01 | Adapter-Command vs. Factory Method | 0.02 |
| | | | | Factory Method vs. Singleton | 0.04 |
| | | | | Factory Method vs. State/Strategy | 0.02 |
| | | | | Factory Method vs. Template Method | < 0.01 |
| | | | | Prototype vs. Template Method | < 0.01 |
| | Performance | 4.21 | < 0.01 | Adapter/Command vs. Template Method | 0.04 |
| | | | | Factory Method vs. Template Method | 0.04 |
| | | | | Singleton vs. Template Method | 0.02 |

investigation of these meta-roles was performed in pairs, as follows: Client vs. Superclass; Client vs. Subclass; and Superclass vs. Subclass. Considering all patterns, we derived a total of 19 pairs; we excluded the Singleton meta-role because it involves one class only.

The descriptive statistics of the analyzed meta-roles are presented in Table 13, showing the number of units of analysis, mean violations per 10 KLOC and standard deviation for each QA metric. In contrast to the previous RQs, we do not highlight the best means, as the comparisons are at the level of pairs of meta-roles rather than overall. To better visualize the comparison between the means, we created eight plots, grouping the means by metric and type of violation. Fig. 4 presents these charts; they are read similarly to Fig. 3 and colors represent sets of related meta-roles (e.g., Subclass and Superclass).

For statistically analyzing the difference between meta-roles in each pair, we performed independent sample *t*-tests. The tests that showed statistically significant difference are presented in Table 14. Based on the results, the most important observation is that *more generic meta-roles* (i.e., *Container and Superclass) are less prone to violations than less generic meta-roles.* This is an intuitive result because more abstract elements of a design usually have less complex logic, being supposedly simpler to implement. Additionally, there are several statistically significant differences that show the clear difference between the meta-roles.

## 5. Discussion

In this section, we discuss the main outcomes of the study, providing more details on their interpretation, as well as implications for researchers and practitioners. Comparison to related work is also presented, when applicable.

**Table 11**
Descriptive statistics of the data subset for $RQ_3$.

| VT | QA | Pattern | N | Mean | SD |
|---|---|---|---|---|---|
| New | Security | Adapter/Command | 371 | 0.98 | 6.22 |
| | | Decorator | 38 | 10.10 | 56.80 |
| | | Factory Method | 602 | 0.20 | 3.30 |
| | | Prototype | 45 | 0.00 | 0.00 |
| | | Singleton | 435 | 0.72 | 5.17 |
| | | State/Strategy | 1,020 | 0.67 | 7.31 |
| | | Template Method | 797 | 1.13 | 7.51 |
| | Correctness | Adapter/Command | 371 | 0.17 | 1.24 |
| | | Decorator | 38 | 0.11 | 0.70 |
| | | Factory Method | 602 | 0.00 | 0.06 |
| | | Prototype | 45 | 0.00 | 0.00 |
| | | Singleton | 435 | 0.25 | 3.32 |
| | | State/Strategy | 1,020 | 0.22 | 2.02 |
| | | Template Method | 797 | 0.29 | 4.62 |
| | Performance | Adapter/Command | 371 | 0.23 | 3.31 |
| | | Decorator | 38 | 0.40 | 1.83 |
| | | Factory Method | 602 | 0.12 | 2.67 |
| | | Prototype | 45 | 0.26 | 1.24 |
| | | Singleton | 435 | 0.37 | 4.90 |
| | | State/Strategy | 1,020 | 0.52 | 6.46 |
| | | Template Method | 797 | 0.79 | 7.54 |
| Total | Security | Adapter/Command | 371 | 16.40 | 81.50 |
| | | Decorator | 38 | 50.40 | 182.00 |
| | | Factory Method | 602 | 1.94 | 30.10 |
| | | Prototype | 45 | 3.28 | 16.00 |
| | | Singleton | 435 | 16.80 | 100.00 |
| | | State/Strategy | 1,020 | 12.20 | 90.00 |
| | | Template Method | 797 | 18.60 | 95.80 |
| | Correctness | Adapter/Command | 371 | 3.20 | 20.80 |
| | | Decorator | 38 | 4.34 | 26.80 |
| | | Factory Method | 602 | 0.11 | 1.95 |
| | | Prototype | 45 | 3.34 | 22.40 |
| | | Singleton | 435 | 2.42 | 23.70 |
| | | State/Strategy | 1,020 | 3.52 | 38.60 |
| | | Template Method | 797 | 6.74 | 64.80 |
| | Performance | Adapter/Command | 371 | 2.75 | 19.30 |
| | | Decorator | 38 | 7.45 | 39.10 |
| | | Factory Method | 602 | 1.96 | 39.10 |
| | | Prototype | 45 | 14.80 | 56.10 |
| | | Singleton | 435 | 2.18 | 18.80 |
| | | State/Strategy | 1,019 | 3.78 | 35.80 |
| | | Template Method | 797 | 11.50 | 75.10 |

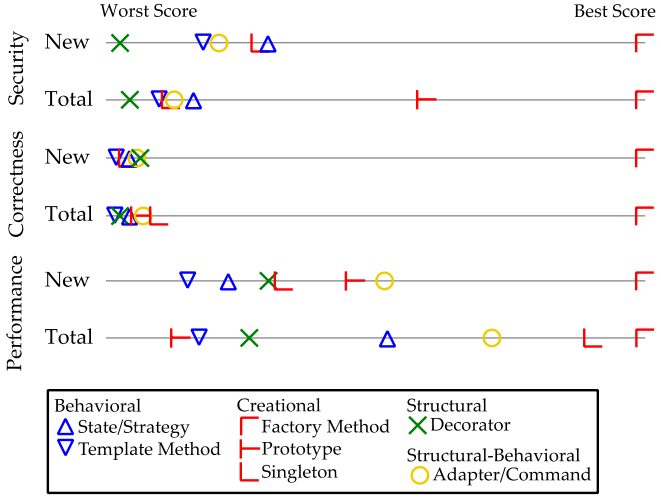


**Fig. 3.** Relationship between patterns and QAs.

**Table 12**
Statistically significant results from the investigation of $RQ_3$.

| VT | QA | ANOVA | | Post-hoc test (Games-Howell) | |
|---|---|---|---|---|---|
| | | F | Sig. | Test | Sig. |
| New | Security | 5.92 | < 0.01 | Adapter/Command vs. Prototype | 0.04 |
| | | | | Factory Method vs. Template Method | 0.03 |
| | | | | Singleton vs. State/Strategy | 0.059 |
| Total | Security | 3.72 | < 0.01 | Adapter-Command vs. Factory Method | 0.02 |
| | | | | Factory Method vs. Singleton | 0.04 |
| | | | | Factory Method vs. State/Strategy | 0.02 |
| | | | | Factory Method vs. Template Method | < 0.01 |
| | | | | Prototype vs. Template Method | < 0.01 |
| | Performance | 4.21 | < 0.01 | Adapter/Command vs. Template Method | 0.04 |
| | | | | Factory Method vs. Template Method | 0.04 |
| | | | | Singleton vs. Template Method | 0.02 |

**Table 13**
Descriptive statistics of the data subset for $RQ_4$.

| VT | QA | Meta-role | N | Mean | SD |
|---|---|---|---|---|---|
| New | Security | Client | 461 | 1.26 | 10.50 |
| | | Compound | 203 | 2.43 | 25.30 |
| | | Containee | 470 | 0.77 | 5.54 |
| | | Container | 152 | 0.04 | 0.53 |
| | | Subclass | 1,382 | 0.90 | 7.05 |
| | | Superclass | 459 | 0.11 | 1.05 |
| | Correctness | Client | 461 | 0.34 | 2.56 |
| | | Compound | 203 | 0.01 | 0.10 |
| | | Containee | 470 | 0.10 | 0.98 |
| | | Container | 152 | 0.09 | 0.88 |
| | | Subclass | 1,382 | 0.19 | 3.51 |
| | | Superclass | 459 | 0.13 | 1.71 |
| | Performance | Client | 461 | 0.68 | 6.43 |
| | | Compound | 203 | 0.32 | 4.58 |
| | | Containee | 470 | 0.17 | 2.94 |
| | | Container | 152 | 0.06 | 0.54 |
| | | Subclass | 1,382 | 0.56 | 5.93 |
| | | Superclass | 459 | 0.41 | 7.13 |
| Total | Security | Client | 461 | 15.40 | 77.20 |
| | | Compound | 203 | 13.00 | 93.70 |
| | | Containee | 470 | 12.80 | 72.40 |
| | | Container | 152 | 1.36 | 16.70 |
| | | Subclass | 1,382 | 17.40 | 103.00 |
| | | Superclass | 459 | 2.54 | 24.20 |
| | Correctness | Client | 461 | 5.29 | 41.70 |
| | | Compound | 203 | 0.32 | 3.35 |
| | | Containee | 470 | 2.13 | 17.80 |
| | | Container | 152 | 1.22 | 9.28 |
| | | Subclass | 1,382 | 4.40 | 49.90 |
| | | Superclass | 459 | 2.16 | 39.20 |
| | Performance | Client | 461 | 7.44 | 51.50 |
| | | Compound | 203 | 4.55 | 64.80 |
| | | Containee | 470 | 2.29 | 19.60 |
| | | Container | 152 | 1.32 | 12.60 |
| | | Subclass | 1,381 | 7.61 | 58.30 |
| | | Superclass | 459 | 2.01 | 25.80 |

### 5.1. Interpretation of results

Firstly, while analyzing the results of $RQ_1$, one can notice that NPP classes are likely to present more violations, regarding run-time QAs, than PP classes (i.e., SPP and CPP). Additionally, SPP classes are more likely to have fewer violations than CPP (we observed an average of 22%). A possible explanation is that PP classes are easier to understand as the patterns also serve as an explicit design documentation and as common language among the team members [51,52]. Therefore, it is expected to be easier to understand and maintain a piece of source code when it is using a pattern. On the other hand, adding an extra responsibility to a class (i.e., by making it participate in more than one pattern) might decrease the readability and understandability of the code.

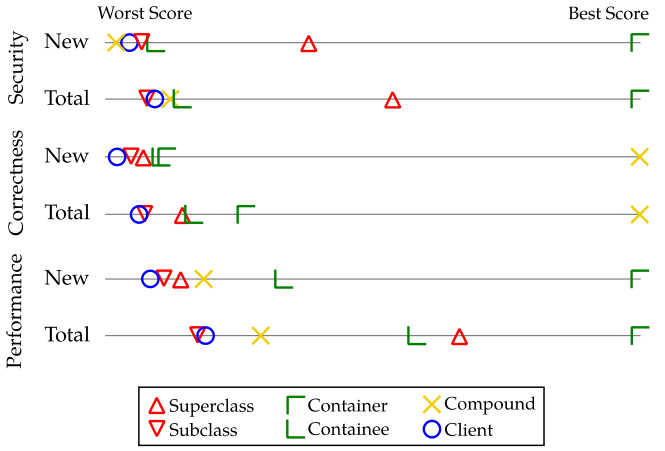


**Fig. 4.** Relationship between meta-roles and QAs.

Concerning security, our findings comply with the results of the literature. For example, Ferraz et al. [34] suggest that GoF design patterns support implementing security requirements. Regarding correctness, Gatrell and Counsell [19] found that PP classes are more fault-prone than NPP classes. Conversely, we observed that PP classes exhibited fewer violations than NPP classes. However, Gatrell and Counsell also observed that their finding was attributed mainly to a tendency of PP classes to be more change-prone. Thus, a normalization of the results could have shown a finding similar to ours, in which the analysis procedure included normalization. Moreover, Ampatzoglou et al. [29], found the overall number of design pattern instances not to be correlated with defect frequency. Run-time defects observed for PP classes are probably related to the complexity of the requirements that pattern instances are involved in, since design patterns are expected to be placed in design hot spots.

Concerning performance, related work have found that PP classes perform worse at times [5,6,31], whereas we observed that PP classes display fewer violations. However, related work also observed that pattern instances could also show improved performance compared to alternative (non-pattern) solutions [6], also suggesting that the usage of design pattern do not necessarily result in change of run-time performance [5]. The energy consumption and/or CPU usage of PP classes can be higher than that of NPP classes because patterns rely on certain object-oriented (OO mechanisms (e.g., polymorphism) that have higher computational cost, but such drawback is not always observed [53]. To further study this matter, in a previous study [53], we investigated parameters that can influence the efficiency of patterns solutions compared against alternative (non-pattern) solutions. We found that the run-time benefits of a pattern can be associated with its application in more appropriate scenarios, e.g., when the implementation logic is complex in terms of size or messaging. The appropriate use can greatly reduce the overhead of OO mechanisms.

Following our RQs, by analyzing the findings of $RQ_2$ one can observe that Creational patterns tend to have the lowest number of violations in most cases. An explanation could be that Creational patterns are implemented with a simple source code structure. A simple implementation naturally supports better understanding and readability of the source code, which in turn would explain the existence of fewer violations. In contrast, Structural patterns had the highest frequency of violations. A possible explanation is that Structural patterns are commonly used to organize complex concepts in an OO design, aiming at reducing the accidental complexity of the software [54] (i.e., the

**Table 14**
Statistically significant results from the investigation of $RQ_4$.

| VT | QA | Test | Eq. of Variance | | Independent T-test | | |
|---|---|---|---|---|---|---|---|
| | | | F | Sig. | t | df | Sig(2-tailed) |
| New | Security | Containee vs. Container | 10.56 | < 0.01 | 2.82 | 494.24 | 0.01 |
| | | Containee vs. Superclass | 25.14 | < 0.01 | 2.54 | 503.48 | 0.01 |
| | | Client vs. Superclass | 21.14 | < 0.01 | 2.34 | 469.33 | 0.02 |
| | | Container vs. Subclass | 8.97 | < 0.01 | -4.42 | 1,489.28 | < 0.01 |
| | | Subclass vs. Superclass | 22.63 | < 0.01 | 4.03 | 1,551.12 | < 0.01 |
| | Correctness | Compound vs. Containee | 7.96 | 0.01 | -2.12 | 491.92 | 0.04 |
| | Performance | Container vs. Subclass | 4.28 | 0.04 | -3.03 | 1,519.27 | < 0.01 |
| Total | Security | Containee vs. Container | 15.33 | < 0.01 | 3.18 | 586.69 | < 0.01 |
| | | Containee vs. Superclass | 33.82 | < 0.01 | 2.91 | 574.87 | < 0.01 |
| | | Client vs. Superclass | 45.97 | < 0.01 | 3.41 | 550.09 | < 0.01 |
| | | Container vs. Subclass | 14.77 | < 0.01 | -5.21 | 1,390.06 | < 0.01 |
| | | Subclass vs. Superclass | 37.49 | < 0.01 | 4.97 | 1,735.38 | < 0.01 |
| | Correctness | Compound vs. Subclass | 5.40 | 0.02 | -3.00 | 1457.75 | < 0.01 |
| | | Compound vs. Containee | 8.39 | < 0.01 | -2.13 | 540.98 | 0.03 |
| | Performance | Client vs. Superclass | 16.05 | < 0.01 | 2.03 | 677.47 | 0.04 |
| | | Container vs. Subclass | 7.04 | 0.01 | -3.36 | 1060.25 | < 0.01 |
| | | Subclass vs. Superclass | 15.76 | < 0.01 | 2.84 | 1704.70 | 0.01 |

complexity imposed by the designer and not the functionality responsibility). However, even if they decrease the accidental complexity, the essential complexity (i.e., the complexity inherent to the implemented functionality) of these components is still high, leading to more violations when compared to simpler designs (e.g., Creational patterns). We also noticed that the most recurrent violations are common among all pattern categories (see Table XVII on Supplementary Appendix A), and are similar to those regarding all SPP classes (see Table XVI in Supplementary Appendix A).

We note that the findings of this study are based on violations concerning run-time qualities derived by a static analysis tool. To facilitate the comparison of observations on the difference between PP and NPP classes and on the violations exhibited by Creational patterns, we summarize in Table 15 our key findings (static analysis column) versus findings derived by dynamic analysis in previous studies [5,6,19,29,31,53]. This summary indicates that the results from static analysis are to a large extent aligned with those from dynamic analysis, with few exceptions as discussed in the preceding paragraphs.

Findings regarding $RQ_3$ suggest that the ordering of the patterns (from best to worst score) tends to follow the ordering of the findings from the pattern categories ($RQ_2$). This may suggest that, regardless of the QA, the type of the responsibility (defined by the category) plays a major role and, thus, the category has influence on the result. For example, a Creational pattern such as Factory Method presented fewer violations in all cases (in average, 48% fewer than the second-ranked pattern). A plausible explanation is the structural simplicity of this pattern. From the studied Creational patterns, Factory Method is potentially the simplest one, because all other patterns allow more complex implementations (e.g., the cloning mechanism of Prototype and the unique instantiation of Singleton). This finding is in accordance with the related work. For both correctness and performance, studies showed that Factory Method is among the patterns with best scores [6,19,28].

Furthermore, we notice that the difference in the patterns' definition and purpose may also reflect on the violations that are accumulated on their instances. By examining Table XVIII (Supplementary Appendix A), we observe that despite similarities, some of the most recurrent violations differ among the patterns. For example, unsafe multithreaded calls are more recurrent for four out of the 12 analyzed patterns. Another interesting observation is that instances of some patterns tend to accumulate violations of higher severity ("scary" or "scariest", according to FindBugs classification). In particular, although instances of Factory Method accumulate fewer violations of correctness, the most recurrent ones are potentially more harmful to the system (see

**Table 15**
Comparable observations between static and dynamic analyses.

| Topic | Observation | |
|---|---|---|
| | Results from Static Analysis (this paper) | Results from Dynamic Analysis |
| PP vs. NPP classes | Suggest that NPP classes are more likely to underperform. | Show cases in which PP classes underperform and cases in which NPP classes underperform. |
| | Results may be attributed to the promotion of best practices. | Results may depend on consideration of good design practices. Proper use of patterns (e.g., to organize complex logic) may virtually remove the gap between pattern and non-pattern solutions. |
| | Suggest that violations are more often removed from PP classes than NPP classes, and that the number of violations in PP classes are lower. | Suggest that PP classes are likely to be more change-prone, which reflect on higher error-proneness. |
| Creational patterns | Suggest that instances of Factory Method are likely to exhibit better scores than other patterns. | Suggest that Factory Method is among patterns with the best performance. |
| | Suggest that creational patterns (e.g., Factory Method) are likely to exhibit better scores than patterns of other categories. | Creational patterns (e.g., Factory Method and Prototype) appear among patterns with the best performance. |

Table XVIII). This shows that even with a tendency of accumulating fewer violations, it might be important to monitor instances of certain patterns.

Finally, the findings from $RQ_4$ suggest that classes playing more generic roles (i.e., Superclass and Container) tend to show better scores than classes playing more concrete ones (i.e., Subclass and Containee). A possible explanation for these observations is that more specialized roles implement more intense and complex business logic and, therefore, are more prone to violations. The counting presented in Table XX (Supplementary Appendix A) supports this hypothesis, as one can see that more specialized roles showed larger number of violations with higher severity. Moreover, the type of violation may also be different depending on the role a class plays (see Table XIX on Supplementary Appendix A). These observations may also be partially related to the likelihood of a class to be changed. It is intuitive to expect that the more frequently a class is changed the more violations are introduced into its source code. Di Penta et al. [41] investigated the correlation of pattern roles to the changes in pattern participants. Their findings suggest that some meta-roles (i.e., Subclass and Client) changed more frequently than the other meta-roles). This may indicate that change- and violation-proneness are related.

### 5.2. Implications for practitioners and researchers

The findings of this article suggest that PP classes are likely to have fewer security, correctness and performance violations than NPP classes, even in cases in which a class participates in more than one pattern. Stated differently, this exploratory study revealed that adhering to good architectural practices (such as the use of patterns) is often accompanied by (or due to) better programming practices, leading to fewer violations. One could argue that a well-designed architecture besides its obvious benefits in supporting software evolution provides a solid basis for developing cleaner code with fewer violations. Building an application around reusable, documented and well-tested pattern instances improves comprehensibility, thereby limiting the possibilities of accidentally introducing violations. Moreover, the clean code structure facilitates easier bug localization and removal.

This study has two main implications to researchers. The exploitation of static analysis and, in particular, source code for investigating run-time QAs can add to the current state of the art on the relationship between the presence of patterns and security, correctness and performance. The comparison of our results to related work has in some cases led to contradictions, due to the different nature of our measurement approach, but some common implications can be retrieved. First, the fact that no universal assessment on the relationship between patterns and run-time qualities can be made (without performing a separate investigation per pattern type) is a common finding in both our study and almost every related work in the field. Second, the fact that the structural complexity of the pattern is playing an important role on the relationship of patterns and run-time qualities is confirmed by our case study (e.g., Creational patterns present fewer violations).

Furthermore, to the best of our knowledge, this study is the first to provide empirical evidence on the relationship between GoF patterns and security. In addition, it is interesting to notice that most of the statistically significant results were w.r.t. security. Therefore, we suggest the usage of static analysis when investigating this QA. Finally, the investigation of pattern roles and, in particular, meta-roles has provided interesting and insightful findings to our study, showing to be a valuable source of information when it comes to investigate GoF patterns. The investigation of roles allows a finer-grained analysis of the patterns, while considering meta-roles brings the discussion to a more abstract level, considering characteristics as mechanisms used by the patterns. Thus, we also encourage the consideration of meta-roles when investigating GoF patterns, especially if such characteristics are clearly relevant for interpreting results of the study.

## 6. Threats to validity

In this section, we present and discuss the threats to the validity of our study, in particular, construct validity, reliability and external validity. Internal validity is not applicable, as the study does not examine causal relations. Construct validity reflects the connection between the object of study, or studied phenomenon, and the RQs. Reliability is related to the possibility of others replicating the performed case study and obtaining the same results. Finally, external validity comprises possible threats to the generalization of the findings on this study to the entire population.

Concerning construct validity, it can be argued that static analysis does not assess run-time qualities as effectively and precisely as dynamic analysis. Indeed, dynamic analysis has been used much more extensively than static analysis in assessing run-time qualities and such results are more well-established. To partially mitigate this threat, we compared some of our results with those from studies using dynamic analysis. The comparison indicates that the results from static analysis are to a large extent aligned with those from dynamic analysis (with some exceptions discussed in Section 5.1), so we consider this threat to some extent addressed. Another threat related to construct validity is that the SSA tool is limited by its precision and recall: false positives and negatives may bias the presented results. However, to the best of our knowledge, the used tool is among the most reputed in the community, and has adequate performance (see Section 3.3.1). For mitigating this threat, we manually verified its precision and recall by checking 50 random pattern instances for each GoF pattern that is detected by SSA tool (i.e., over 500 instances in total—standing for 32% of the total dataset in terms of pattern instances), which were true positives. We note that the level of agreement between the researchers was approximately 98%, since only for very few instances there was an initial disagreement that was resolved through discussion among the two of the authors. Additionally, regarding FindBugs, we acknowledge that the list of bug patterns is by no means exhaustive and additional bugs related to security, correctness and performance could be used. However, to the best of our knowledge this tool is also among the most reputed in the community, and has adequate performance (see Section 3.3.2).

Finally, this study assumes that PP classes in the dataset contribute to pattern instances that are correctly implemented. A pattern implementation might not be the correct one, due to either a programming mistake or pattern grime [55], or because the deployed pattern is not the optimal way to solve the underlying problem considering that the effect of a pattern on quality is affected by different factors [56]. This poses a threat to construct validity. To mitigate it, we checked all the manually verified pattern instances; we found that their implementation was correct and they were a suitable solution for the problem at hand. Since these instances account for 32% of the dataset, we consider that this threat is to a large extent mitigated.

In order to mitigate reliability, two different researchers were involved in the data collection procedure, double-checking all outputs. Furthermore, the same researchers also double-checked the data analysis. Finally, all primitive data can be reproduced by using the same cases and tooling. The pattern detection tool (SSA v4.5) and bug detection tool (FindBugs, v3.0.0) were downloaded from the provided sources, while we have made the tool developed by us (SSA+ v1.0) publicly available.

Finally, concerning external validity, we identified the following threats. First, not all parts of the architecture can benefit from the introduction of patterns, as the pattern goal might be irrelevant to the functionality/design of that part. Thus, the outcomes of this study do not generalize to all parts of the codebase or the design space, but only to those where the use of a design pattern would be beneficial and applicable. Second, we investigated a limited number of OSS projects. However, the five projects selected are the most popular projects in SourceForge that fitted our selection criteria. Additionally, they vary in

terms of both domains and characteristics; this partially alleviates this threat. Next, we investigate a limited number of patterns, as well as pattern instances. A larger sample could strengthen the results and increase our confidence on generalizing our findings. Similarly, we investigated a subset of all run-time QAs and, therefore, our results cannot be generalized to all QAs without further investigation. Finally, we investigated OSS projects written in Java only, while all used tooling was Java-specific. Therefore, our observations focus on this programming language and cannot be generalized to other OO languages without further investigation.

## 7. Conclusion

In this article, we investigated the relationship of 12 GoF patterns to three run-time QAs, namely security, correctness and performance. In particular, we conducted a case study on multiple versions of five OSSs among the most popular Java projects on SourceForge platform (Bonita BPM, Convertigo, Eclipse Checkstyle Plug-in, Hibernate and Logical DOC), from which we collected 12,857 classes, being approx. 25% single-pattern participants, 13% coupled-pattern participants and 62% non-pattern participants.

To investigate the relationship between GoF patterns and the aforementioned QAs, we explored source code violations, defining three metrics for each QA, namely the number of New violations, Removed violations and Total number of violations. Considering these metrics, we estimated the levels of the three QAs for each collected class of each version. We investigated the relation of these metrics to GoF patterns with regards to four different perspectives: pattern participation (i.e., NPP, SPP or CPP), pattern category, pattern and meta-role. Results of the study suggest that classes not participating in any pattern are more prone to violations, as well as that participation in more than one pattern can also be connected to the existence of more violations. In addition, classes participating in Creational patterns, especially Factory Method, or playing more generic meta-roles (e.g., Container) are likely to have fewer violations than other classes. However, we advise being attentive of these violations as we found them to often be of higher severity.

The findings of this study, although they do not imply any causality between the introduction of patterns and the removal of violations, provide evidence that code residing around design patterns adheres to good programming practices. This observation is consistent with the wide-spread belief that the application of patterns complies with the adoption of software architecture principles.

Finally, in light of our findings, we envisage several opportunities of future work. For example, by normalizing PP classes over the size/functionality of each system, it would be interesting to compare the overall level of QAs across projects that have more vs. less pattern instances. In addition, FindBugs reports a severity level for each violation, which was not explored in our study design due to its complexity. Thus, it would be interesting to extend the current work to factor in the severity of violations.

## Acknowledgments

The authors would like to thank the financial support from the Brazilian and Dutch agencies CAPES/Nuffic [grant number 034/12]; CNPq [grant number 204607/2013-2]; and the INCT-SEC [grant number 573963/2008-8, 2008/57870-9].

## Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.infsof.2018.07.014.

## References

[1] E. Gamma, R. Helm, R.E. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., 1995.
[2] M. Riaz, T. Breaux, L. Williams, How have we evaluated software pattern application? A systematic mapping study of research design practices, Inf. Softw. Technol. 65 (2015) 14–38, https://doi.org/10.1016/j.infsof.2015.04.002.
[3] A. Ampatzoglou, S. Charalampidou, I. Stamelos, Research state of the art on GoF design patterns: a mapping study, J. Syst. Softw. 86 (2013) 1945–1964, https://doi.org/10.1016/j.jss.2013.03.063.
[4] B. Bafandeh Mayvan, A. Rasoolzadegan, Z. Ghavidel Yazdi, The state of the art on design patterns: a systematic mapping of the literature, J. Syst. Softw. 125 (2017) 93–118, https://doi.org/10.1016/j.jss.2016.11.030.
[5] A. Litke, K. Zotos, A. Chatzigeorgiou, G. Stephanides, Energy consumption analysis of design patterns, Int. J. Electr. Comput. Energ. Electron. Commun. Eng. 1 (2007) 1663–1667 http://waset.org/publications/2689.
[6] C. Sahin, F. Cayci, I.L.M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, Proc. First Int. Work. Green Sustain. Softw. (GREENS '12), IEEE, Zurich, Switzerland, 2012, pp. 55–61, , https://doi.org/10.1109/GREENS.2012.6224257.
[7] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs, IEEE Softw. 25 (2008) 22–29, https://doi.org/10.1109/MS.2008.130.
[8] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, M.A. Vouk, On the value of static analysis for fault detection in software, Softw. Eng. IEEE Trans. 32 (2006) 240–253, https://doi.org/10.1109/TSE.2006.38.
[9] A.K. Tripathi, A. Gupta, A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for Java programs, Proc. 18th Int. Conf. Eval. Assess. Softw. Eng. (EASE '14), ACM, London, England, United Kingdom, 2014, , https://doi.org/10.1145/2601248.2601288 23:1-23:4.
[10] N. Ayewah, W. Pugh, The Google FindBugs fixit, Proc. 19th Int. Symp. Softw. Test. Anal. (ISSTA '10), ACM, Trento, Italy, 2010, pp. 241–252, , https://doi.org/10.1145/1831708.1831738.
[11] C. Sadowski, J. van Gogh, C. Jaspan, E. Soderberg, C. Winter, Tricorder, Building a program analysis ecosystem, Proc. 37th IEEE Int. Conf. Softw. Eng. (ICSE '15), IEEE, Florence, Italy, 2015, pp. 598–608, , https://doi.org/10.1109/ICSE.2015.76.
[12] W. Schilling, M. Alam, A methodology for quantitative evaluation of software reliability using static analysis, Annu. Reliab. Maintainab. Symp. (RAMS '08), IEEE, Las Vegas, NV, USA, 2008, pp. 399–404, , https://doi.org/10.1109/RAMS.2008.4925829.
[13] H. Khalid, M. Nagappan, A.E. Hassan, Examining the relationship between FindBugs warnings and app ratings, IEEE Softw. 33 (2016) 34–39, https://doi.org/10.1109/MS.2015.29.
[14] K. Goseva-Popstojanova, A. Perhinschi, On the capability of static code analysis to detect security vulnerabilities, Inf. Softw. Technol. 68 (2015) 18–33, https://doi.org/10.1016/j.infsof.2015.08.002.
[15] G. Díaz, J.R. Bermejo, Static analysis of source code security: assessment of tools against SAMATE tests, Inf. Softw. Technol. 55 (2013) 1462–1476, https://doi.org/10.1016/j.infsof.2013.02.005.
[16] H.H. AlBreiki, Q.H. Mahmoud, Evaluation of static analysis tools for software security, Proc. 10th Int. Conf. Innov. Inf. Technol. (IIT '14), IEEE, Al Ain, United Arab Emirates, 2014, pp. 93–98, , https://doi.org/10.1109/INNOVATIONS.2014.6987569.
[17] ISO/IEC, ISO/IEC 25010:2011 - Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models, 2011. http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733.
[18] A. Abran, J.W. Moore, P. Bourque, R. Dupuis, L.L. Tripp, Guide to the Software Engineering Body of Knowledge, third ed., 2014. http://www.computer.org/portal/web/swebok/htmlformat.
[19] M. Gatrell, S. Counsell, Design patterns and fault-proneness a study of commercial C# software, Proc. Fifth Int. Conf. Res. Challenges Inf. Sci. (RCIS '11), IEEE, Gosier, France, 2011, pp. 1–8, , https://doi.org/10.1109/RCIS.2011.6006827.
[20] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, P. Avgeriou, The effect of GoF design patterns on stability: a case study, IEEE Trans. Softw. Eng. 41 (2015) 781–802, https://doi.org/10.1109/TSE.2015.2414917.
[21] L. Aversano, L. Cerulo, M. Di Penta, Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study, IET Softw. 3 (2009) 395, https://doi.org/10.1049/iet-sen.2008.0105.
[22] W.B. McNatt, J.M. Bieman, Coupling of design patterns: common practices and their benefits, Proc. 25th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC '01), IEEE, Chicago, IL, USA, 2001, pp. 574–579, , https://doi.org/10.1109/CMPSAC.2001.960670.
[23] F. Khomh, Y.-G. Gueheneuc, G. Antoniol, Playing roles in design patterns: an empirical descriptive and analytic study, Proc. 25th IEEE Int. Conf. Softw. Maint. (ICSM '09), IEEE, Edmonton, AB, Canada, 2009, pp. 83–92, , https://doi.org/10.1109/ICSM.2009.5306327.
[24] M.Q. Patton, Qualitative Research & Evaluation Methods: Integrating Theory and Practice, SAGE Publications, 2014.
[25] F. Cohen, Identifying And avoiding SOA performance problems, FastSOA, Elsevier, 2007, pp. 75–101, https://doi.org/10.1016/B978-012369513-0/50005-7.
[26] M. Aleksy, A. Korthaus, C. Seifried, Design patterns usage in peer-to-peer systems–an empirical analysis, Proc. 2006 IEEE/WIC/ACM Int. Conf. Web Intell. Intell. Agent Technol. Work. (WI-IATW '06), IEEE Computer Society, Hong Kong, China,

2006, pp. 459–462, , https://doi.org/10.1109/WI-IATW.2006.57.
[27] M. Ali, M.O. Elish, A comparative literature survey of design patterns impact on software quality, Proc. Fourth Int. Conf. Inf. Sci. Appl. (ICISA '13), IEEE, Suwon, South Korea, 2013, pp. 1–7, , https://doi.org/10.1109/ICISA.2013.6579460.
[28] M. Vokac, Defect frequency and design patterns: an empirical study of industrial code, IEEE Trans. Softw. Eng. 30 (2004) 904–917, https://doi.org/10.1109/TSE.2004.99.
[29] A. Ampatzoglou, A. Kritikos, E.-M. Arvanitou, A. Gortzis, F. Chatziasimidis, I. Stamelos, An empirical investigation on the impact of design pattern application on computer game defects, Proc. 15th Int. Acad. MindTrek Conf. Envisioning Futur. Media Environ. (MindTrek '11), ACM, Tampere, Finland, 2011, pp. 214–221, , https://doi.org/10.1145/2181037.2181074.
[30] L. Aversano, L. Cerulo, M. Di Penta, Relating the evolution of design patterns and crosscutting concerns, Proc. Seventh Int. Work. Conf. Source Code Anal. Manip. (SCAM '07), IEEE, Paris, France, 2007, pp. 180–192, , https://doi.org/10.1109/SCAM.2007.21.
[31] T. Afacan, State design pattern implementation of a DSP processor: a case study of TMS5416C, Proc. Sixth Int. Symp. Ind. Embed. Syst. (SIES '11), IEEE, Vasteras, Sweden, 2011, pp. 67–70, , https://doi.org/10.1109/SIES.2011.5953682.
[32] J. Rudzki, How design patterns affect application performance – a case of a multi-tier J2EE application, Proc. Fourth Int. Work. Sci. Eng. Distrib. Java Appl. (FIDJI '04), Springer-Verlag, Luxembourg-Kirchberg, Luxembourg, 2004, pp. 12–23, , https://doi.org/10.1007/978-3-540-31869-9_2.
[33] K. Chantarasathaporn, C. Srisa-an, Energy conscious factory method design pattern for mobile devices with C# and intermediate language, Proc. 3rd Int. Conf. Mob. Technol. Appl. Syst. (Mobility '06), ACM, Bangkok, Thailand, 2006, , https://doi.org/10.1145/1292331.1292364 p. 29:1-29:8.
[34] F.S. Ferraz, R.E. Assad, S.R. Lemos Meira, Relating security requirements and design patterns: reducing security requirements implementation impacts with design patterns, Proc. Fourth Int. Conf. Softw. Eng. Adv. (ICSEA '09), IEEE, Porto, Portugal, 2009, pp. 9–14, , https://doi.org/10.1109/ICSEA.2009.10.
[35] D.G. Firesmith, Engineering security requirements, J. Object Technol. 2 (2003) 53–68, https://doi.org/10.5381/jot.2003.2.1.c6.
[36] M. VanHilst, E.B. Fernandez, Reverse engineering to detect security patterns in code, Proc. 1st Int. Work. Softw. Patterns Qual. Information Processing Society of Japan, 2007, pp. 1–6.
[37] M. Hafiz, P. Adamczyk, R.E. Johnson, Organizing security patterns, IEEE Softw. 24 (2007) 52–60, https://doi.org/10.1109/MS.2007.114.
[38] P. Runeson, M. Host, A. Rainer, B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples, Wiley Blackwell, 2012.
[39] R. van Solingen, V. Basili, G. Caldiera, H.D. Rombach, Goal question metric (GQM) approach, Encyclopedia Software Engineering, John Wiley & Sons, Inc, Hoboken, NJ, USA, 2002, pp. 528–532, https://doi.org/10.1002/0471028959.sof142.
[40] A. Ampatzoglou, S. Charalampidou, I. Stamelos, Design pattern alternatives, Proc. 17th Panhellenic Conf. Informatics (PCI '13), ACM, Thessaloniki, Greece, 2013, pp. 122–127, , https://doi.org/10.1145/2491845.2491857.
[41] M. Di Penta, L. Cerulo, Y.G. Guéhéneuc, G. Antoniol, An empirical study of the relationships between design pattern roles and class change proneness, Proc. 24th Int. Conf. Softw. Maint. (ICSM '08), IEEE, Beijing, China, 2008, pp. 217–226, , https://doi.org/10.1109/ICSM.2008.4658070.
[42] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. Halkidis, Design pattern detection using similarity scoring, IEEE Trans. Softw. Eng. 32 (2006) 896–909, https://doi.org/10.1109/TSE.2006.112.
[43] G. Kniesel, A. Binun, Standing on the shoulders of giants - a data fusion approach to design pattern detection, Proc. 17th Int. Conf. Progr. Compr. (ICPC '09), IEEE, Vancouver, BC, Canada, 2009, pp. 208–217, , https://doi.org/10.1109/ICPC.2009.5090044.
[44] N. Pettersson, W. Löwe, J. Nivre, Evaluation of accuracy in design pattern occurrence detection, IEEE Trans. Softw. Eng. 36 (2010) 575–590, https://doi.org/10.1109/TSE.2009.92.
[45] D. Feitosa, P. Avgeriou, A. Ampatzoglou, E.Y. Nakagawa, The evolution of design pattern grime: an industrial case study, Proc. 18th Int. Conf. Prod. Softw. Process Improv. (PROFES '17), Innsbruck, Austria, 2017, pp. 165–181, , https://doi.org/10.1007/978-3-319-69926-4_13.
[46] D. Hovemeyer, W. Pugh, Finding bugs is easy, ACM SIGPLAN Not. 39 (2004) 92–106, https://doi.org/10.1145/1052883.1052895.
[47] N. Ayewah, W. Pugh, J.D. Morgenthaler, J. Penix, Y. Zhou, Evaluating static analysis defect warnings on production software, Proc. 7th ACM SIGPLAN-SIGSOFT Work. Progr. Anal. Softw. Tools Eng. (PASTE '07), ACM Press, San Diego, California, USA, 2007, pp. 1–8, , https://doi.org/10.1145/1251535.1251536.
[48] D. Feitosa, A. Ampatzoglou, P. Avgeriou, E.Y. Nakagawa, Investigating quality trade-offs in open source critical embedded systems, Proc. 11th Int. ACM SIGSOFT Conf. Qual. Softw. Archit. (QoSA '15), ACM, Montréal, QC, Canada, 2015, pp. 113–122, , https://doi.org/10.1145/2737182.2737190.
[49] A. Hora, N. Anquetil, S. Ducasse, S. Allier, Domain specific warnings: are they any better? Proc. 28th Int. Conf. Softw. Maint. (ICSM '12), IEEE, Trento, Italy, 2012, pp. 441–450, , https://doi.org/10.1109/ICSM.2012.6405305.
[50] A. Field, Discovering Statistics Using IBM SPSS Statistics, fourth ed., SAGE Publications Ltd, 2013.
[51] D. Riehle, Lessons learned from using design patterns in industry projects, Transactions on Pattern Languages of Programming II, Springer, Berlin / Heidelberg, 2011, pp. 1–15, https://doi.org/10.1007/978-3-642-19432-0_1.
[52] R. Ahlgren, J. Markkula, Design patterns and organisational memory in mobile application development, Proc. Sixth Int. Conf. Prod. Focus. Softw. Process Improv. (PROFES '05), Oulu, Finland, 2005, pp. 143–156, , https://doi.org/10.1007/11497455_13.
[53] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, E.Y. Nakagawa, Investigating the effect of design patterns on energy consumption, J. Softw. Evol. Process. 29 (2017) e1851, https://doi.org/10.1002/smr.1851.
[54] F.P. Brooks Jr., No silver bullet essence and accidents of software engineering, Computer (Long. Beach. Calif). 20 (1987) 10–19, https://doi.org/10.1109/MC.1987.1663532.
[55] C. Izurieta, J.M. Bieman, A multiple case study of design pattern decay, grime, and rot in evolving software systems, Softw. Qual. J. 21 (2013) 289–323, https://doi.org/10.1007/s11219-012-9175-x.
[56] A. Ampatzoglou, G. Frantzeskou, I. Stamelos, A methodology to assess the impact of design patterns on software quality, Inf. Softw. Technol. 54 (2012) 331–346, https://doi.org/10.1016/j.infsof.2011.10.006.