IET Software

Review Article

Impact of design patterns on software quality: a systematic literature review

ISSN 1751-8806
Received on 13th December 2018
Revised 29th July 2019
Accepted on 9th September 2019
E-First on 16th December 2019
doi: 10.1049/iet-sen.2018.5446
www.ietdl.org

Fadi Wedyan[1] ✉, *Somia Abufakher*[1]

[1]*Department of Software Engineering, The Hashemite University, P.O. Box 330127, Zarqa 13133, Jordan*

✉ *E-mail: fadi.wedyan@hu.edu.jo*

Abstract: The impact of design patterns on quality attributes has been extensively evaluated in studies with different perspectives, objectives, metrics, and quality attributes, leading to contradictive and hard to compare results. The authors' objective is to explain these results by considering confounding factors, practices, metrics, or implementation issues that affect quality. Furthermore, there is a lack of research that connects design patterns evaluations to patterns development studies. Accordingly, they also aim at providing an initiate on how patterns structure and implementation can be improved, to promote software quality. To achieve their goals, conducted a systematic literature review by searching the literature for related studies. The study covers the period between years 2000 and 2018. They identified 804 candidate papers. After applying inclusion and exclusion criteria, they were left with 50 primary studies. Their results show that documentation of patterns, size of pattern classes, and the scattering degree of patterns have clear impact on quality. In case studies, researchers used different metrics applied to different modules. Controlled experiments have major design differences. Reaching consensuses on the effect of patterns requires considering influencing factors, using unified metrics, and an agreement on what modules to measure. Studying how to improve patterns modularity is recommended for future research.

1 Introduction

Design patterns represent solutions to frequently occurring software problems for designing good quality software. They were proposed for the first time by Alexander *et al.* [1] in 1977 in the field of building architecture. In the mid-1990s, Gamma *et al.* [2] proposed approaches for applying architectural patterns on object-oriented (OO) software. They proposed 23 design patterns called gang of four (GoF) patterns. GoF design patterns are classified into three categories: structural, creational, and behavioural patterns.

Design patterns are solutions to problems at the design and implementation levels of software components. Design patterns are claimed to provide various benefits including (i) support for better design decisions, (ii) improving communication among developers, (iii) enhancing or easing software maintainability, and reusability, (iv) help in satisfying non-functional requirements of the system, and (v) saving cost, effort, and time by reusing existing proven solutions [2–4].

Many researchers conducted studies to investigate the above claims both empirically and analytically. Some of the studies indicate that design patterns impede the achievement of the required level of quality for the subject software. However, some other studies concluded that developers should use design patterns in order to improve software quality. For example, some studies concluded that the abstract factory pattern improves design extensibility (e.g. [5–7]); while other studies found that the use of abstract factory has a negative impact on extensibility [8–10]. Such contradictory results have been reported for most of the evaluated patterns [11, 12]. Therefore, there is little consensus about the real impact of any particular design pattern on software quality.

Studies were performed in order to collect evidence from available research in order to reach an agreement on the effect of using design patterns on certain quality attributes. These studies were performed as a mapping study (e.g. [12, 13]), or a literature review (e.g. [11, 14, 15]). The results of these studies clearly show that generally there is no consensus on the effect of design patterns on software quality. Riaz *et al.* [13] found that it is hard to compare the findings of empirical studies due to differences in study design and execution. They also concluded that the available empirical findings of the primary studies they evaluated are not well understood, and have limited generalisability.

In this study, we build upon previous work on the effort of providing a better understanding of the effect of using design patterns on software quality. We aim at providing an explanation of the contradictive results obtained by various results by studying the factors, practices or implementation issues that affect quality attributes when design patterns are used. We also collected and compared the various metrics used in evaluating design patterns and how these metrics are applied. Moreover, by providing an explanation of the relationship between the design pattern and quality that is built on a variety of empirical studies, we also aim at giving suggestions on how design patterns structure and implementation can be improved, to promote building high-quality software.

In order to achieve our study goals, we conducted a systematic literature review (SLR) of existing literature on the effect of using design patterns on software quality, in particular, GoF design patterns. The SLR covers the period between the years 2000 and 2018 (inclusive). We searched for relevant publications in primary digital libraries. We identified 804 candidate studies. After applying the inclusion and exclusion criteria, we were left with 50 primary studies. The primary studies include both analytical and empirical research.

The rest of this paper is organised as follows. Section 2 presents related work. Section 3 presents the procedure followed in the SLR and the research questions. Section 4 provides a summary of the primary studies. Section 5 discusses the results obtained from the selected primary studies and answers the research questions. Threats to validity are discussed in Section 6. Finally, Section 7 concludes the work.

2 Related work

The study of design patterns has gained significant attention since they were first proposed. Many studies have been performed to detect, categorise, utilise, and evaluate patterns empirically and analytically. Literature surveys play an important role in evaluating studies and shed light on the state of research and the milestones achieved.

Weiss [14] surveys 16 studies published between the years 2000 and 2008 on design patterns and their impact on system concerns. The author divides the primary studies into three categories; which are: (i) studies that explicitly link design patterns and system concerns, (ii) studies with the goal of supporting the selection of patterns, and (iii) studies that document the rationale for architectural decisions.

Zhang and Budgen [15] present a SLR in the form of a mapping study. The study investigates which of the GoF design patterns have been subjected to empirical studies, and what conclusions are available about their evaluation of the effects of using them. The study considers 11 empirical studies documented in ten papers published between the years 1995 and 2009.

Ampatzoglou *et al.* [12] present a mapping study of about 120 primary studies to provide an overview of the research state-of-the-art on GoF design patterns. They investigate: (a) if research efforts on design patterns can be categorised in research subdomains, (b) what are the most active subdomains, and (c) what evidence exists about the effect of patterns on software quality attributes.

Ali and Elish [11] survey the literature for existing empirical evidence on the impact of GoF design patterns on software quality attributes. They evaluated the coverage of empirical evidence in terms of both quality attributes and design patterns and provided a summary of the impact of design patterns on software quality attributes. The authors studied 17 papers published between the years 2001 and 2012.

Riaz *et al.* [13] present a study in which they analysed 19 empirical studies documented in 17 papers published between the years 2000 and 2009. They extracted beneficial information, such as variables associated with participants' demographics, pattern and problem presentations, in addition to ten evaluation criteria with their associated observable measures. They also identified challenges that researchers may face during conducting an empirical study.

Mayvan *et al.* [16] performed a mapping study on design patterns. They extracted data from 637 articles. Their results show that the design patterns field is an active and attractive research field. Moreover, they found that pattern development, pattern mining, and pattern usage are the most active topics, while less publication found on pattern evaluation and pattern specification topics.

Previous literature reviews and mapping studies show that there are problems in how design patterns are evaluated, in terms of their effect on quality. These problems lead to contradictions in results obtained from different studies. Moreover, studying the impact of design patterns or any design module is far from being simple, taking into consideration the large range of factors that influence how software is crafted. In order to successfully deploy a solution, a good understanding of the solution needs to be provided, including how to successfully deploy the solution, and how to identify and measure its effect.

In this study, we present a SLR of 50 primary studies published during the years 2000–2018. Our study differs from the above studies in many aspects. We studied all types of approaches, i.e. both empirical and analytical, with respect to all quality attributes and GoF design patterns, which makes our work the most recent and comprehensive. Our results clearly identify confounding factors that affect software quality when design patterns are used. We also analysed the differences in metrics used to measure quality and differences in the measured software artefacts. Our findings can increase the understanding of the inconsistency in the results obtained by different studies. Moreover, we outlined the areas that need more research, mainly on how to improve software quality by carefully deploying design patterns, and how to study patterns effect on quality.

3 Systematic review procedure

According to Kitchenham [17], 'a SLR is a means of identifying, evaluating, and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest.' In this work, we carry out a SLR by following the guidelines provided by Kitchenham [17]. The objectives of this SLR are:

- Identifying confounding factors that affect the deployment of design patterns.
- Identifying whether the structure and implementation of a design pattern had an effect on quality attributes.
- Identifying the metrics used to measure quality attributes in the studies that evaluate the effect of design patterns on quality.
- Understanding how design patterns affect software quality and explain how to reach consistency for this effect.
- Categorising approaches used in evaluating the effect of design patterns on software quality.
- Identifying and cataloguing threats to validity reported in the studies that evaluate the effect of design patterns on software quality.

In the following sections, we present the details of our methodology.

3.1 Research questions

Based on the study objectives, the research questions of this SLR are:

- RQ1: What confounding factors, practices, or programming constructs affect quality attributes when design patterns are used?
- RQ2: What quality attributes are evaluated, what is measured, and what metrics are used?
- RQ3: What are the common threats to validity reported in the primary studies?

Software quality is often described as a context-dependent concept [18]. In order to achieve high-quality software, there are various measures that have to be considered. Many factors can contribute to producing better quality software. When design patterns are deployed, these factors cannot, and should not be eliminated. Moreover, the correct use of design patterns can also be affected by how developers are using design patterns and their level of experience. By answering this question, we collect from the primary studies the factors that researchers had considered when evaluating design patterns. We also identify whether more factors should be considered.

The second question is related to the relevance of quality attributes, which attributes researchers considered to be related to the use of design patterns. Software quality is multidimensional, consisting of many attributes, some of which might contradict. Therefore, we need to identify which attributes can be influenced by the use of design patterns, which have been largely evaluated and ignored.

Another important issue is the metrics used to evaluate quality attributes and how these metrics are applied. This helps in explaining the differences in results obtained by different studies.

Finally, the answer to the third question can help (i) future researchers avoid these threats while performing their empirical evaluations and (ii) validate the results of existing studies.

3.2 Search process

The search process was performed in two steps. In the first step, search for relevant publications was performed in digital libraries. This step includes identifying data sources and specifying search terms. In the second step, the study selection strategy is applied. The second step requires specifying the inclusion/exclusion criteria and the quality assessment strategy. In the following sections, we discuss the details of these steps.

3.2.1 Search terms and search resources: We derived the search terms using our research questions. We determined alternative spellings for each term and then performed the search process. We used the following search string:

Table 1 Number of obtained papers from the searched digital libraries

Digital library	No. obtained papers
ACM Digital Library	560
IEEE Digital Library	212
Science@Direct	647
SpringerLink	369
Wiley Online Library	24
total	1812

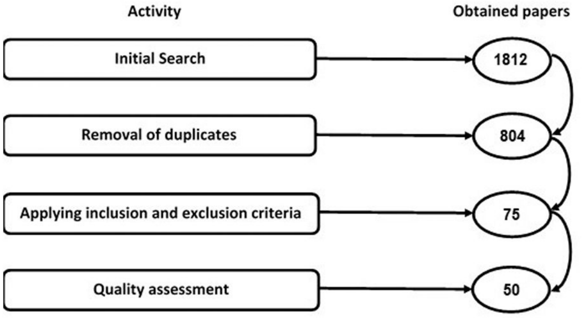

Fig. 1 *Primary studies selection process*

('design patterns and quality' OR 'design patterns and non-functional requirements' OR 'design patterns and software concerns' OR 'impact of design patterns' OR 'design patterns advantages' OR 'design patterns benefits' OR 'evaluation of design patterns' OR 'why to use design patterns')

Search for relevant publications was performed in the following well-known digital libraries:

- ACM Digital Library (http://portal.acm.org)
- IEEE Digital Library (http://www.computer.org)
- Science@Direct (http://www.sciencedirect.com)
- SpringerLink (http://www.springerlink.com)
- Wiley Online Library (http://onlinelibrary.wiley.com)

In Table 1, we show the number of obtained papers from each of the digital libraries. The search was performed using the search string on the metadata (title, abstract, and keywords). The search was limited to the period between 2000 and 2018 (inclusive), which is the period covered in this SLR. The results obtained from the different databases contain many redundant articles (i.e. the same article is returned by more than one search engine). There is a high percentage of redundant articles because many articles might be indexed in more than one database (e.g. an ACM/IEEE conference article). The percentage of redundant retrieved articles in our study is about 55%. This is close to what was found by Jabangwe *et al.* [18] (~51% of redundant articles). Dyba *et al.* [19] also reported that databases relevant to software engineering, such as ScienceDirect and Wiley, returned similar search results as ACM and IEEE. The total number of retrieved articles at this stage (after excluding duplication) is 804 articles.

3.2.2 Primary study selection: The following inclusion and exclusion criteria are applied:

- Only peer reviewed articles published in journals, conferences, and workshops indexed in Scopus [20] are included.
- Articles that study the impact of using at least one of the GoF design patterns on OO software systems (e.g. Java, C#, or C++) are included. Articles that study non-GoF design patterns are excluded (e.g. agent design patterns [21], patterns for distributed, concurrent, or web-based systems, and user interface patterns).
- Only articles written in English are included.
- In the case where the same study is published in more than one article, i.e. when a study is first published as a short paper in a conference and subsequently as an extended journal version, we consider only the journal version as a primary study (e.g. Izurieta and Bieman's [22] short paper was extended in the form of a journal paper [23]).

Papers that do not meet the inclusion criteria are excluded. Exclusion of irrelevant articles is decided by reading the paper title, abstract and keywords.

We applied the inclusion and exclusion criteria on 804 papers. Accordingly, 729 papers were removed. This left us with 75 papers. We then applied the quality assessment strategy on the remaining papers.

We assessed the 75 candidate primary studies using the following factors (see Fig. 1):

- F1: The article has clear objectives that are related to our study.
- F2: The article clearly describes the research methodology.
- F3: The article specifies what metrics are used to measure quality attributes and describes how these metrics are computed.
- F4: The article discusses threats to validity and/or study limitations.

Applying the quality assessment requires reading the full-text of each candidate's primary study. We performed a quality assessment in meetings between the first and second authors. First, we sorted the candidate's primary studies in ascending order using publication date. Starting from the oldest papers in the list and before each meeting, we select a group of candidate primary studies to discuss. The number of articles discussed in each meeting depends on many factors including the meeting duration and the size of articles to be discussed. In each meeting, the first and second authors discussed whether the paper qualifies to be one of the primary studies by assessing the quality factors. In the case where there is no agreement on whether to qualify an article or one of the authors did not understand an aspect of the article, the article is added to a list of papers that need a second round of discussion. In this round, we invited reviewers to join the meeting (either graduate students or faculty members) to solve the dispute.

3.3 Information extraction

For each study, we collected two sets of information. The first set, which we called general information, includes the following variables: type of publication (journal, conference), publication venue, publication date, and number of pages. We used this information to find some descriptive statistics about the obtained papers. In the second set, which we called detailed information, we extracted the following information:

- The design pattern(s) investigated.
- The evaluated quality attributes,
- Metrics used for evaluating the quality attributes. Including artefacts that the metric evaluates
- Research methodology (analytical, controlled experiment, and case study).
- Threads to validity reported.

The retrieved information is then classified according to (i) design patterns, (ii) quality attributes, and (3) research methodology.

4 Summary of primary studies

The list of 50 primary studies used in this SLR is given in the Appendix (Section 10). In the rest of this paper, we will be using the ID given to each primary study in Section 10 to refer to the corresponding paper. In this section, we summarise information obtained from the primary studies including trend of publication in years, publication venues, design patterns evaluated, data sets, evaluated quality attributes, and research methods.

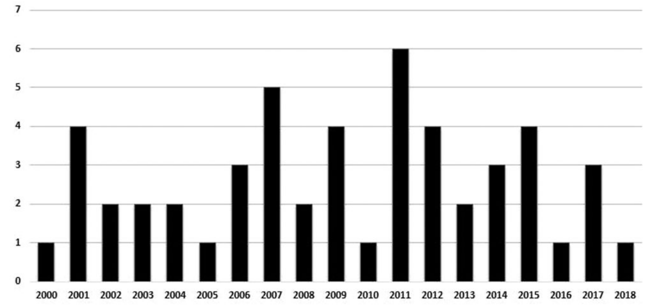

Fig. 2 *Number of primary studies per publication year*

Table 2 Primary studies types and publication venues

Type	Number
Journals	23
Information and Software Technology	5
IEEE Transactions on Software Engineering	4
Empirical Software Engineering	3
Journal of Systems and Software	3
ACM Transactions on Software Engineering and Methodology	1
Software Quality Journal	1
IET Software	1
IEEE Access	1
Journal of Software: Evolution and Process (Wiley)	1
Journal of Computer Science and Technology (Springer)	1
Transactions on AO Software Development	1
International Journal of Information System Modeling and Design	1
Conferences and Workshops	27

4.1 Publication years, Venues, and authors

In order to understand the general trend of evaluating the effect of design patterns on software quality, we counted the number of publications by year from 2000 till 2018 (period covered in our study). The results are shown in Fig. 2. While our results do not show a consistent increase in the number of publications, it shows a close number of annual publications, which indicates that the interest in the field has not changed during the last 19 years. However, there is no direct evidence to explain the spikes in the years 2007 and 2011.

Table 2 shows the primary studies publication venues. Journal papers contribute 46% of the primary studies, while the remaining are published in conferences and workshops. All primary studies are published in specialised software engineering journals and conferences, except for PS50, which is published in a multidisciplinary journal.

Identifying main researchers in a field of study can be helpful for those interested in the field. In Table 3, we show the most active researchers who studied the effect of design patterns on quality (researchers with three or more papers in the primary studies). We counted the number of papers published by each author, whether the first author or not (column 2). We show in column 3, the primary studies authored by each researcher. In total, we identified 107 authors in the primary studies, of whom, 32 have two or more studies (nine authors have three papers, ten authors have four papers, and only one author has more).

4.2 Research methods

In this SLR, we included both analytical and empirical studies. Fig. 3 shows the number of primary studies that use each of the research methods. Empirical evaluation, whether in the form of case studies, controlled/quasi-experiments, or surveys are widely used (total of 42 of the primary studies). In the following sections, we summarise our findings and observations about the characteristics of the primary studies that used each of the encountered research methods.

Table 3 Active researchers on the effect of design patterns on quality

Author	Primary studies
Ampatzoglou, A.	PS18, PS31, PS33, PS35, PS39, PS45, PS47, PS49, PS50
Avgeriou, P.	PS45, PS47, PS49, PS50
Bieman, J. M.	PS3, PS4, PS9, PS38
Cerulo, L.	PS16, PS19, PS21, PS23
Chan, W.	PS13, PS14, PS17, PS37
Cheung, S.	PS13, PS14, PS17, PS37
Guéhéneuc, Y.G.	PS20, PS21, PS26, PS42
Ng, T.	PS13, PS14, PS17, PS37
Stamelos, I.	PS31, PS33, PS35, PS49
Tichy, W.F	PS1, PS5, PS7, PS11
Yu, Y.	PS13, PS14, PS17, PS37
Aversano, L.	PS16, PS19, PS23
Counsell, S.	PS24, PS28, PS36
Di Penta, M.	PS16, PS19, PS23
Gatrell, M.	PS24, PS28, PS36
Gravino, C.	PS27, PS29, PS43
Risi, M.	PS27, PS29, PS43
Scanniello, G.	PS27, PS29, PS43
Tortora, G.	PS27, PS29, PS43
Unger, B.	PS1, PS5, PS7

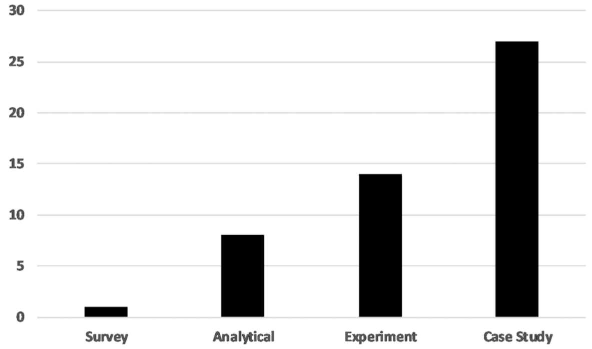

Fig. 3 *Research methods used in primary studies*

4.2.1 Analytical: In analytical research methods, researchers analyse information and facts to perform a critical evaluation of the studied material or phenomenon [24]. In the case of design patterns, researchers analyse the effect of a particular pattern structure, pattern implementation, variations, and the pattern uses to evaluate the effect of the pattern on selected software quality attributes. Primary studies that applied analytical methods used synthetic examples for illustration. Analytical methods are used in eight of the primary studies; these are PS2, PS4, PS6, PS8, PS22, PS30, PS35, and PS49.

4.2.2 Case studies: Case studies are widely used in software engineering to provide empirical evidence to support a hypothesis about a new approach. As discussed by Runeson and Höst [25], case studies are important for software engineering research because they study contemporary phenomena in their natural context. Case studies are used in 27 primary studies. The characteristics of these case studies are given in Table 4. In the table, for each primary study (column one), we show which programming language used (column two), subject programmes (column three), and the approach used in finding the design patterns in the subject programmes (column four).

For case studies, we can recognise three observations from Table 4. First, Java is the dominant programming language for the data sets, especially in recent studies. Second, some datasets are

Table 4 Characteristics of case studies

Primary study	Programming language	Subject programmes	Pattern mining approach
PS3	C++	one commercial software	manual
PS9	C++, Java	three commercial software (C++), NetBeans, JRefactory	manual
PS10	C++	one commercial software	developed their own tool
PA12	Java	two OSS – JBoss, JOnAS	manual
PS 16	Java	two OSS – JHotDraw, TomCat	documented patterns
PS18	C++, Java	two OSS-games	SSA for Java, manual with reverse engineering tool for C++
PS19	Java	three OSS – JHotDraw, ArgoUML, Eclipse-JDT	SSA
PS21	Java	three OSS – JHotDraw, Xerces-J, Eclipse-JDT	Demima
PS23	Java	three OSS – JHotDraw, ArgoUML, Eclipse-JDT	SSA
PS24	C#	one commercial software	manual
PS28	C#	one Commercial software	manual
PS31	Java	97 OSS (games)	SSA
PS32	Java	three OSS – JHotDraw, Xerces, ExclipseJDT	Demima
PS33	Java	100 OSS, reusability	SSA
PS34	Java	300 revisions of JHotDraw	documented patterns
PS36	C#	one commercial software	manual
PS38	Java	three OSS – JRefactory, ArgoUML, eXist	design pattern finder, design pattern seeker, manual
PS39	Java	26 OSS	SSA
PS40	Java	three OSS- JHotDraw, ArgoUML, Eclipse JDT	SSA
PS41	Java, AspectJ	three OSS – OpenOrb, AJATO, Expert Committee	documented patterns
PS42	Java	three OSS – ArgoUML, JFreeChart, Xerces-J	Demima
PS44	Java	five OSS – JHotDraw v5.1, JUnit v3.7, Lexi v0.1.1, Nutch v0.4, PMD v1.8	documented patterns from P-mart repository [26]
PS45	Java	537 OSS	SSA, PINOT
PS46	Java	two OSS – Apache Maven (32 releases) and JFreeChart (55 releases)	SSA
PS47	Java	two OSS – JHotdraw v7.6 and Joda Time v2.9.2	SSA
PS48	Java	52 OSS	SSA
PS50	Java	five industrial software	SSA

Table 5 Most common datasets in case studies

Programme	Programming language	No. studies	Primary studies
JHotDraw	Java	9	PS16, PS19, PS21, PS23, PS32, PS34, PS40, PS44, PS47
Eclipse	Java	5	PS19, PS21, PS23, PS32, PS40
ArgoUML	Java	5	PS19, PS23, PS38, PS40, PS42
Xerces-J	Java	3	PS21, PS32, PS42
JFreeCart	Java	2	PS42, PS46
JRefactory	Java	2	PS9, PS42

repeatedly used in many primary studies. In Table 5, we show the most common used datasets in case studies (used two or more times). The use of these datasets in many studies might be due to having documented design patterns, as in the case of JHotDraw, or might be due to the limitations of design pattern mining tools, as researchers might tend to use subject programmes for which the mining tools are guaranteed to work, because they worked with these programmes in previous studies. Working on the same datasets helps in comparing results, however, this can also decrease the generalisation of the results and is a clear external threat to validity.

Third, design pattern mining (detection) tools are used in all primary studies that evaluated subject programmes implemented in Java, except when the patterns are documented. In primary studies that evaluated subject programmes implemented in C++ and C#, design patterns are detected manually (except PS10). While this might indicate that researchers generally trust design patterns mining tools for Java programmes, other researchers argue that the precision and recall of these tools are still unsatisfactory, therefore, it is better to study software with documented design patterns (e.g. PS44), or verify the tool with manual inspection (e.g. PS38). For a comparison of design patterns mining tools, please refer to [27]. In the primary studies included in this study, the following tools were used to detect design patterns in Java programmes:

- The tool developed by Tsantalis *et al.* [28], which is based on a similarity scoring algorithm (SSA). The algorithm finds scores that reflect similarity of graphs representing original patterns and the analysed code. A pattern is considered as correctly identified if the similarity score exceeds a pre-defined threshold. As shown in Table 4, the tool was used in 12 out of 18 case studies performed on Java programmes with undocumented patterns. The current version of the tool can detect the following 11 design patterns: adapter/command, composite, decorator, factory method, observer, prototype, proxy, singleton, state/strategy, template method, and visitor. Notably, the tool cannot distinguish between some patterns (i.e. adapter and command, state and strategy) because of their identical static structure.
- Design motif identification multi-layered approach (DeMIMA) [29]. DeMIMA is built on the Ptidej [30] framework which constructs static and dynamic models of Java source code. DeMIMA consists of three layers: two layers to recover an abstract model of the source code, and a third layer to identify design patterns in the abstract model. The current version of

Table 6 Characteristics of controlled experiments

Primary study	Programming language	Participants (subjects)	Tasks
PS1	C++	students (15 grad)	two maintenance tasks on two small programmes from Academia
PS5	C++	29 SW engineers	two maintenance tasks on two self-developed programmes, tasks performed on papers
PS7	C++, Java	students (74 grad, 22 undergrad)	four maintenance tasks on two small programmes from Academia, two versions for each programme (Java, C++)
PS11	C++	five grad students, 39 SW engineers	two maintenance tasks on four self-developed small programmes
PS13	Java	98 grad. students with five years of industrial experience	rewrite one programme from Academia to satisfy three requirements
PS14	Java	students (55 undergrad, 63 grad)	three tasks performed on a part of JHotDraw (original) and on refactored a version of JHotDraw without patterns
PS15	Java	20 SW engineers, 10 undergrad. students	self-developed five programming tasks
PS17	Java	215 undergrad students	three maintenance tasks on three OSS (JHotDraw, MCM, HMS)
PS25	UML	students (37 grad, 10 undergrad)	eight UML class diagrams, one modification task on the diagram, one understandability task (questionnaire)
PS26	Java	students (17 grad, 7 post grad)	six tasks on three OSS (JHotDraw, JRefactory, and PADL)
PS27	Java	students (24 grad)	one task on code from JHotDraw, post experiment survey
PS29	Java	students (41 grad)	one task on code from JHotDraw, post experiment survey
PS37	Java	students (118 undergrad)	three tasks on part of JHotDraw
PS43	Java	25 professional SW eng. students (40 grad, 23 undergrad	task on part of JHotDraw, post-experiment survey

SW, software.

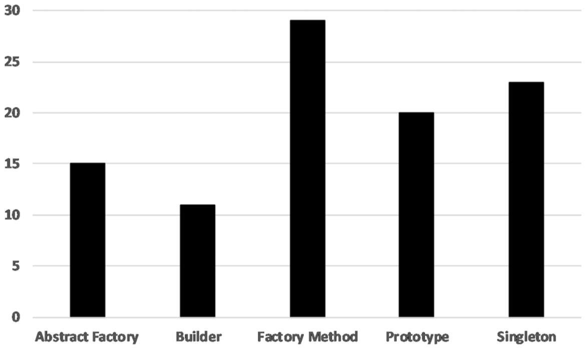

Fig. 4 *Evaluation of creational patterns in primary studies*

DeMIMA can detect the following 16 design patterns: abstract factory, adapter, builder, chain of responsibility, command, composite, decorator, factory method, facade, observer, prototype, proxy, singleton, state/strategy, template method, and visitor. DeMIMA was used by three of the primary studies as shown in Table 4.

- Pattern inference and recovery tool (PINOT) [31]. The tool uses static analysis and knowledge about the behaviour aspects of the pattern. The current version of the tool can detect the following 17 patterns: abstract factory, adapter, bridge, chain of responsibility, composite, decorator, facade, factory method, flyweight, mediator, observer, proxy, singleton, strategy, state, template method, and visitor. PINOT was used by primary study PS45.
- Design pattern finder and design pattern Seeker. The tools were developed at Colorado State University for identifying intended design patterns. The tools were used by PS38 with the help of a script written by the paper authors. The authors also verified the results manually.

4.2.3 Controlled or quasi-experiments: Experiments are used for measuring the effects of manipulating one variable on other variables while controlling other variables that might affect the results at a fixed level [32]. There are 14 primary studies that used controlled experiments. Their characteristics are given in Table 6. In the table, for each primary study (column one), we show which programming language used (column two), the subjects participated in the study (column three), and the approach used in preparing the task performed by the participating subjects (column four).

In controlled experiments, researchers need to consider limiting the effect of the environmental settings in order to increase the ability to generalise and replicate the study. One observation about the controlled experiments performed as appears in Table 6 is that most of the participants are students. Only four experiments used software engineers from the industry (PS5, PS11, PS15, and PS43). As discussed by Kitchenham [33], subjects must be representative of the population or you cannot draw conclusions from the experiment. However, it is very hard to perform experiments using subjects from the industry. In the primary studies that used software engineers, these were actually part-time graduate students. Therefore, the authors could reach them and convince them to participate. As for case studies, most controlled experiments used programmes in Java. Moreover, the authors tend to perform the required tasks on open source software (OSS) such as JHotDraw, which was used in seven experiments (PS14, PS17, PS26, PS27, PS29, PS37, and PS43).

4.2.4 Surveys: Robson [34] defined surveys as ‘collection of standardised information from a specific population’. Usually surveys are performed by the means of questionnaires and interviews. The survey was used in primary study PS20. Also, a post experiment survey was used in primary studies PS27, PS29, and PS43.

Surveys in software engineering suffer from the problem of identifying the population from which the subjects are selected [33]. As stated by Kitchenham *et al.* [33]: ‘the absence of a scientific sampling method means that the results cannot be generalised to any defined population’.

4.3 Design patterns evaluated in the primary studies

We counted the number of primary studies that evaluated or studies each of the GoF design patterns. The results for these evaluations are given in Figs. 4–6, for creational, structural, and behavioural design patterns, respectively. We also give in Tables 7–9, the number of evaluations and the primary studies that evaluated each design pattern (Table 7 for creational patterns, Table 8 for structural patterns, and Table 9 for behavioural patterns).

Our findings show that all of the 23 GoF design patterns were evaluated in the primary studies but with varying rates. The factory method and singleton are the most evaluated creational patterns.

Among the structural patterns, composite, decorator, and adapter are the most evaluated ones, while observer, state, command, and strategy are the most evaluated behavioural patterns. Among all patterns, observer, state, factory method, composite, decorator, and strategy are the most evaluated patterns; their impact on software quality is evaluated in 27 or more of the 50 primary studies. On the other hand, facade, flyweight, chain of responsibility, interpreter,

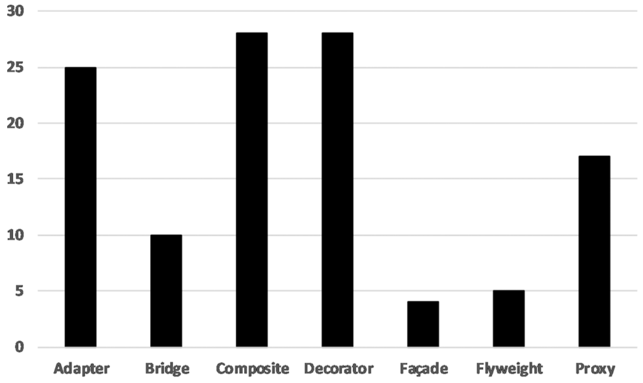

Fig. 5 *Evaluation of structural patterns in primary studies*

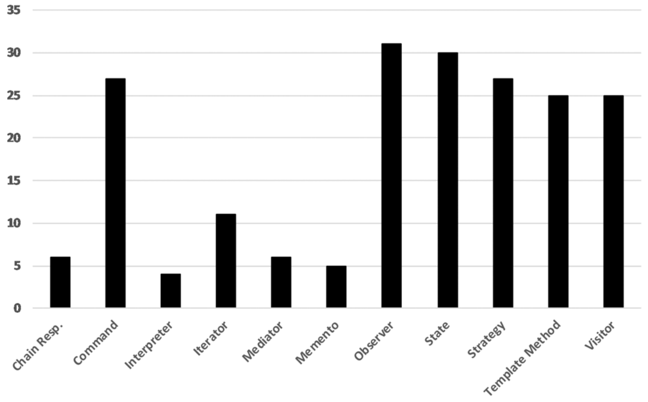

Fig. 6 *Evaluation of behavioural patterns in primary studies*

Table 7 Evaluation of creational patterns in primary studies

Design pattern	No. evaluations	Primary studies
abs. factory	15	PS4, PS5, PS6, PS11, PS15, PS20, PS21, PS22, PS31, PS32, PS35, PS36, PS41, PS44, PS45
builder	11	PS1, PS4, PS6, PS8, PS9, PS22, PS24, PS28, PS36, PS41, PS44
factory method	29	PS3, PS4, PS6, PS9, PS10, PS14, PS15, PS17, PS19, PS21, PS22, PS23, PS24, PS28, PS32, PS33, PS34, PS36, PS37, PS38, PS39, PS40, PS41, PS42, PS44, PS45, PS46, PS48, PS50
prototype	20	PS4, PS6, PS19, PS22, PS23, PS27, PS29, PS31, PS32, PS33, PS34, PS37, PS39, PS40, PS41, PS42, PS44, PS45, PS46, PS48
singleton	23	PS1, PS3, PS4, PS6, PS9, PS10, PS19, PS22, PS23, PS24, PS28, PS31, PS33, PS36, PS38, PS39, PS40, PS41, PS44, PS45, PS46, PS48, PS50

Table 8 Evaluation of structural patterns in primary studies

Design pattern	No. evaluations	Primary studies
adapter	25	PS4, PS6, PS9, PS16, PS19, PS21, PS22, PS23, PS27, PS28, PS29, PS31, PS32, PS33, PS34, PS36, PS38, PS39, PS40, PS41, PS44, PS45, PS46, PS48, PS50
bridge	10	PS2, PS4, PS6, PS8, PS18, PS22, PS24, PS35, PS41, PS44
composite	28	PS1, PS4, PS6, PS7, PS11, PS14, PS16, PS17, PS19, PS20, PS21, PS22, PS23, PS25, PS27, PS29, PS32, PS33, PS34, PS37, PS40, PS41, PS42, PS44, PS45, PS46, PS48
decorator	28	PS4, PS5, PS6, PS10, PS11, PS14, PS16, PS19, PS21, PS22, PS23, PS27, PS29, PS31, PS32, PS33, PS34, PS37, PS39, PS40, PS41, PS42, PS44, PS45, PS46, PS48, PS49, PS50
facade	4	PS4, PS12, PS41, PS45
flyweight	5	PS4, PS6, PS20, PS22, PS41
proxy	17	PS3, PS4, PS6, PS9, PS22, PS24, PS28, PS31, PS33, PS36, PS38, PS39, PS41, PS44, PS45, PS46, PS48

Table 9 Evaluation of behavioural patterns in primary studies

Design pattern	No. evaluations	Primary studies
chain resp.	6	PS1, PS4, PS6, PS16, PS22, PS25
command	27	PS4, PS6, PS9, PS12, PS16, PS17, PS19, PS21, PS22, PS23, PS24, PS27, PS28, PS29, PS30, PS32, PS33, PS34, PS36, PS37, PS40, PS41, PS42, PS44, PS46, PS48, PS50
interpreter	4	PS4, PS6, PS22, PS41
iterator	11	PS3, PS4, PS6, PS9, PS22, PS24, PS28, PS36, PS38, PS41, PS44
mediator	6	PS2, PS4, PS6, PS22, PS41, PS45
memento	5	PS4, PS6, PS22, PS41, PS44
observer	31	PS1, PS4, PS5, PS6, PS7, PS10, PS11, PS14, PS16, PS17, PS18, PS19, PS21, PS22, PS23, PS27, PS29, PS31, PS32, PS33, PS34, PS37, PS39, PS40, PS41, PS42, PS44, PS45, PS46, PS48, PS50
state	30	PS4, PS6, PS9, PS13, PS17, PS18, PS19, PS21, PS22, PS23, PS24, PS25, PS27, PS28, PS29, PS32, PS33, PS34, PS36, PS37, PS38, PS39, PS40, PS41, PS44, PS45, PS46, PS47, PS48, PS50
strategy	27	PS4, PS6, PS9, PS18, PS19, PS21, PS22, PS23, PS24, PS27, PS28, PS29, PS30, PS32, PS33, PS34, PS36, PS37, PS39, PS40, PS41, PS44, PS45, PS46, PS47, PS48, PS50
template M.	25	PS4, PS6, PS7, PS10, PS19, PS22, PS23, PS27, PS29, PS30, PS31, PS32, PS33, PS34, PS36, PS37, PS39, PS40, PS41, PS44, PS45, PS46, PS47, PS48, PS50
visitor	25	PS1, PS2, PS4, PS5, PS6, PS7, PS9, PS11, PS16, PS19, PS22, PS23, PS24, PS26, PS28, PS32, PS33, PS35, PS36, PS38, PS40, PS41, PS44, PS45, PS48

iterator, mediator, and memento are evaluated in six or fewer primary studies.

These results can be explained by the following reasons. First, software that contains documented patterns is used frequently as subject programmes. As shown in Table 5, JHotDraw, which contains a set of documented instances of design patterns, has been used in nine of the case studies in the primary studies. While the number design patterns instances in JHotDraw vary in different versions of the software, design patterns that are evaluated more than others have instances in most of the JHotDraw versions. Even in some controlled experiments, participating subjects were asked to perform tasks on real subject programmes with documented design patterns such as JHotDraw.

Second, design patterns that can be mined from the subject programmes using tools are evaluated more. For example, as we discussed in Section 4.2, Tsantalis *et al.* [28] tool is widely used. The patterns that the tool can detect are among the most evaluated patterns.

Third, the level of training and knowledge of the developers affects the choice of patterns when controlled experiments are performed. Finally, there might be a relationship between the number of evaluations of design patterns and their usage (popularity) in software. However, such a claim needs further investigation.

5 Results

In the following sections, we answer the research questions of this study.

5.1 What confounding factors, practices, or programming constructs effect quality attributes when design patterns are used?

Many factors contribute to the quality of the software. When studying the effect of design patterns on quality, other factors that affect the use of patterns need to be identified. In the following, we discuss the factors, programming constructs, or practices, addressed in the primary studies that influence software quality in the presence of design patterns.

5.1.1 Documentation of design pattern instances: A software document can be described as any artefact which aims at communicating information about the software it describes to readers involved in the software production [35]. Proper documentation of software is one of the oldest practices in software development that is still emphasised due to its importance for software development, understanding, and maintenance [36]. In primary studies PS7, PS27, PS29, and PS43, the effect of documenting design patterns on software quality is evaluated.

In primary studies, PS7, PS27, PS29, and PS43, the effect of documenting design pattern instances on performing maintenance tasks has been evaluated using controlled experiments. Prechelt *et al.* (PS7) performed two controlled experiments and found that providing well-documented programmes, commented with instances of design patterns, significantly decrease the cost of maintenance, in terms of time cost and errors, compared with code with no such comments.

In PS27, Scanniello *et al.* conducted a more recent controlled experiment that confirms the finding of Prechelt *et al.* that maintenance is performed faster with proper documentation of design patterns (which the authors called effort). They also provided a new measurement which they called efficiency, computed by dividing the subject performance by the subject effort. They found that efficiency is also improved when design pattern instances are documented.

In PS29, Gravino *et al.* conducted two controlled experiments to measure the impact of documenting design pattern instances on the ability of subjects to understand a given source code (i.e. source code comprehension). They used two types of documentation, graphical [unified modelling language (UML)], and textual (source code comments). There results do not favour any type of documentation over another. They also found that the ability to understand the code is increased when developers correctly identify design patterns in the code.

In PS43, Scanniello *et al.* extended the experiments in PS28 and PS30 where the controlled experiments are performed with subjects with different levels of experience. Their results, while confirming their previous experiments, also show that experienced developers benefit more from the documentation of design patterns, compared with less experienced developers.

The primary studies' results agree on the positive effect of documenting the source code with information about instances of design patterns on software comprehension and maintenance. Taking into account that primary studies show that it is enough to depend on source code comments about instances of design patterns, therefore, we expect that the effort needed by the developers to add such comments is not high. However, this effort needs to be evaluated in future studies.

Primary studies did not, however, study the effect of the lack of documentation for unintended design patterns, which are instances of design patterns that developers implement in the code unintentionally. Having these patterns instances in the source code without documentation makes it hard to understand the code. One suggestion here is to use a design pattern mining tool in order to find unintended patterns. Needless to say, this suggestion depends on having a tool with high mining precision and low cost. The primary studies discussed the importance of documenting design patterns for maintainability and programme understanding. Future research can study the effect of documentation of design patterns on other quality attributes (e.g. extensibility).

5.1.2 Module size: The size of a software module has been shown by many studies to have a clear effect on many software quality attributes, which is known as the confounding effect of class size. Studies show that large modules [classes in object-oriented programming (OOP)] tend to be hard to maintain, test, and error-prone. El Emam *et al.* [37] examined confounding effect of class size in the validation of object-oriented (OO) metrics, mainly the C&K metrics [38]. Their results confirm the confounding effect. In many primary studies, we found that the class size is admitted as a threat to validity. In primary studies PS3, PS9, PS10, PS32, and PS48, the effect of size of classes participating in design patterns is evaluated. While in PS44, the class size is used to normalise the metrics used in order to eliminate its effect.

In PS3, Bieman *et al.* performed a case study to evaluate whether there is a relationship between the use of design patterns and the number of changes in evolving versions of industrial software. While the goal of the study is to evaluate classes that participate in design patterns, they found a strong relationship between the class size and change proneness, where larger classes changed more frequently. For the effect of design patterns, they found that classes that participate in design patterns are among the most change prone classes in the software.

In PS9, Bieman *et al.* studied five systems to evaluate the effects of the use of design patterns on changes that occur as the systems evolve. They found that classes that participate in design patterns [pattern classes (PCs) for short] are change prone as other classes in four out of five systems. PCs in one of the systems were less change prone. In this study, the authors reported after normalising the effect class size. Moreover, they found that larger classes are more change prone to two of the five systems.

In PS10, Vokáč performed a case study to evaluate the relationship between the use of design patterns and number of defects in the code. The author compared defect rates for classes that participated in selected design patterns to the code at large. They found that there are significant differences in defect rates among the patterns. Vokáč analysed the effect of class size as a confounding factor. The results show that the size effect is significantly correlated to defect frequency. However, no certain results for the correlation between certain patterns and class size (i.e. if the use of a pattern requires larger participant classes).

In PS32, Posnett *et al.* studied the effect of the size of pattern classes and classes playing a metapattern role on change proneness. The concept of a metapattern was introduced by Pree [39], which aims at capturing the pure structure of design patterns. Structurally similar design patterns instantiate the same metapatterns. Posnett *et al.* found that size explains more of the variance in change-proneness than either design pattern or metapattern roles. They also found that both design patterns and metapattern roles were strong determinants of size. Therefore, size is a stronger determinant of change proneness than either design pattern or metapattern roles. Differences in change proneness between roles can be better explained by differences in the sizes of classes playing these roles.

In PS44, Elish and Mohammed measured and compared the fault density of PCs with non-PCs (NPCs). The fault density of class is a measure of the number of faults in the class divided by the class size. The authors' aim at normalising the number of faults by the class size is to reduce the confounding effect of class size.

In PS48, Hussain *et al.* performed a case study to investigate the correlation between the frequent use of design patterns and quality attributes. The authors studied the effect of the software size, measured in a number of classes, on the correlation. Their results support the existence of the size effect as a confounding factor.

Since the class size effect on software metrics is widely accepted, it is necessary to evaluate whether the use of certain design patterns has an effect on the class size (i.e. requires increasing or decreasing the size of participating classes). This can provide better understanding of the conflicting results of evaluating design patterns. In PS28 and PS36, the authors mentioned that some design patterns (e.g. singleton) increase the class size. However, this is an observation that needs to be empirically supported.

It is widely agreed in the software engineering community that large modules have a harmful effect on software quality. The primary studies discussed here support that size is a confounding factor that should be considered. However, the effect of the use of design patterns on module size is still not clear. Whether classes become larger because of the use of a design pattern or due to other factors (e.g. the task that a pattern performs) is an important issue that needs further research.

5.1.3 Design patterns as crosscutting concerns: The software can contain two types of concerns: core, which is the main behaviour that is needed by the software, and crosscutting, which refer to a behaviour that is common to multiple system core modules [40]. With the lack of a structure that modules a crosscutting concern in OOP, a crosscutting concern implementation is scattered through core concerns modules, resulting in a tangled code for the core concerns. In other words, the implementation of a crosscutting concern decreases modularity, which in turn, negatively impacts software quality. Primary studies PS6, PS16, PS22, PS23, and PS41 discusses the degree of scattering (DOS) of a design pattern and how a design pattern modularity can be increased.

In PS15 and PS22, Aversano *et al.* stated that the introduction of some design patterns in the code might result in scattering the code of the classes that interact with the design patterns (i.e. the clients and the design patterns). When the software evolves, the presence of such a scattered code might make the software hard to change and introduce faults.

In PS16, Aversano *et al.* empirically investigated the relationships between the evolution of design patterns and the evolution of crosscutting concerns they induce. They identified crosscutting concerns containing invocation of methods belonging to design PCs. Their results indicate a consistent change of crosscutting with the pattern.

In PS23, Aversano *et al.* studied whether there is a relationship between the scattering degree of the crosscutting concern and fault proneness for crosscutting concerns induced by design patterns. The DOS describes how the code of concern is distributed among elements (classes or methods) [41]. Aversano *et al.* identified the DOS in the context of a design pattern over its clients, i.e. the number of callers spread among different classes with respect to the number of callers for classes participating in a design pattern. Aversano *et al.* metric is different from the DOS metric created by Eaddy *et al.* [42] in that the latter is a measure of the variance of the concentration of a concern (how many of the source lines related to a concern are contained within a specific component) over all components with respect to the worst case.

The results of PS16 show that a pattern induced crosscutting concern scattering degree can have a moderate to high correlation in the presence of defects. They also found that the scattering degree is correlated to the fault proneness of the design pattern code itself.

The results of primary studies PS16 and PS23 can be interpreted in many ways, mainly, the results show that not all design patterns induce crosscutting concerns and when that happens, then the problem is not related to the design pattern but to the existence of scattered code in PP programmes. Eddy *et al.* [41, 42] study shows that there is a significant correlation between the scattering degree of any crosscutting concern and fault proneness. They also found that most of the concerns are crosscutting (95%) with various degrees, which suggests a potential need for improving modularity. In other words, crosscutting concerns are the norm in OO programmes and not the exception.

Modularisation of crosscutting concerns can be obtained using aspect-oriented programming (AOP). In AOP, a crosscutting concern can be modularised using a construct called *Aspect*. AOP is common in many industrial frameworks such as JBoss and Spring Framework [43]. Most of the common programming languages have an extension that supports developing AOPs (e.g. AspectJ [44] for Java, AspectC++ [44] for C/C++).

In PS6, Hannemann and Kiczales developed AspectJ implementations of the 23 GoF design patterns. For each of the 23 GoF patterns, they developed a representative example that makes use of the pattern and implemented the example in both Java and AspectJ. Hannemann and Kiczales' study shows that using AspectJ improves the implementation of many GoF patterns, in terms of modularisation. In particular, design patterns with crosscutting concerns got the most improvement. Another result the authors obtained is that by implementing a pattern in an aspect, the pattern implementation becomes reusable. The last result applies to half of the GoF patterns.

Primary study PS6 quantified modularity using the four attributes of locality, reusability (of the pattern code), composition transparency, and (un)pluggability.

In PS22 and PS41, two case studies were conducted to evaluate the AOP implementation of design patterns, compared with OOP implementation. In PS22, Garcia *et al.* performed a study comparing the AOP implementation of design patterns developed by Hannemann and Kiczales (PS6) with OO implementation. Garcia *et al.* used the implementation from Hannemann and Kiczales' study and added some implementations they developed for the sake of the study. The comparison was performed using the following attributes: separation of concerns, coupling, cohesion, and size. They used ten metrics for measurements. The results show that for most patterns, aspect-oriented (AO) implementations improved separation of concerns, reduced coupling, increased cohesion, and reduced size of code.

In PS41, Cachoa *et al.* investigated how AOP can help to reduce the complexity involved in composing design patterns, i.e. when a class or a method that plays a role in a particular pattern plays another role in a different pattern. In this case, the design patterns implementations become tangled with each other and with the core responsibilities their implementation becomes tangled with each other and with the core responsibilities of the patterns client classes. They used three software originally implemented in Java and reengineered the existing Java implementations to produce their AO versions. They used Hannemann and Kiczales (PS6) AspectJ implementations of the design patterns in the AO versions. They measured pattern modularity with four attributes, these are a separation of concerns, coupling, cohesion, and size. Their results were in favour of the AO implementations in most of the cases.

The modularisation of design patterns using AOP is promising as the primary studies suggest. Providing an off-the-shelf version of design patterns can promote reusability and ease of use of design patterns. However, there are some technical issues related to AOP that need to be handled in order to ease software development via AO programming languages. For example, in PS41, Cachoa *et al.* compared the design patterns implementations in AspectJ (the deFacto AO compiler for Java) with implementations in another AO complier called Compose* [45]. Their results show that Compose* implementation provide either similar or better separation of concerns, compared with AspectJ. AOP is not new anymore, research in AOP need to solve issues that prevent the wider adoption of the paradigm. Tanter *et al.* [46] presented several technical issues associated with the current state of affairs of aspect languages that need to be handled. Legar and Fukuda [47] asked the question of why developers do not take advantage of the progress in modularity (including AOP), despite modularity clear benefits. While the authors intend to answer the question in future work, we think, based on the discussed primary studies, a solution that allows modularisation of design patterns, either using aspects or another approach, can promote software quality.

5.2 What quality attributes are evaluated, what is measured, and what metrics are used?

Results reported by the primary studies are obtained empirically using experiments with various settings and evaluation approaches. Here, in an effort to provide a better understanding of how these results compare to each other and yet to accumulatively build on these results, we raise the following questions:

- What quality attributes have been measured, are these attributes measured directly or internal quality attributes are used as surrogates?
- What metrics are used for evaluating quality attributes?
- What software artefacts are measured using the chosen metrics? Even when the same metrics are used, empirical studies apply these metrics on different constructs or design motifs (e.g. classes that participate in design patterns, client classes, NPCs, subsystems or libraries that contain design patterns).
- What design patterns are evaluated in the primary studies? Patterns differ in their design and implementation and consequently can have different influences on certain quality attributes.

In order to answer these questions, we summarised the setting and evaluation approaches used in the primary studies in Table 10, for case studies, and Table 11 for controlled experiments. In Table 10, for each primary study in column one, we give in column two, the quality attribute and the surrogate attribute in parentheses

Table 10 Quality attributes, artefacts, and metrics used in case studies

Primary study	Quality attribute (surrogate)	Artefacts	Metrics
PS3	maintainability (change proneness)	PC, NPC	no. changes in each class in each version of the system
PS9	maintainability (change proneness)	PC, NPC	no. changes in each class in each version of the system
PS10	maintainability (defect frequency)	PC, NPC	no. defects in each class
PS12	performance (throughput, reliability)	AllC	no. requests served within a specified unit of time time a client sending a request has to wait for the response no. correctly served requests no. requests processed within a threshold time value
PS18	maintainability (size, complexity, coupling, cohesion)	PC, AllC	lines of code (LoC), no. classes (NoC) attribute complexity, weighted methods per class 1, weighted methods per class 2 coupling factor, cohesion: lack of cohesion of methods
PS19	maintainability (change proneness)	PC, CoPC	no. snapshots (committed changes) where PC changed/no. snapshots LoC co-changed in NPCs when a PC changed
PS21	maintainability (change proneness)	PC	no. changes in classes with different roles in a design pattern
PS24	maintainability (change proneness)	RoPC, NPC	no. changes committed to a class
PS28	maintainability (fault proneness)	PC, NPC	no. faults per class LoC changed (either added, deleted or modified) to repair a fault
PS31	maintainability (fault proneness)	AllC	defect frequency (no. open bugs), debugging frequency (no. bugs fixed/ bugs opened)
PS32	maintainability (change proneness)	RoPC	number of changes (unique commits) to a class in each release
PS33	reusability	PC, PaC	class reusability calculated according to the QMOOD [48] corresponding metric average reusability of all classes that participate in a component
PS34	maintainability	AllC	an absolute measure of maintainability defined in [49]
PS36	maintainability (fault proneness)	PC PaC	number of faults per class
PS39	effectiveness, extendibility, flexibility, functionality, reusability, understandability	standalone, libraries	QMOOD [48] corresponding metric for each measured quality attribute
PS40	maintainability (change proneness)	PC,CoPC	number of snapshots where PC changed divided by the number of snapshots LoC in NPC co-changed when a PC changed
PS42	maintainability (fault proneness, change proneness)	PC, StPC, CoPC	number of faults per class number of changes per class
PS44	maintainability (fault proneness)	PC, NPC	fault density – no. faults per class/class size (LoC)
PS45	maintainability (stability)	PC, NPC	ripple effects measurement (REM) as defined in [50]
PS47	performance (energy consumption)	AllC	energy consumption (in joules)
PS48	effectiveness, extendibility, flexibility, functionality, reusability, understandability	AllC	QMOOD [48] corresponding metric for each measured quality attribute
PS50	performance, correctness, security	PC	number of violations of rules of good coding practices according to a tool called FindBugs (http://findbugs.sourceforge.net/)

measured by the study. In column three, we give the module(s) used for measurements, while in column four, we give the metrics used. Table 11 is organised in the same way except for the modules, which does not apply for the metrics used in controlled experiments. The following abbreviations are used in both tables: PCs, NPCs, classes that co-change with PCs (CoPCs), classes that have static relationships with PCs (StPCs), classes of a certain rule in a design pattern (RoPC), package classes (PaC), and all classes in the software (AllC).

Evaluation of software quality requires relating measurements of software artefacts to external quality attributes or to one or more of the attribute characteristics. Models for software quality have been researched for decades and a large number of such models have been proposed. In the primary studies, researchers did not always specify which quality model they follow, although this is important, especially when an internal attribute is used as a surrogate for an external attribute. From Table 10, we can clearly conclude that maintainability is the most evaluated external quality attribute. We can also note that the internal quality attributes evaluated are mapped to maintainability using various quality models.

In the following, we discuss results for each of the quality attributes.

5.2.1 Maintainability: Most of the primary studies evaluated design patterns in the context of software maintenance and evolution. This trend can be explained by the fact that the benefits (if any) related to the use of design patterns become clear when the software is evolved and maintained. For example, the flexibility provided by design patterns becomes clear when the software is extended. On the other hand, the effect of complexity added due to the use of design patterns, if any, also becomes clear when the software is extended or maintained. In controlled experiments, which require performing a task and measure subjects' performance, it is easier to perform the experiment in the form of a maintenance task on an existing code due to time and cost limitations of the study.

ISO-9126 [51] defined maintainability as ‘software quality characteristic concerning the effort needed to make specified modifications to an already implemented system’. ISO-9126 decomposes maintainability into four characteristics: analysability, changeability, stability, and testability. The more recent ISO/IEC 25010 [52] quality model rearranged maintainability characteristics into modularity, reusability, analysability, modifiability, and testability. In the primary case studies, the authors evaluated maintainability using the following attributes:

(1) *Change proneness*. The attribute was evaluated by eight of the case studies. These are PS3, PS9, PS19, PS21, PS24, PS32, PS40, and PS42. Except in case studies PS19 and PS40, change proneness was measured by counting the number of changes to a class. A change is counted as a single change regardless of the changing size, importance, or required effort. In case studies PS19 and PS40, change is normalised by dividing the number of committed changes (snapshots) that involve PCs by the number of snapshots. Moreover, the number of lines of code that co-changed in NPCs is counted.

The change proneness metric was measured on PCs and NPCs in order to find whether the use of design patterns requires more or fewer changes (PS3, PS9, PS21, and PS24). In PS32 and PS21, the metric is applied to classes with certain roles in a design pattern in order to find whether the role a class plays in a design pattern influences the change proneness of the class. The change proneness metric is applied to PCs and classes that co-changes when a PC is changed in primary studies PS19, PS40, and PS42.

Changeability is considered as one of the characteristics of maintainability according to the ISO-9126 quality model [51], in which changeability is defined as ‘attributes of software that relate to the effort needed for modification, fault removal or for environmental change’. Change proneness is a measure of frequent change, while changeability is concerned with the effort of change. Although the two concepts are related, a clear difference exists. The question here becomes, what does change proneness tell us about the class? Or why classes change?

Table 11 Quality attributes, metrics, and constructs evaluated in controlled experiments

Primary study	Quality attribute (surrogate)	Metrics
PS1	programme comprehension (team communication)	episodes of explanation by one member and balanced discussions during work on a maintenance task
PS5	maintainability	time to perform a maintenance task no. faults in a maintenance task
PS7	maintainability (program comprehension)	time to perform a maintenance task no. faults in a maintenance task
PS11	maintainability	time to perform a maintenance task no. faults in a maintenance task
PS14	maintainability	time to perform a maintenance task no. failed test cases
PS17	maintainability	percentage of maintainers utilising relevant design patterns no. faults in a maintenance task
PS25	maintainability (understandability, modifiability)	using a questionnaire on a design and modifications performed by subjects on the design: understandability – time subjects spent on answering questions, no. correct answers modifiability – time to perform modifications, score (correctness) of modifications
PS27	maintainability	using a questionnaire on a maintenance task: effort-time to answer the questions efficiency – correctness (precision and recall) divided by effort
PS29	maintainability (program comprehension)	using a questionnaire on the comprehension of a maintenance task: effort – time to answer the questions comprehension – a quantitative measure based on precision and recall efficiency – correctness (precision and recall) divided by effort
PS43	maintainability (program comprehension)	using a questionnaire on the comprehension of a maintenance task on source code: correctness of understanding – correct answers for questions on a comprehension task time – time required to accomplish a comprehension task confidence – subjective confidence subjects display regarding their own understanding perceived difficulty – subjective judgment on ease of obtaining information using the provided artefacts

Changes performed in a class can be corrective, adaptive, perfective, or preventive [53]. These changes can occur due to new requirements, debugging, changes that propagate from changes in other classes and refactoring [50]. A class that changes more frequent than others might be because the class is easy to extend or because the class is correlated to other classes, which raises an alarm about the class modularity. In primary studies PS3, PS9, PS24, PS32, and PS42, changes were not classified. Primary studies PS19, PS21, and PS40 examined types of changes in order to find whether the role of a class in a design pattern is related to certain types of changes. However, the primary studies did not investigate why these changes occur.

Another concern rises from ignoring the size of change. The size can be a proxy of the effort needed to perform the change (maintenance) task.

(2) *Fault proneness, defect frequency or fault density*. It was evaluated in case studies PS10, PS28, PS31, PS36, PS42, and PS44. Faults information was obtained from the bug reports and change logs of the studied systems. In case studies PS10, PS36, and PS42, fault proneness was measured by counting the number of faults per class, comparing PC with NPCs. In PS28, the authors measured the effort needed to fix a fault by considering the number of lines of code changed to fix a fault (added, deleted or modified). In PS31, the authors counted the number of open bugs as a measure of defect frequency and the ratio (no. bugs fixed/no. bugs opened) as a measure of debugging frequency. The authors of PS44 measured fault density of a class by dividing the number of faults found by the class size in order to reduce the effect of the class size as a confounding factor.

The results of the primary studies regarding fault proneness do not show a clear agreement. Primary studies PS28 and PS36 show that PC is more fault prone than NPCs. Primary studies PS28 and PS36 found that some PCs are more or less fault prone than NPCs (e.g. structural patterns classes are less fault prone), depending on which pattern they belong to. Primary study PS42 did not find clear tendency. In PS44, the authors found that classes that have static and co-change dependency with PCs are more fault prone than other classes.

At this end, we need to ask ourselves here ‘what do these results tell us?’ There is no firm answer on whether PCs are more fault prone. Even if some studies can show that still, we need to know why and when this happens. There are many factors that can determine the fault proneness of a class including class responsibility or dependency relations. The developers' level of experience, lack of proper documentation of the software, or lack of maintainers understanding of the software are main factors that can lead to few or more faults. In the end, we have no advice to software engineers on when to use design patterns to avoid producing software with more faults.

(3) *Stability*. In PS45, Ampatzoglou *et al.* evaluated stability of classes that participate in design patterns. According to ISO-9126 [51], stability refers to the ‘attributes of software that relate to the risk of unexpected effect of modifications’. The standard also considers stability as one of the maintainability subfactors. To measure stability, Ampatzoglou *et al.* defined a metric called the ripple effect measurement (REM), which attempts to quantify the probability of a change occurring in class to be propagated to a dependent class. Their results show that design PCs are more stable than NPCs, depending on the class role in the pattern.

(4) *Size, complexity, coupling, and cohesion*. In PS18, Ampatzoglou and Chatzigeorgiou evaluated the effect of using design patterns on maintainability of games software. The authors calculated four categories software metrics from the Chidamber and Kemerer (C&K) metrics suite [38]: size, complexity, coupling, and cohesion. The results of the study were in favour of using design patterns. Mainly for coupling, cohesion, and complexity.

In the study performed by Jabangwe *et al.* [18] that aims at finding a link between OO metrics and external quality attributes, the authors found that it is a common practice to link OO measures and proxies for maintainability. Moreover, they found that the C&K measurement suite was the most popular across studies. While such a trend is not followed by the primary studies that we investigated, except for PS18.

In the controlled experiments included in our study, researchers performed the study by preparing a maintenance task on existing software. The subjects are then asked to perform the maintenance task on the software, and/or are asked to answer a questionnaire about the task. For evaluation, the following metrics are used in four experiments (PS5, PS7, PS11, PS14, PS25):

- Time to perform the maintenance task.
- The number of fault subjects introduced to the code when performing the maintenance task.

In these experiments, the aim was to measure the effect of using design patterns on maintenance. To do so, the authors prepared two versions of each programme, with and without design patterns. Then two groups of subjects, a group for each version of the programme, are asked to perform the maintenance task. Time to perform the task is recorded, and the maintenance code is inspected for faults, except in PS14, where the authors ran test cases and counted the failed test cases as a measure of the number of faults.

The authors of PS25, PS27, PS29, and PS43 used a questionnaire about how to perform a maintenance task on a part of a software that uses design patterns. Subjects’ answers are then evaluated in terms of time spent to answer the questions, as a measure of understandability, and the correctness of the answers, as a measure of modifiability.

Results obtained by controlled experiments are contradictive. The results of PS5 show that the same pattern can have a different effect on maintainability depending on the programme complexity or domain. PS11 results show that some patterns (decorator, observer) promote maintainability, while other patterns have a negative impact (composite, visitor). PS14 results show that design patterns have a positive impact on maintainability, while PS25 results show that the use of design patterns decreases understandability and no significant effect on modifiability.

In the remaining controlled experiments, primary studies PS27, PS29, and PS43 evaluated the effect of documentation on maintaining programmes that use design patterns, while PS1 found that the use of design patterns can improve team communication which improves programme understandability.

5.2.2 Performance:

Performance is a well-known external quality attribute, which can be measured, according to ISO-9126 [51] and ISO/IEC 25010 [52] quality models, by time behaviour (response time), and resource utilisation.

The study of Rudzki in PS12 evaluated two quality attributes, throughput, measured by the number of requests served per unit of time, and reliability, measured by the number of correctly served requests. The author also measured response time for a request. The study of Rudzki aims at finding whether the changes that design patterns introduce to the software structure can have an effect on software performance. The author evaluated two design patterns, command and facade, and found that in most of the cases, facade outperformed command design pattern.

In PS47, Feitosa *et al.* investigated the effect of using design patterns on performance by measuring energy consumption. They compared pattern participating methods of energy consumption to the consumption of functionally equivalent alternatives non-pattern solutions. Their results show that for most of the cases the alternative solutions are better, in terms of energy consumption. There results also show that for larger and complex modules (methods), the design pattern solution consumes less energy.

The results reported in PS47 are important and should lead to more studies due to the increasing importance of energy efficient software, especially for portable devices. In conclusion, there is a shortage of studies that link software performance to the use of design patterns. The reason might be that increasing performance is not among the promised advantages of using design patterns. Theoretically, however, since design patterns increase flexibility and reusability, which can (or cannot) be at the price of decreasing performance. Therefore, there is a need to empirically investigate this issue.

5.2.3 Quality attributes from QMOOD quality model: In primary studies PS33, PS39, and PS48, the authors used the quality model for object-oriented design (QMOOD) [48] metrics suite to evaluate the effectiveness of using design patterns. The QMOOD model describes a set of measures that aims at enabling quantitative quality evaluation of internal characteristics that are unique to OO designs. The measures are obtainable during the design phases, and, therefore, can be used to identify design flaws early in the software development cycle.

The QMOOD defined six external properties: effectiveness, extendibility, flexibility, functionality, reusability, and understandability. These are linked to design (internal) properties for which the model provides corresponding OO metrics [48].

In PS33, Ampatzoglou *et al.* investigated reusability in software packages. According to QMOOD, class reusability is calculated as follows:

$$\begin{aligned}\text{reusability} = {} & \frac{1}{4}\text{cohesion} + 0.5 \times \text{messaging} \\ & + \frac{1}{2}\text{size} - \frac{1}{4}\text{coupling}\end{aligned}$$

The authors computed the average reusability of design PCs and compared that with the average reusability of the software package. The results suggest that each pattern has a different level of reusability.

In PS39, Sfetsos *et al.* evaluated the six quality attributes suggested by QMOOD. These are effectiveness, extendibility, flexibility, functionality, reusability, and understandability. The authors evaluated quality in software libraries and in standalone applications. The results, as described by the authors, are mixed, as they show that some design patterns have a stronger effect on quality in standalone applications than in libraries while some other patterns have an opposite influence. There was no explanation of this.

In PS48, Hussain *et al.* evaluated the six quality attributes suggested by QMOOD. The authors also studied the effect of size as a confounding factor, and the effect of design patterns on software evolvability (how changing the number of patterns instances in subsequent releases affects quality attributes). Their results show that generally, design patterns improve reusability and flexibility attributes.

5.3 What are the common threats to validity?

Validity is concerned with the credibility of results obtained by an empirical study. As pointed out by Yin [54], and Runeson and Martin [25], there are four important aspects of validity, these are: (i) construct validity, (ii) internal validity, (iii) external validity, and (iv) reliability. Based on the threats to validity reported in the primary studies, we identify the following main validity concerns for empirical studies on evaluating the effect of using design patterns on software quality.

5.3.1 Construct validity: Construct validity refers to the meaningfulness of measurements [55]. Do the measurements actually measure what they claim? The following common threats were reported in the primary studies:

- *Accuracy of pattern mining approach*: many tools are available that can mine design patterns from the design, source code, or the executable code. However, all these tools have limitations and can miss some patterns (i.e. false negatives). The same issue occurs when mining for patterns is performed manually. When design patterns in the studied programmes are documented (e.g. JHotDraw), unintentional patterns are excluded (i.e. pattern realisations that occur in the code unintentionally). Pattern mining concerns are discussed in the following primary studies: PS10, PS16, PS19, PS21, PS23, PS28, PS34, PS38, PS42, and PS50. One way to limit this threat is to use the two mining approaches, manual and using a tool. However, it is hard to perform manual inspections when the subject programmes are large. Therefore, improving the accuracy of design pattern mining tools is important to limit this threat.
- *Assessment of changeability*: in the primary studies, the changeability of design patterns is assessed by measuring changes in classes that play a RoPC. However, changes that occur in a class playing a role in a pattern might not necessary be related to the use of the pattern. The class might provide services to clients that require frequent changes, whether a pattern is used or not. This threat was reported in primary studies: PS21, PS24, and PS28. Another changeability assessment concern is treating all changes equally. As reported by Bieman *et al.* in PS9, some changes require more effort than others. In PS3 and PS9, Bieman *et al.* stated that a large number of changes over many versions should minimise the impact of change effort variability. In PS19 and PS23, Aversano *et al.* reported a threat related to the assessment of change impact (change in classes that interact with patterns). They used change sets to determine the impact of such changes in PCs. Change sets represent changes performed by a developer in terms of added, deleted, and altered source code lines.
- *Social threats (or social desirability bias)*: this refers to the tendency of individuals to present themselves in the most favourable manner [56]. These threats appear in controlled experiments and questionnaires. The threats were reported in primary studies PS26, PS27, and PS29.
- Transformation or preparing alternative solutions may be error-prone due to their complexity. This threat can occur in studies that compare non-pattern solutions with pattern solutions, where the authors have to prepare either solution. The threat was reported in PS47.
- The mapping between quality attributes. This threat can be minimised by using validated metrics. The threat was reported in PS49.

5.3.2 Internal validity: Internal validity is concerned with the cause and effect relationships, where certain conditions are believed to lead to other conditions [54]. In experiments, the following common threats are reported in the primary studies:

- Plagiarism, which occurs in controlled experiments when subjects communicate with each other to solve a task or pass information about the task to other groups: the threat was reported in primary studies PS7, PS11, PS13, PS14, PS17, PS25, and PS27.
- Mortality, which refers to having some subjects giving up on a task before completing it: this threat was reported in primary studies PS11, PS14, and PS25.
- Learning and fatigue effect: this threat was reported in primary studies PS11, PS25, PS26, PS27, and PS29.
- Differences between subject groups (group balancing): this threat was reported in primary studies PS5 and PS11.
- Incorrect time measurement: this threat was reported in primary studies PS5 and PS7.
- Communication and team work skills of subjects. Reported in primary study PS1.
- The accuracy of the metrics in measuring the quality attributes. Reported in primary study PS39.

In the case studies, primary studies PS3, PS23, PS38, PS42, and PS47, reported that it is not enough to illustrate statistically significant relationships. It is also important to show temporal precedence, evidence that the cause occurs before the result. To conclude, researchers need to report any confounding factors that might influence the studied relation between dependent and independent variables.

5.3.3 External validity: External validity is concerned with the ability to generalise results [54]. The following common threats were reported in the primary studies:

- In case studies, a generalisation of results can be affected by the programming language of the subject programmes, the domain of the subject programmes, the number of subject programmes, the sizes of the subject programmes, and the number of realisations (instances) of design patterns in subject programmes. These threats were reported in the following primary studies PS3, PS4, PS9, PS10, PS14, PS15, PS16, PS17, PS19, PS21, PS23, PS24, PS28, PS29, PS33, PS34, PS35, PS38, PS39, PS42, PS47, PS48, and PS50. In PS34, Hegedüs *et al.* reported that a small number of changes can affect the evaluation of changeability in the results of their study. Ng *et al.* also discussed the effect of the number and type of changes on external validity in PS17.

- In controlled experiments and surveys, generalisation of results is affected by the number of participants, the experience level of participants (all of them are either students or senior software engineers), the size and complexity of the tasks, the suitability of the task, the effect of team work since realistic programming is a team work, programme and task representativeness, and maintenance situations. These threats were reported in the following primary studies: PS1, PS5, PS7, PS11, PS13, PS15, PS20, PS25, PS26, and PS27.

5.3.4 Reliability: Reliability is concerned with the ability to replicate the study, where the operations of a study can be repeated, with the same results [54]. To facilitate replication of empirical studies, data sets, the analysis approach, the tools used, the source code of the programmes (whether subject programmes in case studies or tasks in experiments), and the design pattern detection tool used (if any), need to be publicly available. Moreover, the details of the statistical analysis performed need to be described and row data must be available to allow replicating statistical analyses. Reliability concerns were discussed in the following primary studies: PS14, PS19, PS21, PS23, PS42, PS47, PS49, and PS50.

6 Threats to validity

In this section, we discuss possible threats to the validity of our study. We identified three types of threats to our study, these are: construct validity, internal validity, and external validity.

Construct validity refers to the meaningfulness of measurements, to what extend the measurements represent what is investigated [25, 55]. In a literature review, construct validity threats are related to the identification of primary studies including the following factors: inappropriate or incomplete search terms in automatic search, incorrect search method, incomprehensible venues or database, inappropriate inclusion and exclusion criteria, or lack of expert evaluation [57]. In our study, we performed the search on the candidate article metadata including the article title, abstract, and keywords. Articles that do not mention the phrase ‘design pattern’ in their metadata might be omitted. However, articles that evaluate design patterns would state that either in the article title or on the abstract and keywords. We searched five digital libraries that cover high-quality publications in software engineering. The inclusion and exclusion criteria that we used ensures that only quality articles are included (at least indexed by SCOPUS). We further assessed the list of articles manually by meetings and discussions between the authors using predefined quality factors. We can be fairly confident that we had identified the most relevant articles related to our study.

Internal validity is concerned with the causal relationship, where certain independent variables lead to other dependent variables [25, 55]. In a literature review, internal validity threats are related to data extraction [57, 58]. We extracted data from the primary studies in a two-phase process carried out by the first and second authors where each had extracted the data from the paper and compared the results with each other. Sometimes we faced difficulties in classifying data because primary studies might use different names for the same quality attribute or metrics. In such cases, we depend on the details provided by the primary study to properly classify the results.

External validity is concerned with the generalisation of the study findings. The generalisability of this review is limited by the generalisability of the included primary studies [59].

7 Conclusions and future work

In this study, we reported the results of a SLR analysing the current state of research on the effect of using design patterns on software quality. We identified confounding factors that can affect software quality, which was studied in previous research or recognised as a threat to validity. Our study has shown that the primary studies provide empirical evidence on the positive effect of documentation of designs pattern instances on programme comprehension, and therefore, maintainability. While this result is not surprising, it has, however, two indications. First, developers should pay more effort to add such documentation, even if in the form of simple comments in the source code. Second, when comparing results of different studies, the effect of documentation has to be considered. Moreover, the cost of documentation needs to be counted. Unintended design patterns need also to be considered. One way to find these patterns is to use a pattern mining tool.

The size of a software module, known as the confounding effect of class size in OO software, is agreed upon to affect many software quality attributes. However, the effect of using design patterns on modules, whether classes or sometimes methods, needs further investigation. Some design patterns can affect the size of participating classes, either requiring smaller classes or requiring classes with larger sizes and larger methods. Furthermore, the pattern responsibility needs also to be considered as design patterns might perform tasks that increase the size of participating classes. Studying these relations can provide a better understanding of the conflicting results of evaluating design patterns and guide researchers and practitioners on how to evaluate and deploy design patterns.

Modularity and separation of concerns are key concepts in OO software. When using design patterns, providing a modular perceptive of a pattern can ease maintaining and extending patterns. From the primary studies, we can identify two main issues when studying modularity of design patterns. First, it is not clear how to limit the design pattern into a module. There are many dependencies between design PCs and other classes. Considering participating classes only neglects such dependencies. Second, what metrics should be used to measure modularity should be decided. As for measuring quality attributes, various metrics are available.

Some of the primary studies suggested modularising design patterns using AOP. The results obtained from these studies are promising (PS6, PS22, and PS41). However, there are some issues related to AOP that need to be handled in order to ease software development via AO paradigm. Research in AOP needs to solve technical issues that prevent the wider adoption of the paradigm. However, whether using AOP or another approach, we recommend studying how to improve modularity of design patterns, since providing modular solutions for patterns can promote software quality and ease maintainability.

Maintainability is the most evaluated external quality attribute. Internal quality attributes, mainly change proneness, fault proneness, and stability, are mapped to maintainability using various quality models. However, comparison of results is hard due to the use of different metrics and measuring a variety of artefacts (e.g. PCs versus NPCs, PCs versus system classes or PCs and their clients versus NPCs). Additionally, the results of primary studies regarding change proneness and fault proneness are contradictive with fewer explanations for why such results are obtained. In controlled experiments, our findings confirm with the results of Riaz *et al.* [13], which shows that there are subtle differences in the studies design which limit comparison of the results.

The results of our study highlight the current state of evaluating the impact of using design patterns on software quality. We hope that our findings and recommendations will provide an important guide for future studies.

8 Acknowledgments

The authors would like to thank the editors and the anonymous reviewers for their valuable comments and suggestions. We also sincerely thank professor Sudipto Ghosh from the Department of Computer Science at Colorado State University for his insightful advice and many valuable suggestions to improve the quality of this paper.

9 References

[1] Alexander, C., Ishikawa, S., Silverstein, M.: '*A pattern language: towns, buildings, construction*', vol. **2** (Oxford University Press, New York, NY, USA, 1977)

[2] Gamma, E., Helm, R., Johnson, R., *et al.*: '*Design patterns: elements of reusable object-oriented software*' (Addison-Wesley, Boston, MA, USA, 1994)

[3] Schmidt, D., Stal, M., Rohnert, H., *et al.*: '*Pattern-oriented software architecture, volume 1: a system of patterns*' (John Wiley & Sons, Chichester, UK, 1996)

[4] Schmidt, D., Stal, M., Rohnert, H., *et al.*: '*Patterns for concurrent and networked objects, volume 2 of pattern-oriented software architecture*' (Wiley, Chichester, UK, 2000)

[5] Prechelt, L., Unger, B., Tichy, W.F., *et al.*: 'A controlled experiment in maintenance: comparing design patterns to simpler solutions', *IEEE Trans. Softw. Eng.*, 2001, **27**, (12), pp. 1134–1144

[6] Aversano, L., Cerulo, L., Di Penta, M.: 'Relating the evolution of design patterns and crosscutting concerns'. Seventh IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, 30 September–1 October 2007, pp. 180–192

[7] Gatrell, M., Counsell, S., Hall, T.: 'Design patterns and change proneness: a replication using proprietary C# software'. 16th Working Conf. on Reverse Engineering (WCRE'09), Lille, France, October 2009, pp. 160–164

[8] Vokáč, M., Tichy, W., Sjøberg, D.I., *et al.*: 'A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment', *Empir. Softw. Eng.*, 2004, **9**, (3), pp. 149–195

[9] Ampatzoglou, A., Frantzeskou, G., Stamelos, I.: 'A methodology to assess the impact of design patterns on software quality', *Inf. Softw. Technol.*, 2012, **54**, (4), pp. 331–346

[10] Nanthaamornphong, A., Carver, J.C.: 'Design patterns in software maintenance: an experiment replication at University of Alabama'. Second Int. Workshop on Replication in Empirical Software Engineering Research (RESER), Banff, AB, Canada, September 2011, pp. 15–24

[11] Ali, M., Elish, M.O.: 'A comparative literature survey of design patterns impact on software quality'. Int. Conf. on Information Science and Applications (ICISA), Suwon, South Korea, June 2013, pp. 1–7

[12] Ampatzoglou, A., Charalampidou, S., Stamelos, I.: 'Research state of the art on GoF design patterns: a mapping study', *J. Syst. Softw.*, 2013, **86**, (7), pp. 1945–1964

[13] Riaz, M., Breaux, T., Williams, L.: 'How have we evaluated software pattern application? a systematic mapping study of research design practices', *Inf. Softw. Technol.*, 2015, **65**, pp. 14–38

[14] Weiss, M.: 'Patterns and their impact on system concerns'. 13th Annual European Conf. on Pattern Languages of Programming (EuroPLoP), Irsee, Germany, July 2008, pp. S2-1–S2-10

[15] Zhang, C., Budgen, D.: 'What do we know about the effectiveness of software design patterns?', *IEEE Trans. Softw. Eng.*, 2012, **38**, (5), pp. 1213–1231

[16] Mayvan, B.B., Rasoolzadegan, A., Yazdi, Z.G.: 'The state of the art on design patterns: A systematic mapping of the literature', *J. Syst. Softw.*, 2017, **125**, pp. 93–118

[17] Kitchenham, B.: 'Procedures for performing systematic reviews'. TR/SE-0401, Keele University, Keele, UK, 2004

[18] Jabangwe, R., Börstler, J., Šmite, D., *et al.*: 'Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review', *Empir. Softw. Eng.*, 2015, **20**, (3), pp. 640–693

[19] Dyba, T., Dingsoyr, T., Hanssen, G.K.: 'Applying systematic reviews to diverse study types: an experience report'. First Int. Symp. on Empirical Software Engineering and Measurement (ESEM 2007), Madrid, Spain, September, 2007, pp. 225–234

[20] Scopus: abstract and citation database of peer-reviewed literature. Elsevier, 2019. Available at https://www.elsevier.com/solutions/scopus

[21] Aridor, Y., Lange, D.B.: 'Agent design patterns: elements of agent application design'. Proc. Second Int. Conf. on Autonomous Agents, Minneapolis, MN, USA, May 1998, pp. 108–115

[22] Izurieta, C., Bieman, J.M.: 'How software designs decay: a pilot study of pattern evolution'. First Int. Symp. on Empirical Software Engineering and Measurement (ESEM) 2007, Madrid, Spain, September 2007, pp. 449–451

[23] Izurieta, C., Bieman, J.M.: 'A multiple case study of design pattern decay, grime, and rot in evolving software systems', *Softw. Qual. J.*, 2013, **21**, (2), pp. 289–323

[24] Kothari, C.R.: '*Research methodology: methods and techniques*' (New Age International, New Delhi, India, 2004)

[25] Runeson, P., Höst, M.: 'Guidelines for conducting and reporting case study research in software engineering', *Empir. Softw. Eng.*, 2008, **14**, (2), pp. 131–164

[26] Guéhéneuc, Y.G.: 'P-MARt: pattern-like micro architecture repository'. Proc. 1st EuroPLoP Focus Group on Pattern Repositories, 2007, pp. 1–3

[27] Dong, J., Zhao, Y., Peng, T.: 'A review of design pattern mining techniques', *Int. J. Softw. Eng. Knowl. Eng.*, 2009, **19**, (6), pp. 823–855

[28] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., *et al.*: 'Design pattern detection using similarity scoring', *IEEE Trans. Softw. Eng.*, 2006, **32**, (11), pp. 896–909

[29] Guéhéneuc, Y.G., Antoniol, G.: 'DeMIMA: a multilayered approach for design pattern identification', *IEEE Trans. Softw. Eng.*, 2008, **34**, (5), pp. 667–684

[30] Guéhéneuc, Y.G.: 'Ptidej: a flexible reverse engineering tool suite'. IEEE Int. Conf. on Software Maintenance, 2007 (ICSM 2007), Paris, France, October 2007, pp. 529–530

[31] Shi, N., Olsson, R.A.: 'Reverse engineering of design patterns from Java source code'. 21st IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'06), Tokyo, Japan, September 2006, pp. 123–134

[32] Wohlin, C., Höst, M., Henningsson, K.: 'Empirical research methods in software engineering' in Conradi, R., Wang, A.I. (Eds.): '*Empirical methods and studies in software engineering*' (Springer, Verlag, Berlin & Heidelberg, 2003), pp. 7–23

[33] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., *et al.*: 'Preliminary guidelines for empirical research in software engineering', *IEEE Trans. Softw. Eng.*, 2002, **28**, (8), pp. 721–734

[34] Robson, C.: '*Real world research: a resource for social scientists and practitioners–researchers*' (John Wiley & Sons, Chichester, UK, 1993)

[35] Forward, A., Lethbridge, T.C.: 'The relevance of software documentation, tools and technologies: a survey'. Proc. 2002 ACM Symp. on Document Engineering, McLean, VA, USA, November 2002, pp. 26–33

[36] de Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: 'A study of the documentation essential to software maintenance'. Proc. 23rd Annual Int. Conf. on Design of Communication: Documenting & Designing for Pervasive Information, Coventry, UK, September, 2005, pp. 68–75

[37] El Emam, K., Benlarbi, S., Goel, N., *et al.*: 'The confounding effect of class size on the validity of object-oriented metrics', *IEEE Trans. Softw. Eng.*, 2001, **27**, (7), pp. 630–650

[38] Chidamber, S.R., Kemerer, C.F.: 'A metrics suite for object oriented design', *IEEE Trans. Softw. Eng.*, 1994, **20**, (6), pp. 476–493

[39] Pree, W.: 'Meta patterns – a means for capturing the essentials of reusable object-oriented design'. European Conf. on Object-Oriented Programming (ECOOP'94), Bologna, Italy, July, 1994, pp. 150–162

[40] Wedyan, F., Ghosh, S., Vijayasarathy, L.R.: 'An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs', *Inf. Softw. Technol.*, 2015, **59**, pp. 233–254

[41] Eaddy, M., Zimmermann, T., Sherwood, K.D., *et al.*: 'Do crosscutting concerns cause defects?', *IEEE Trans. Softw. Eng.*, 2008, **34**, (4), pp. 497–515

[42] Eaddy, M., Aho, A., Murphy, G.C.: 'Identifying, assigning, and quantifying crosscutting concerns'. Proc. First Int. Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), Minneapolis, MN, USA, May 2007, p. 2. Available at http://dx.doi.org/10.1109/ACOM.2007.4

[43] Wedyan, F., Ghosh, S.: 'A dataflow testing approach for aspect-oriented programs'. 2010 IEEE 12th Int. Symp. on High-Assurance Systems Engineering (HASE), San Jose, CA, USA, November 2010, pp. 64–73

[44] The AspectJ Team: 'Aspectj Compiler 1.8.10', 2017. Available at http://www.eclipse.org/aspectj/

[45] Roo, A., Hendriks, M., Havinga, W., *et al.*: 'Compose*: a language-and platform-independent aspect compiler for composition filters'. Int. Workshop on Advanced Software Development Tools and Techniques, Paphos, Cyprus, July 2008, pp. 1–14

[46] Tanter, É., Figueroa, I., Tabareau, N.: 'Execution levels for aspect-oriented programming: design, semantics, implementations and applications', *Sci. Comput. Program.*, 2014, **80**, pp. 311–342

[47] Leger, P., Fukuda, H.: 'Why do developers not take advantage of the progress in modularity?'. Proc. 8th Int. Conf. on Bioinspired Information and Communications Technologies, Boston, MA, USA, December 2014, pp. 388–389

[48] Bansiya, J., Davis, C.G.: 'A hierarchical model for object-oriented design quality assessment', *IEEE Trans. Softw. Eng.*, 2002, **28**, (1), pp. 4–17

[49] Bakota, T., Hegedűs, P., Körtvélyesi, P., *et al.*: 'A probabilistic software quality model'. 2011 27th IEEE Int. Conf. on Software Maintenance (ICSM), Williamsburg, VI, USA, September 2011, pp. 243–252

[50] Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., *et al.*: 'The effect of GoF design patterns on stability: a case study', *IEEE Trans. Softw. Eng.*, 2015, **41**, (8), pp. 781–802

[51] ISO 9126: '*Information technology: software product evaluation, quality characteristics and guidelines for their use*' (International Organization for Standardization, Geneva, Switzerland, 1992)

[52] ISO 25010: '*ISO/IEC 25010:2011: 2011 systems and software engineering–systems and software quality requirements and evaluation (SQuaRE)–system and software quality models*' (International Organization for Standardization, Geneva, Switzerland, 2011)), pp. 1–34

[53] Bieman, J.M., Jain, D., Yang, H.J.: 'OO design patterns, design structure, and program changes: an industrial case study'. Proc. IEEE Int. Conf. on Software Maintenance, Florence, Italy, November 2001, pp. 580–589

[54] Yin, R.K.: '*Case study research: design and methods*' (Sage Publications, Thousand Oaks, CA, USA, 2009, 4th edn.)

[55] Kerlinger, F.N., Lee, H.B.: '*Foundations of behavioral research*' (Wadsworth Publishing, Belmont, CA, USA, 1999)

[56] Fisher, R.J.: 'Social desirability bias and the validity of indirect questioning', *J. Consum. Res.*, 1993, **20**, (2), pp. 303–315

[57] Zhou, X., Jin, Y., Zhang, H., *et al.*: 'A map of threats to validity of systematic literature reviews in software engineering'. 2016 23rd Asia–Pacific Software Engineering Conf. (APSEC), Hamilton, New Zealand, December 2016, pp. 153–160

[58] Afacan, T.: 'State design pattern implementation of a DSP processor: a case study of TMS5416C'. 6th IEEE Int. Symp. on Industrial Embedded Systems (SIES), Vasteras, Sweden, June 2011, pp. 67–70

[59] Petersen, K., Vakkalanka, S., Kuzniarz, L.: 'Guidelines for conducting systematic mapping studies in software engineering: an update', *Inf. Softw. Technol.*, 2015, **64**, pp. 1–18

10 Appendix

10.1 List of primary studies

PS1 Unger, B., Tichy, W.F.: ‘Do design patterns improve communication? An experiment with pair design’. International Workshop Empirical Studies of Software Maintenance, Limerick, Ireland, 2000, pp. 1–5

PS2 Huston, B.: ‘The effects of design pattern application on metric scores’, *Journal of Systems and Software*, 2001, **58**, (3), pp. 261–269

PS3 Bieman, J.M., Jain, D., Yang, H.J.: ‘OO design patterns, design structure, and program changes: an industrial case study’. Proceedings of the IEEE International Conference on Software Maintenance, Florence, Italy, 2001, pp. 580–589

PS4 McNatt, W.B., Bieman, J.M.: ‘Coupling of design patterns: common practices and their benefits’. 25th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, IL, USA, 2001, pp. 574–579

PS5 Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G.: ‘A controlled experiment in maintenance: comparing design patterns to simpler solutions’, *IEEE Transactions on Software Engineering*, 2001, **27**, (12), pp. 1134–1144

PS6 Hannemann, J., Kiczales, G.: ‘Design pattern implementation in java and AspectJ’. Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA'02, Seattle, Washington, USA, 2002, pp. 161–173

PS7 Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.F.: ‘Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance’, *IEEE Transactions on Software Engineering*, 2002, **28**, (6), pp. 595–606

PS8 Baudry, B., Traon, Y., Sunyé, G., Jézéquel, J.M.: ‘Measuring and improving design patterns testability’. Proceedings Ninth International Software Metrics Symposium, Sydney, NSW, Australia, 2003, pp. 50–59

PS9 Bieman, J.M., Straw, G., Wang, H., Munger, P.W., Alexander, R.T.: ‘Design patterns and change proneness: an examination of five evolving systems’. Proceedings of the Ninth International Software Metrics Symposium, Sydney, NSW, Australia, 2003, pp. 40–49

PS10 Vokáč, M.: ‘Defect frequency and design patterns: an empirical study of industrial code’, *IEEE Transactions on Software Engineering*, 2004, **30**, (12), pp. 904–917

PS11 Vokáč, M., Tichy, W., Sjøberg, D.I., Arisholm, E., Aldrin, M.: ‘A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment’, *Empirical Software Engineering*, 2004, **9**, (3), pp. 149–195

PS12 Rudzki, J.: ‘How design patterns affect application performance–a case of a multi-tier J2EE application’. FIDJI'04 Proceedings of the 4th International Conference on Scientific Engineering of Distributed Java Applications, Luxembourg-Kirchberg, Luxembourg, 2005, pp. 12–23

PS13 Ng, T., Cheung, S., Chan, W., Yu, Y.: ‘Toward effective deployment of design patterns for software extension: a case study’. Proceedings of the 2006 International Workshop on Software Quality, Shanghai, China, 2006, pp. 51–56

PS14 Ng, T., Cheung, S., Chan, W.K., Yu, Y.T.: ‘Work experience versus refactoring to design patterns: a controlled experiment’. Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Portland, Oregon, USA, 2006, pp. 12–22

PS15 Ellis, B., Stylos, J., Myers, B.: ‘The factory pattern in API design: a usability evaluation’. Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, 2007, pp. 302–312

PS16 Aversano, L., Cerulo, L., Di Penta, M.: ‘Relating the evolution of design patterns and crosscutting concerns’. Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, 2007, pp. 180–192

PS17 Ng, T., Cheung, S., Chan, W., Yu, Y.T.: ‘Do maintainers utilize deployed design patterns effectively?’. Proceedings of the 29th international conference on Software Engineering, Minneapolis, MN, USA, 2007, pp. 168–177

PS18 Ampatzoglou, A., Chatzigeorgiou, A.: ‘Evaluation of object-oriented design patterns in game development’, *Information and Software Technology*, 2007, **49**, (5), pp. 445–454

PS19 Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., Di Penta, M.: ‘An empirical study on the evolution of design patterns’. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Dubrovnik, Croatia, 2007, pp. 385–394

PS20 Khomh, F., Guéhéneuc, Y.G.: ‘Do design patterns impact software quality positively?’. 12th European Conference on Software Maintenance and Reengineering (CSMR), Athens, Greece, 2008, pp. 274–278

PS21 Penta, M.D., Cerulo, L., Guéhéneuc, Y.G., Antoniol, G.: ‘An empirical study of the relationships between design pattern roles and class change proneness’. IEEE International Conference on Software Maintenance (ICSM), Beijing, China, 2008, pp. 217–226

PS22 Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: ‘Modularizing design patterns with aspects: a quantitative study’. Transactions on Aspect-Oriented Software Development I, 2006, pp. 36–74

PS23 Aversano, L., Cerulo, L., Di Penta, M.: ‘Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study’, *IET Software*, 2009, **3**, (5), pp. 395–409

PS24 Gatrell, M., Counsell, S., Hall, T.: ‘Design patterns and change proneness: a replication using proprietary C# software’. 16th Working Conference on Reverse Engineering (WCRE'09), Lille, France, 2009, pp. 160–164

PS25 Garzás, J., Garca, F., Piattini, M.: ‘Do rules and patterns affect design maintainability?’, *Journal of Computer Science and Technology*, 2009, **24**, (2), pp. 262–272

PS26 Jeanmart, S., Gueheneuc, Y.G., Sahraoui, H., Habra, N.: ‘Impact of the visitor pattern on program comprehension and maintenance’. Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, Florida USA, 2009, pp. 69–78

PS27 Scanniello, G., Gravino, C., Risi, M., Tortora, G.: ‘A controlled experiment for assessing the contribution of design pattern documentation on software maintenance’. Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Bolzano-Bozen, Italy, 2010, article 52

PS28 Gatrell, M., Counsell, S.: ‘Design patterns and fault-proneness a study of commercial C# software’. Fifth International Conference on Research Challenges in Information Science (RCIS), Gosier, France, 2011, pp. 1–8

PS29 Gravino, C., Risi, M., Scanniello, G., Tortora, G.: ‘Does the documentation of design pattern instances impact on source code comprehension? Results from two controlled experiments’. 18th Working Conference on Reverse Engineering (WCRE), Limerick, Ireland, 2011, pp. 67–76

PS30 Hsueh, N.L., Wen, L.C., Ting, D.H., Chu, W., Chang, C.H., Koong, C.S.: ‘An approach for evaluating the effectiveness of design patterns in software evolution’. IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW), Munich, Germany, 2011, pp. 315–320

PS31 Ampatzoglou, A., Kritikos, A., Arvanitou, E.M., Gortzis, A., Chatziasimidis, F., Stamelos, I.: 'An empirical investigation on the impact of design pattern application on computer game defects'. Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, Tampere, Finland, 2011, pp. 214–221

PS32 Posnett, D., Bird, C., Dévanbu, P.: 'An empirical study on the influence of pattern roles on change-proneness', *Empirical Software Engineering*, 2011, **16**, (3), pp. 396–423

PS33 Ampatzoglou, A., Kritikos, A., Kakarontzas, G., Stamelos, I.: 'An empirical investigation on the reusability of design patterns and software packages', *Journal of Systems and Software*, 2011, **84**, (12), pp. 2265–2283

PS34 Hegedüs, P., Bán, D., Ferenc, R., Gyimóthy, T.: 'Myth or reality? Analyzing the effect of design patterns on software maintainability'. Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity, 2012, pp. 138–145

PS35 Ampatzoglou, A., Frantzeskou, G., Stamelos, I.: 'A methodology to assess the impact of design patterns on software quality', *Information and Software Technology*, 2012, **54**, (4), pp. 331–346

PS36 Gatrell, M., Counsell, S.: 'Faults and their relationship to implemented patterns, coupling and cohesion in commercial C# software', *International Journal of Information System Modeling and Design (IJISMD)*, 2012, **3**, (2), pp. 69–88

PS37 Ng, T.H., Yu, Y.T., Cheung, S.C., Chan, W.K.: 'Human and program factors affecting the maintenance of programs with deployed design patterns', *Information and Software Technology*, 2012, **54**, (1), pp. 99–118

PS38 Izurieta, C., Bieman, J.M.: 'A multiple case study of design pattern decay, grime, and rot in evolving software systems', *Software Quality Journal*, 2013, **21**, (2), pp. 289–323

PS39 Sfetsos, P., Ampatzoglou, A., Chatzigeorgiou, A., Deligiannis, I., Stamelos, I.: 'A comparative study on the effectiveness of patterns in software libraries and standalone applications'. 9th International Conference on the Quality of Information and Communications Technology (QUATIC), Guimaraes, Portugal, 2014, pp. 145–150

PS40 Rossi, B., Russo, B.: 'Evolution of design patterns: a replication study'. Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Torino, Italy, 2014, article 38

PS41 Cacho, N., Sant'Anna, C., Figueiredo, E., Dantas, F., Garcia, A., Batista, T.: 'Blending design patterns with aspects: a quantitative study', *Journal of Systems and Software*, 2014, **98**, pp. 117–139

PS42 Jaafar, F., Guéhéneuc, Y.G., Hamel, S., Khomh, F., Zulkernine, M.: 'Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults', *Empirical Software Engineering*, 2015, **21**, (3), pp. 896–931

PS43 Scanniello, G., Gravino, C., Risi, M., Tortora, G., Dodero, G.: 'Documenting design-pattern instances: a family of experiments on source-code comprehensibility', *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2015, **24**, (3), p. 14

PS44 Elish, M.O., Mohammed, M.A.: 'Quantitative analysis of fault density in design patterns: an empirical study', *Information and Software Technology*, 2015, **66**, pp. 58–72

PS45 Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., Avgeriou, P.: 'The effect of GoF design patterns on stability: a case study', *IEEE Transactions on Software Engineering*, 2015, **41**, (8), pp. 781–802

PS46 Walter, B., Alkhaeir, T.: 'The relationship between design patterns and code smells: an exploratory study', *Information and Software Technology*, 2016, **74**, pp. 127–142

PS47 Feitosa, D., Alders, R., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y.: 'Investigating the effect of design patterns on energy consumption', *Journal of Software: Evolution and Process*, 2017, **29**, (2), p. e1851

PS48 Hussain, S., Keung, J., Khan, A.A.: 'The effect of gang-of-four design patterns usage on design quality attributes'. IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 2017, pp. 263–273

PS49 Charalampidou, S., Ampatzoglou, A., Avgeriou, P., Sencer, S., Arvanitou, E.M., Stamelos, I.: 'A theoretical model for capturing the impact of design patterns on quality: the decorator case study'. The 32nd ACM SIGAPP Symposium on Applied Computing, Marrakech, Morocco, 2017, pp. 1231–1238

PS50 Feitosa, D., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y.: 'Correlating pattern grime and quality attributes', *IEEE Access*, 2018, **6**, pp. 23065–23078