Information and Software Technology 54 (2012) 331–346

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

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

A methodology to assess the impact of design patterns on software quality

Apostolos Ampatzoglou [a,*], Georgia Frantzeskou [b], Ioannis Stamelos [a]

[a] Department of Informatics, Aristotle University, Thessaloniki, Greece
[b] Information and Communication Systems Engineering Department, University of the Aegean, Samos, Greece

ARTICLE INFO

Article history:
Received 8 June 2011
Received in revised form 24 October 2011
Accepted 29 October 2011
Available online 4 November 2011

Keywords:
Structural quality
Design patterns
Object-oriented metrics
Quality

ABSTRACT

Context: Software quality is considered to be one of the most important concerns of software production teams. Additionally, design patterns are documented solutions to common design problems that are expected to enhance software quality. Until now, the results on the effect of design patterns on software quality are controversial.
Aims: This study aims to propose a methodology for comparing design patterns to alternative designs with an analytical method. Additionally, the study illustrates the methodology by comparing three design patterns with two alternative solutions, with respect to several quality attributes.
Method: The paper introduces a theoretical/analytical methodology to compare sets of "canonical" solutions to design problems. The study is theoretical in the sense that the solutions are disconnected from real systems, even though they stem from concrete problems. The study is analytical in the sense that the solutions are compared based on their possible numbers of classes and on equations representing the values of the various structural quality attributes in function of these numbers of classes. The exploratory designs have been produced by studying the literature, by investigating open-source projects and by using design patterns. In addition to that, we have created a tool that helps practitioners in choosing the optimal design solution, according to their special needs.
Results: The results of our research suggest that the decision of applying a design pattern is usually a trade-off, because patterns are not universally good or bad. Patterns typically improve certain aspects of software quality, while they might weaken some other.
Conclusions: Concluding the proposed methodology is applicable for comparing patterns and alternative designs, and highlights existing threshold that when surpassed the design pattern is getting more or less beneficial than the alternative design. More specifically, the identification of such thresholds can become very useful for decision making during system design and refactoring.

1. Introduction

Object oriented design patterns have been introduced in mid 1990s as a catalog of common solutions to common design problems, and are considered as standard of "good" software designs [31]. The notion of patterns was firstly introduced by Alexander et al. [2] in the field of architecture. Later the notion of patterns has been transformed in order to fit software design by Gamma, Helm, Johnson and Vlissides (GoF) [31]. The authors cataloged 23 design patterns, classified according to two criteria. The first, i.e. purpose, represents the motivation of the pattern. Under this scope patterns are divided into creational, structural and behavioral patterns. The second criterion, i.e. scope, defines whether the pattern is applied on object or class level. In [31], the authors suggest that using specific software design solutions, i.e. design patterns, provide easier maintainability and reusability, more understandable implementation and more flexible design. At this point it is necessary to clarify GoF are not the first or the only design patterns in software literature. Some other well known patterns are architectural patterns, computational patterns, game design patterns, etc. In recent years, many researchers have attempted to evaluate the effect of GoF design patterns on software quality. Reviewing the literature on the effects of design pattern application on software quality provides controversial results. Until now, researchers attempted to investigate the outcome of design patterns with respect to software quality through empirical methods, i.e. case studies, surveys and experiments, but safe conclusions cannot be drawn since the results lead to different directions. As mentioned in [37,39,53,59,69], design patterns propose elaborate design solutions to common design problems that can be implemented with simpler solutions as well.

In this paper we propose a methodology for comparing pattern designs. The proposed method is analytical in the sense that

* Corresponding author.
E-mail addresses: apamp@csd.auth.gr (A. Ampatzoglou), gfran@aegean.gr (G. Frantzeskou), stamelos@csd.auth.gr (I. Stamelos).

0950-5849/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2011.10.006

system comparison is not performed on specific system instances, but on design motifs that can describe every possible instance of a system during its extension. The methodology is generic so that it can be applied for comparing design alternatives that describe equivalent functionality and have specified axes of change. The proposed method attempts to formulate several quality attributes as functions of functionality addition on multiple, equivalent solutions to a design problem. Then the functions are compared so as to identify cut-off points during system maintenance when one solution gets better or worse than the other. In this study we illustrate the applicability of the methodology by employing it for comparing GoF design patterns to equivalent adhoc solutions. In the next section, we present background information on design patterns. In Section 3, the methodology is presented and in Section 4, three exploratory cases are demonstrated. A discussion on the methodology is presented in Section 5. Conclusions, threats to validity and future work are presented by the end of the paper.

2. Related work

In this section of the paper, previous scientific research related to design patterns is presented. More specifically, the section is organized into paragraphs concerning indications on pattern identification according to metric scores, qualitative evaluation of pattern employment, quantitative evaluation of pattern employment, discussion on pattern application and class change proneness, results of controlled experiments on the comparison of design pattern application versus simpler solutions with respect to code maintainability, research on pattern formalization, systematic pattern selection and finally the use of metrics for measuring design structures.

2.1. Metrics as indicators of pattern existence

Firstly, in several papers [5,32,33,53] it is implied that object oriented software metrics can provide indications on the necessity or emergence of specific design patterns. In [5], the authors compute several metrics (number of public, private and protected attributes, number of public, private and protected operations, number of associations, aggregations and inheritance, total number of attributes, methods and relations) so as to get indications on the existence of five structural patterns (Adapter, Proxy, Composite, Bridge and Decorator). Empirical results on the methodology show that software metrics are essential in order to reduce the problem search space and therefore enhance the proposed design pattern detection algorithm. In [53], the authors attempt to introduce some metrics for conditional statements and inheritance trees so as to provide indications for the necessity of applying design patterns, in a low quality design, through refactoring. The proposed methodology, apart from identifying the need for a pattern in a specific set of classes, provides suggestions concerning the pattern that should be applied for solving the problem. Gueheneuc et al. [32], propose a methodology on identifying design motif roles through the use of object-oriented metrics. The suggested fingerprints are ranges of metric scores that imply the existence of design motif role. The authors have chosen to use size, cohesion and coupling metrics, while the patterns under consideration are Abstract Factory, Adapter, Builder, Command, Composite, Decorator, Factory Method, Iterator, Observer, Prototype, Singleton, State, Strategy, Template Method and Visitor. In [33] the authors improve their identification process by combining a structural and a numerical approach. This fact leads to the identification of both complete and incomplete pattern instances, and the reduction of false positive results.

2.2. Qualitative evaluation of design patterns

Additionally, several studies attempted to qualitatively evaluate the effects of object-oriented design patterns on software quality. According to McNatt and Bieman [55], the authors claim that patterns should be considered as structural units and therefore issues such as cohesion and coupling between the patterns should be examined. More specifically, the couplings between patterns are characterized as "loose" and as "tight" and their benefits and costs with respect to maintainability, factorability and reusability are being examined. Although the paper introduces several coupling types, namely intersection, composite and embedded, the way that they are demarcated is not clear. Specifically, there is a default coupling category, i.e. intersection, and any type of coupling that does not fit any other group is classified in the default category. In [75], the author presents an industrial case study, where inappropriate pattern application has led to severe maintainability problems. The reasons of inappropriately using design patterns have been classified into two categories, i.e. (1) software developers have not understood the rationale behind the patterns that they have employed and (2) the patterns that have been employed have not met project requirements. Additionally, the paper highlights the need for documenting pattern usage and the fact that pattern removal is extremely costly. In [42], the authors investigate the correlation among the quality of the class and whether the class play any roles in a design pattern instance. The results suggest that there are several differences in quality characteristics of classes that participate in patterns.

2.3. Quantitative evaluation of design patterns

Furthermore, regarding quantitative evaluation of design pattern application, in [39], the author attempts to demonstrate the effect of three design patterns (Mediator, Bridge and Visitor) on metric scores of three different categories (coupling, inheritance and size). According to the paper, there are several metric thresholds that, when surpassed, the pattern application is beneficial. The study's methodology is solid since it is based on pattern definitions and mathematical equations, but it deals with only one metric per pattern. Additionally, in [38], the authors have investigated the effect of the patterns on one quality attribute, i.e. the most obvious quality attribute that the pattern has effect on. The selection of the quality attribute has been based on the pattern's non functional requirements, whereas the selection of the metric has been based on [8]. The drawback of this research is that it does not take into account possible trade-offs that pattern usage induces. For example, when a pattern is employed, the coupling of the system may decrease, but as a side effect the size may increase. If a quality attribute is related to size and coupling, drawing a conclusion that this attribute is enhanced because of the decrease in coupling is not safe.

In [4], the authors attempt to investigate the effect of design pattern application in game development through a case study. The results of the case study are both qualitative and quantitative, but the domain of the research is limited to games and therefore results cannot be generalized. The patterns under study are Bridge and Strategy, whereas the considered metric categories are size, complexity, coupling and cohesion. The results of the case study suggest that pattern application enhance cohesion, coupling and complexity metrics, but as a side effect the size metrics increase. In [41], Khomh et al. performed a survey with professional software engineers. The results of the empirical study suggest that design patterns do not always impact software quality positively. More specifically, the negatively influenced quality issues are suggested to be simplicity, learnability and understandability. However, as it is referenced in the paper, marginal results (e.g.

understandability, reusability) should be reconsidered or cautiously adopted, because neutral opinions are considered as negative.

2.4. Design patterns and change proneness

Moreover, an interesting domain concerning design patterns and internal software quality is a possible correlation between pattern employment and class change proneness [11,12,23]. In [11], the authors conducted an industrial case study that aimed at investigating correlation among code changes, reusability, design patterns, and class size. The results of the study suggest that the number of changes is highly correlated to class size and that classes that play roles in design patterns or that are reused through inheritance are more change prone than others. This study is well-structured and validated, although the results refer only to a specific maintainability issue, namely change proneness. Other maintainability aspects such as change effort and design quality are not considered. In addition to that, in [12] the authors replicate the case study described above into three professional and two open-source projects. The results of the second study do not fully agree with those of the prior case study. The relationships between class size, design patterns participation and change proneness are still valid, but appear weaker. In [23] the authors have attempted to investigate correlations among class change proneness, the role that a class holds in a pattern and the kind of change that occurs. The design patterns under study are Abstract Factory, Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State/Strategy, and Template. The empirical study that has been employed was conducted in three open source projects. The results of the paper comply with common sense for the majority of design patterns. However, there are cases where the obtained results differ from the expected ones.

2.5. Controlled experiments on design patterns and maintainability

Additionally, in [59,69], controlled experiments concerning the maintainability of systems with and without design patterns, have been conducted. In [59] the patterns considered have been Abstract Factory, Observer, Decorator, Composite and Visitor, while the subjects of the experiment have been professional software engineers. The results of the experiment suggest that it is usually useful to apply a design pattern rather than the simpler solution. The dilemma of employing a design pattern or a simple solution is best answered by the software engineer's common sense. The experiment in [69] is a replication of the aforementioned experiment and therefore uses the same patterns and similar subject groups. In addition to the former experiment, the authors increased experimental realism in the sense that subjects have used a real programming environment instead of pen and paper. The results suggest that design patterns are not universally beneficial or harmful with respect to maintenance, since each design pattern has its own nature. In the discussion of the paper conclusions concerning each design pattern are solidly presented.

2.6. Design pattern formalization

The articles that deal with pattern formalizations can be further divided in smaller categories according to their research topic. However, they all share the common aim of investigating, identifying and specifying innovative approaches that deal with ways of modeling and formalizing design patterns.

In [25,30,43,76] the authors deal with the visualization of architectural and design patterns with UML artifacts. These studies aid in providing an easy way to use design patterns with several practical benefits concerning tools that help practitioners in applying design patterns. In [16] the authors present a repository including formal specifications of the problems that each pattern solves. Additionally, the study suggests that patterns should be described as reusable artifacts that include the problem to be solved and a set of transformations that guide the designer through the application of the proposed solution.

Furthermore, several papers deal with presenting innovative approaches of pattern representations and specifications. In [9,10,65] the authors use first order predicate logic in order to specify the behavioral and structural characteristics of design patterns. In [1] a component based specification of design patterns is suggested, guided by the architectural design artifacts that are involved in patterns. Moreover, in [24] the authors propose new symbols on class and collaboration diagrams that help in pattern representation. In [7,14] the authors provide tools that are based on constraints and logical graphs in order to formalize design patterns.

Additionally, [27,51,64] introduce meta-programming languages for describing the way a pattern is applied. In three articles [28,52,79] the authors deal with enhancing the descriptions of design patterns. More specifically, they propose transformations for pattern application and they provide documentation on pattern usage. In [49,58,73] constructional attributes of patterns, a comparison of pattern visualization methods and general information on pattern comprehension are presented. Finally, in [57] the authors demonstrate a language for formally describing the Visitor pattern in order to capture its essence in a reusable library.

2.7. Systematic design pattern selection

In [13] the author presents a literature review on pattern search and pattern selection strategies. The problem of searching patterns is defined as the effort of getting information about existing patterns; whereas pattern selection is described as the problem of deciding which pattern to choose among all available solutions. Additionally, it is suggested that there are five approaches for searching and selecting patterns, namely (a) pattern repositories and pattern catalogs, (b) recommendation systems, (c) formal languages, (d) search engines and (e) other approaches.

In [26,61,70,72] the authors create online pattern repositories in order to increase the availability of patterns. In such repositories patterns can be retrieved through searching criteria and by manual browsing among various patterns. Furthermore, in [34,35] several recommendation systems are suggested in order to suggest the appropriate pattern, according to the problem that the developer wants to solve. In addition, several papers describe approaches that use formal languages in order to represent design patterns and select patterns according to such a representation [3,71,77]. Selecting patterns through search engines corresponds to searches through keywords in engines that crawl and index pattern descriptions. Finally, there are several other approaches that cannot be classified in any of the above categories such as [45,80].

2.8. Metrics as measurements of design structures

In literature there are many studies that are using metrics in order to measure the quality of systems and data structures. Some of these studies evaluate complete systems [8,17,19,21,22,46,47,62] and others evaluate design structures other than design patterns [6,40,67]. More specifically, in [6], the authors identify four coding bad-smells,[1] apply an automated transformation to fix the problem and evaluate the applied refactoring. In [67], the authors investigate the possible gains of applying the "move method" refactoring and

[1] Code that is expected to have low quality levels.

evaluate the benefits by coupling and cohesion metrics. Similarly, in [40] the authors perform an experiment in order to evaluate several refactorings with coupling, cohesion, complexity and size metrics.

3. Methodology

This paper is inspired by the work of Hsueh et al. [38] and Huston [39]. Our methodology can be used to compare design patterns and alternative design solutions of equivalent functionality, during system extension. The method of our paper can be summarized in the following steps:

Prerequisite: Suppose a design problem that can be treated with a pattern.

1. Catalog alternative design solutions from literature, open-source solutions and personal experiences.
2. Identify the major axes of change for the system
3. Extend the design in all axes. For example if two axes of change are available, extend in the first axis by adding (n) functionalities, and the second axis by (m) additional functionalities.
4. Select a quality model that fit the designer's needs, or simply a set of metrics. Any development team is free to select whether they want to evaluate their solutions w.r.t an existing quality model, w.r.t. a customized model, or w.r.t. just a set of metrics that are not aggregated or composed.
5. Construct equations for quality model components or quality metrics that are functions of functionality extension based on the axes of change identified in step 3.
6. Compare the functions provided by step 5, in pairs and identify which alternative is preferable w.r.t. every quality attribute described in 4, and under which constraints.

The main points of contribution w.r.t. the research state of the art on pattern evaluation are:

1. An analytical method is applied rather than an empirical one. The main benefit is that we do not need multiple case studies on each pattern, because the quality of every pattern instance can be calculated from the proposed mathematical equations.
2. The proposed methodology is capable of assessing different structural quality aspects and handle possible trade-offs between quality attributes.
3. The effect of design pattern application in a wider variety of attributes is investigated instead of a single quality attribute.
4. Certain cut-off points are identified (when possible), that when surpassed, the pattern or the alternative design becomes more beneficial.
5. The results of (3) and (4), can be employed in terms of goal oriented software project management, i.e. the designer can choose between pattern and alternative solutions w.r.t. system requirements and according to designer's preference for specific quality attributes.
6. It compares pattern solutions to design solutions identified in open-source software and not only to designs that are derived from literature review.

The proposed methodology is based on a relatively complex mathematical background and therefore it might be difficult to be applied by the average designer. For this reason we created a tool that implements all necessary steps and simulates the methodology for every possible pattern instance. The tool is presented in Section 5.

An alternative suggestion would be to work on pattern instances rather than on model level. Such an approach is much easier for developers, but it has several drawbacks compared to an analytical methodology. For example, if someone works with pattern instances he will assess the quality of a specific instance and will make a decision on the design he/she will use. However, it is possible that new requirements will arrive and the system will be extended. The new requirements will demand a greater number of classes and a completely different set of metrics will be calculated. At that point another design alternative might produce better results, compared to the original design. In this case, the corresponding part of the system will have to be refactored to the optimum design solution. In the case of an analytical methodology, that works on the model level the designs under consideration can be compared for unlimited number of pattern instances and design decisions can be drawn by taking into account future adoptions at an early designing stage.

4. Methodology illustration

As illustrative examples of using the proposed method, we investigate six research questions, evaluating design patterns w.r.t. various quality aspects. The research questions are listed below:

[RQ_1]Which is the effect of employing an instance of the bridge pattern on the maintainability of a system?
[RQ_2] Which is the effect of employing an instance of the abstract factory pattern on the maintainability of a system?
[RQ_3] Which is the effect of employing an instance of the bridge pattern on the code quality of a system?
[RQ_4] Which is the effect of employing an instance of the visitor pattern on design quality of a system?
[RQ_5] Which is the effect of employing an instance of the bridge pattern on design quality of a system?
[RQ_6] Which is the effect of employing an instance of the abstract factory pattern on design quality of a system?

The patterns are going to be presented in their general form, i.e. roles instead of class names, in order to show that the method can be applied on any instance of the specified design pattern. Similarly, the classes and functions of the adhoc solutions will be named according to the corresponding methods and classes in the pattern solution. The fact that all three examples are related to GoF design patterns, should not be interpreted as loss of generality, i.e. the method is applicable on any other form of patterns, e.g. architectural patterns [29].

The patterns that are going to be investigated in this study have been selected in order to cover all pattern categories described in [31]. The patterns are presented in their standard form. In the literature, several pattern variations have been suggested. However, such designs have not been considered in our research, because the count of alternative solutions, which can implement the functionality of the pattern, is high and the investigation of every possible combination is impractical. Thus, we preferred to extract an equivalent design solution from an open-source project, rather than an additional literature solution, in order to increase the realism of our approach. At this point it is necessary to clarify that the provided analysis is based on the current alternatives designs and the results do not aim at comparing patterns, literature solutions and open-source solutions, but only at comparing the corresponding design.

Additionally, the extension of the systems has been made according to [56]. In [56], the authors suggest that design patterns can be maintained in three possible manners: (a) adding a class as

a concrete participant, (b) modifying the existing interface participants and (c) introducing a new client. In that study it is suggested that (a) is the most common type of maintenance. Thus, in our extension scenario we preferred to maintain our systems by adding concrete participant classes. Selecting the axes of change from the design pattern perspective provides a uniform axe of change selection strategy. If we had decided to select the axes of change from the open source solution or from the literature solution, it would not be possible to use rules for selecting the axes of change.

In applying our approach, in the case of existing system classes that are reused or included in the pattern instance, we assume that their effect on quality will be uniform across all design solutions. This assumption is considered a threat to validity and is discussed in Section 6. In addition to that, we have chosen to present separate examples for every pattern, and not a system that includes more patterns, because of the undefined effects of pattern coupling [55]. The method of comparison is the solution of inequalities that arise from subtracting the mathematical formula calculating one metric score for one solution, from the corresponding formula for another solution and compare it to zero. In order to solve the aforementioned inequalities, we have used Wolfram Mathematica© that uses the Reduce algorithm [74] to solve inequalities.

At this point it is necessary to clarify that the selected quality models or the selected set of metrics are indicative. Thus, a software team that applies the method can select any metric suite that they believe better fits their needs. The fact that in each research question a different set of metrics will be used is based on the fact there is no uniform set of metrics that deals with maintainability (RQ_1, RQ_2), structural quality (RQ_3) and design quality (RQ_4, RQ_5, RQ_6). In the next sections, we will thoroughly present our method for the first research question. The results of applying our method in order to explore the rest of research questions are presented in a corresponding technical report,[2] in order to improve the readability of the paper.

4.1. Bridge design pattern

In this section we investigate the maintainability of systems that use the Bridge design pattern. Bridge is a structural pattern that it is used "when the designer wants to decouple an abstraction from its implementation so that the two can vary independently" [31]. Section 4.2 presents the structure of any Bridge pattern instance and two alternative designs. Section 4.3 discusses the metric suite that is used for assessing the structural quality of each design and Section 4.4 presents the results of applying our method for comparing the maintainability of the bridge pattern to the maintainability of the adhoc designs.

4.2. Design solutions

The literature alternative to the bridge pattern is retrieved from [39,66], where the authors used it for similar reasons. The use of a deeper inheritance tree has been preferred so as to eliminate the "cascaded if" statements that appear in the former design (see Fig. 1). The open source solution design employs a cascaded if statement for representing the dynamic behavior of the system. This solution uses the minimum possible number of classes and is considered to be the simplest design that an analyst could use. The class diagram of the design is depicted in Fig. 2. The bridge design pattern, as described in [31], prefers object composition over class inheritance, so the depth of inheritance tree decreases and the design conforms to the open-closed design principle [50]. The class diagram for the bridge pattern solution is presented in Fig. 3.

According to [56] the major axes of change in the pattern solution are: adding refined abstractions, adding concrete implementers, adding clients and adding methods and attributes to any class of the pattern. Since the most frequent change in a design pattern is the addition of subclasses [56], we have chosen to extend the system in the first two axes, i.e. add new *refined abstractions* and add new *concrete implementers*. More specifically, we suppose that the system under consideration has (n) *refined abstractions* and (m) *concrete implementers*. In the open source solution, the system has (n) *refined abstractions* and each *doOperation* method has (m) cascaded if statements. In the literature solution, similarly to the pattern solution the system has (n) *refined abstractions* and (m) *concrete implementers*. The third level of the hierarchy has ($n * m$) classes that represent every possible combination of abstractions and implementers. The source code of all solutions can be found in the web (http://sweng.csd.auth.gr/~apamp/material/patterns_code.rar).

4.3. Metric suite for assessing maintainability

For the purpose of our study we need to predict maintainability from code and design measurements. Therefore, we need to choose code/design metrics and a model to calculate maintainability from them. In [60] the authors present the results of a systematic review that deals with the metrics that can be used as predictors for software maintainability. In addition to that, the authors have ranked the studies that they have identified and suggested that the work of vanKoten and Gray [68] and Zhou and Leung [78] were the most solid ones [60].

In [68,78], the authors suggest that there are ten object-oriented metrics (see Table 1) that can be used as maintainability predictors. These metrics have been validated with several datasets and techniques such as Bayesian networks, regression trees and regression models, but each dataset indicated different metrics as the more influential with respect to maintainability. Thus, it was not possible for us to safely employ a regression model that would estimate the maintainability of a system. Consequently, we have investigated the effect of design patterns on the ten maintainability predictors and draw several conclusions assuming that all predictors are equally weighted.

According to [36,48] all metrics defined in Table 1 are inversely proportional to maintainability. Thus, the solution with the higher count of lower metric values is considered more maintainable.

4.4. Results

By taking into account the two major axes of change described in Section 4.2 and the metric definitions described in Section 4.3 we produced equations of metric scores as functions of functionality addition.

4.4.1. Literature solution

Concerning depth of inheritance, the Client and Abstraction class DIT = 0. For the (n) RefinedAbstractions DIT = 1. For the (nm) RefinedAbstractionNConcreteEmployeeM, DIT = 2.

$$DIT_{bridge(lss)} = \frac{n + 2nm}{n + m + nm + 2}$$

For the Abstraction class NOC = n. For each of the (n) classes that RefinedAbstractions, NOC = m. For all the other classes, NOC = 0.

$$NOC_{bridge(lss)} = \frac{n + nm}{n + m + nm + 2}$$

[2] AUTH/SWENG/TR-2011-10-01, available at http://sweng.csd.auth.gr/~apamp/material/TR-2011-10-01.pdf.

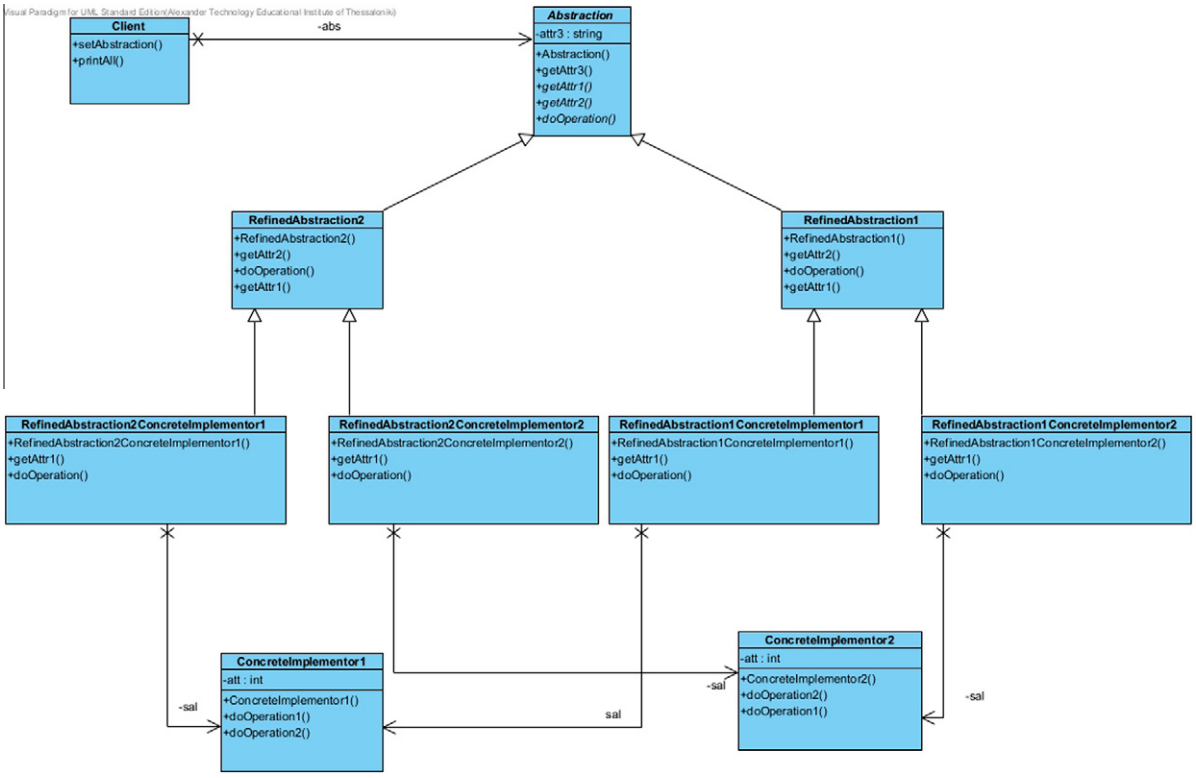


Fig. 1. Class diagram of literature solution.

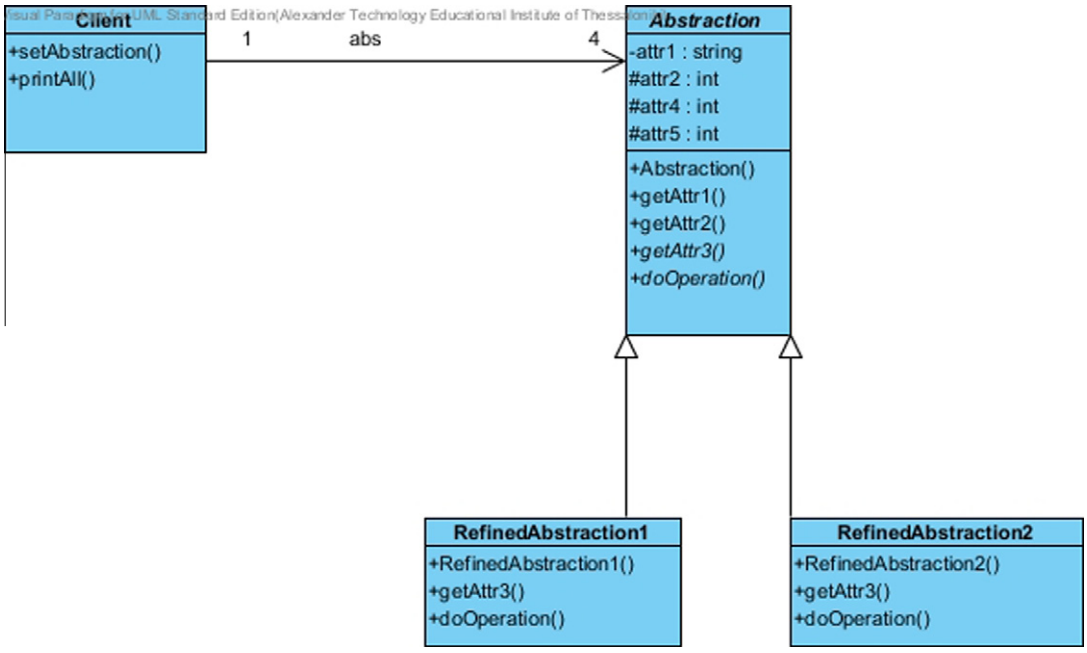


Fig. 2. Class diagram of open source solution.

The Client class sends four messages to the Abstraction class (MPC = 4). The (n) RefinedAbstraction classes send one message (MPC = 1). The (nm) RefinedAbstractionNConcreteEmployeeM send two messages to a ConcreteImplementor class and one to the Abstraction class (MPC = 3). For all the other classes MPC = 0.

$$\mathrm{MPC}_{\mathrm{bridge(lss)}} = \frac{4 + 3nm + n}{n + m + nm + 2}$$

The Client class invokes two local methods and four remote methods (RFC = 6). The Abstraction class has five local methods (RFC = 5). The (n) classes that represent Refined Abstractions have four local methods and call one remote method (RFC = 5). The (nm) RefinedAbstractionNConcreteEmployeeM classes have three local methods and invoke three remote methods (RFC = 6). The (m) classes that represent ConcreteImplementors have ($n + 1$) local methods (RFC = $n + 1$).

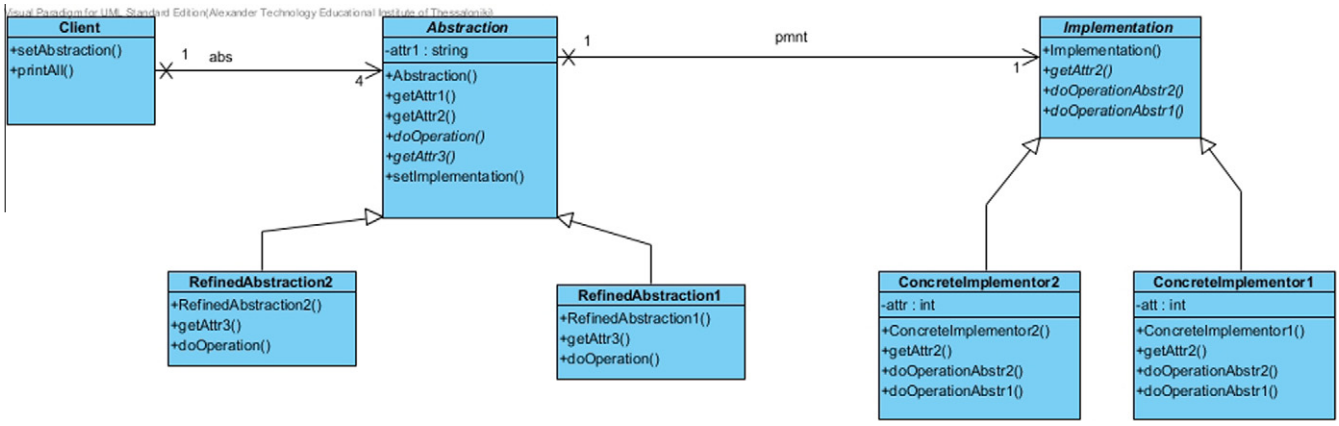


Fig. 3. Class diagram of bridge design pattern.

Table 1
Maintainability predictors.

Metric	Description
DIT	Depth of the inheritence tree (=inheritence level number of the class, 0 for the root class). Range of value [0,+∞)
NOC	Number of children (=number of direct sub-classes that the class has). Range of value [0,+∞)
MPC	Message-passing couple (=number of send statements defined in the class). Range of value [0,+∞)
RFC	Response for a class (=total number of local methods and the number of methods called by local methods in the class). Range of value [0,+∞)
LCOM	Lack of cohesion of methods (=number of disjoint sets of local methods, i.e. number of sets of local methods that do not interact with each other, in the class). Range of value [0,+∞)
DAC	Data abstraction coupling (=number of abstract data types defined in the class). Range of value [0,+∞)
WMPC	Weighted method per class (=sum of McCabe's cyclomatic complexity [54] of all local methods in the class). Range of value [0,+∞)
NOM	Number of methods (=number of local methods in the class). Range of value [0,+∞)
SIZE1	Lines of code (=number of semicolons in the class). Range of value [0,+∞)
SIZE2	Number of properties (=total number of attributes and the number of local methods in the class). Range of value [0,+∞)

$$\mathrm{RFC}_{\mathrm{bridge(lss)}} = \frac{11 + 5n + 7nm + m}{n + m + nm + 2}$$

For the Client class, LCOM = 0 − 1 = 0. For the Abstraction class, LCOM = 9 − 1 = 8. For the (n) RefinedAbstraction classes LCOM is not defined. For the (nm) classes that represent the combinations between types of RefinedAbstractions and types ConcreteImplementers, LCOM = 2 − 1 = 1. For the (m) classes that represent ConcreteImplementors, LCOM = 0.

$$\mathrm{LCOM}_{\mathrm{bridge(lss)}} = \frac{nm + 8}{m + nm + 2}$$

For the Client class (DAC = 1). For the (nm) classes that represent the combinations between types of RefinedAbstractions and types of ConcreteImplementors (DAC = 1).

$$\mathrm{DAC}_{\mathrm{bridge(lss)}} = \frac{1 + nm}{n + m + nm + 2}$$

For any method that does not involve any for, if, case, while, etc., statements CCmethod = 1. For methods that employ only one for statement CCmethod = 2. Any method that employs one cascaded if statement with m alternatives CCmethod = m. For the Client class, CC = 3. For the Abstraction class, CC = 5. For the (n) classes that represent RefinedAbstractions, CC = 4. For the (nm) classes that represent the combinations between types of RefinedAbstractions and types of ConcreteImplementors, CC = 3. For the (m) classes that represent ConcreteImplementors, CC = n + 1.

$$\mathrm{WMPC}_{\mathrm{bridge(lss)}} = \frac{8 + 4n + 4nm + m}{n + m + nm + 2}$$

The Client class holds two public methods; Abstraction holds 5, the (n) classes that represent the types of RefinedAbstraction hold 4, the (nm) classes which represent all the combinations between types of RefinedAbstractions and types of ConcreteImplementors, hold three and finally the (m) classes that represent the types of ConcreteImplementors hold (n + 1).

$$\mathrm{NOM}_{\mathrm{bridge(lss)}} = \frac{7 + 4n + 4nm + m}{n + m + nm + 2}$$

For the Client class, SIZE1 = 10. For the Abstraction class, SIZE1 = 7. For the (n) classes that represent RefinedAbstraction, SIZE1 = 4. For the (nm) classes that represent the combinations between types of RefinedAbstractions and types of ConcreteImplementors, SIZE1 = 5. For the (m) classes that represent ConcreteImplementors, SIZE1 = n + 3.

$$\mathrm{SIZE1}_{\mathrm{bridge(lss)}} = \frac{17 + 4n + 6nm + 3m}{n + m + nm + 2}$$

For the Client class, SIZE2 = 3. For the Abstraction class, SIZE2 = 6. For the (n) classes that represent RefinedAbstractions, SIZE2 = 4. For the (nm) classes that represent the combinations between types of RefinedAbstractions and types of ConcreteImplementors, SIZE2 = 4. For the (m) classes that represent ConcreteImplenters, SIZE2 = n + 2.

$$\mathrm{SIZE2}_{\mathrm{bridge(lss)}} = \frac{9 + 4n + 5nm + 2m}{n + m + nm + 2}$$

4.4.2. Open source solution

For classes Client, Abstraction DIT = 0. For the n classes that represent RefinedAbstractions DIT = 1.

$$\mathrm{DIT}_{\mathrm{bridge(OSS)}} = \frac{n}{n + 2}$$

For the Abstraction class NOC = n. For all the other classes, NOC = 0.

$$NOC_{bridge(OSS)} = \frac{n}{n+2}$$

The Client class sends four messages to the Abstraction class (MPC = 4). For the Abstraction class MPC = 0. The (n) RefinedAbstractions send one message to the Abstraction class (MPC = 1).

$$MPC_{bridge(OSS)} = \frac{n+4}{n+2}$$

The Client class invokes two local methods and four remote methods (RFC = 6). The Abstraction class has five local methods (RFC = 5). The (n) RefinedAbstractions have three local methods and call one remote method (RFC = 4).

$$RFC_{bridge(OSS)} = \frac{11+4n}{n+2}$$

For the Client class, LCOM = 0 − 1 = 0. For the Abstraction class, LCOM = 8 − 2 = 6. For the (n) RefinedAbstractions LCOM is not defined.

$$LCOM_{bridge(OSS)} = 3$$

For the Client class, DAC = 1. For all the other classes, DAC = 0.

$$DAC_{bridge(OSS)} = \frac{1}{n+2}$$

For any method that does not involve any for, if, case, while, etc., statements CCmethod = 1. For methods that employ only one for statement CCmethod = 2. For the Client class, CCClient = 3. For the Abstraction class, CCAbstraction = m + 4. For the n RefinedAbstractions, CCRefinedAbstraction = m + 2.

$$WMPC_{bridge(OSS)} = \frac{2n+nm+m+7}{n+2}$$

The Client class holds two public methods; Administrator holds five and the (n) RefinedAbstractions hold 3.

$$NOM_{bridge(OSS)} = \frac{7+3n}{n+2}$$

For the Client class, SIZE1 = 10. For the Abstraction class, SIZE1 = 14 + m. For the (n) RefinedAbstractions, SIZE1 = 4 + m.

$$SIZE1_{bridge(OSS)} = \frac{24+m+4n+nm}{n+2}$$

For the Client class, SIZE2 = 3. For the Abstraction class, SIZE2 = 9. For the (n) RefinedAbstractions, SIZE2 = 3.

$$SIZE2_{bridge(OSS)} = \frac{12+3n}{n+2}$$

4.4.3. Design pattern solution

For classes Client, Abstraction and Implementor DIT = 0. For the n RefinedAbstractions and for the m ConcreteImplementors DIT = 1.

$$DIT_{bridge(dps)} = \frac{n+m}{n+m+3}$$

For the Abstraction class NOC = n. For the Implementor, NOC = m.

$$NOC_{bridge(dps)} = \frac{n+m}{n+m+3}$$

The Client class sends four messages to the Abstraction class (MPC = 4). The Abstraction class sends one message to the Implementor class (MPC = 1). For the Implementor class, MPC = 0. The (n) RefinedAbstractions send one message to the Abstraction class and one to the Implementor class (MPC = 2). The (m) ConcreteImplementors send one message to the Implementor class (MPC = 1).

$$MPC_{bridge(dps)} = \frac{5+2n+m}{n+m+3}$$

The Client class invokes two local methods and four remote methods (RFC = 6). The Abstraction class has six local methods and one remote method (RFC = 7). The Implementor class has five local methods (RFC = 5). The (n) RefinedAbstractions have three local methods and call two remote methods (RFC = 5). The (m) ConcreteImplementors have five local methods and call one remote method (RFC = 6).

$$RFC_{bridge(dps)} = \frac{18+5n+6m}{n+m+3}$$

For the Client class, LCOM = 0 − 1 = 0. For the Abstraction class, LCOM = 13 − 2 = 11. For the (n) RefinedAbstraction and the Implementor class, LCOM is not defined. For the (m) ConcreteImplementors LCOM = 0.

$$LCOM_{bridge(dps)} = \frac{11}{m+2}$$

For the Client class, DAC = 1. For the Abstraction class, DAC = 1. For all the other classes, DAC = 0.

$$DAC_{bridge(dps)} = \frac{2}{n+m+3}$$

For any method that does not involve any for, if, case, while, etc., statements CCmethod = 1. For methods that employ only one for statement CCmethod = 2. For the Client class, CCClient = 3. For the Abstraction class, CCAbstraction = 6. For the Implementor class, CCImplementer = m + 2. For the n RefinedAbstractions, CCRefinedAbstraction = 3. For the m ConcreteImplementors, CCConcreteImplementer = m + 2.

$$WMPC_{bridge(dps)} = \frac{11+3m+3n+m^2}{n+m+3}$$

The Client class holds two public methods; Abstraction holds 6, Implementor holds (n + 2), the (n) RefinedAbstractions hold three and the (m) ConcreteImplementors hold (n + 2).

$$NOM_{bridge(dps)} = \frac{4n+10+nm+2m}{n+m+3}$$

For the Client class, SIZE1 = 10. For the Abstraction class, SIZE1 = 9. For the Implementor class, SIZE1 = m + 2. For the (n) RefinedAbstractions, SIZE1 = 3. For the (m) ConcreteImplementors, SIZE1 = m + 4.

$$SIZE1_{bridge(dps)} = \frac{21+m+3n+m(m+4)}{n+m+3}$$

For the Client class, SIZE2 = 3. For the Abstraction class, SIZE2 = 8. For the Implementor class, SIZE2 = m + 2. For the (n) RefinedAbstractions, SIZE2 = 3. For the (m) ConcreteImplementors, SIZE2 = m + 3.

$$SIZE2_{bridge(dps)} = \frac{13+m+3n+m(m+3)}{n+m+3}$$

In Table 2, we summarize the metrics scores for each solution, whereas in Table 3, we summarize the identified cut-off points where a design solution is getting better than another. The cut-off points are calculated by subtracting the function of solution S2 from the function of solution S1. The extracted formula is compared to zero, in (n) and (m) ranges that the value of the function is less than zero S2 has a higher metric score than S1.

5. Discussion

This section discusses our analytical methodology and the findings of our work concerning the research questions stated in

Table 2
Maintainability predictor scores for bridge design pattern and design alternatives.

Metric	LSS	OSS	DPS
DIT	$\frac{n+2nm}{n+m+nm+2}$	$\frac{n}{n+2}$	$\frac{n+m}{n+m+3}$
NOC	$\frac{n+nm}{n+m+nm+2}$	$\frac{n}{n+2}$	$\frac{n+m}{n+m+3}$
MPC	$\frac{4+3nm+n}{n+m+nm+2}$	$\frac{n+4}{n+2}$	$\frac{5+2n+m}{n+m+3}$
RFC	$\frac{11+5n+7nm+m}{n+m+nm+2}$	$\frac{11+4n}{n+2}$	$\frac{18+5n+6m}{n+m+3}$
LCOM	$\frac{nm+8}{m+nm+2}$	3	$\frac{11}{m+2}$
DAC	$\frac{1+nm}{n+m+nm+2}$	$\frac{1}{n+2}$	$\frac{2}{n+m+3}$
WMPC	$\frac{8+4n+4nm+m}{n+m+nm+2}$	$\frac{2n+nm+m+7}{n+2}$	$\frac{11+3m+3n+m^2}{n+m+3}$
NOM	$\frac{7+4n+4nm+m}{n+m+nm+2}$	$\frac{7+3n}{n+2}$	$\frac{4n+10+nm+2m}{n+m+3}$
SIZE1	$\frac{17+4n+6nm+3m}{n+m+nm+2}$	$\frac{24+m+4n+nm}{n+2}$	$\frac{21+m+3n+m(m+4)}{n+m+3}$
SIZE2	$\frac{9+4n+5nm+2m}{n+m+nm+2}$	$\frac{12+3n}{n+2}$	$\frac{13+m+3n+m(m+3)}{n+m+3}$

Table 3
Maintainability cut-off points.

$DIT_{DPS} < DIT_{OSS} < DIT_{LSS}$	$\forall n \geqslant 3, \forall m < \frac{n}{2}$
$DIT_{OSS} < DIT_{DPS} < DIT_{LSS}$	$(n = 2, \forall m \geqslant 2) \mid\mid (\forall n \geqslant 3, \forall m < \frac{n}{2})$
$NOC_{OSS} < NOC_{LSS} < NOC_{DPS}$	$\forall n = 2, \forall m > m_1$
$NOC_{OSS} < NOC_{DPS} < NOC_{LSS}$	$\forall n = 2, \forall m \leqslant m_1$
$NOC_{DPS} < NOC_{OSS} < NOC_{LSS}$	$\forall n \geqslant 3, \forall m < \frac{n}{2}$
$NOC_{OSS} < NOC_{DPS} < NOC_{LSS}$	$\forall n \geqslant 3, \forall m < \frac{n}{2}$ && $m \leqslant m_1$
$NOC_{OSS} < NOC_{LSS} < NOC_{DPS}$	$\forall n \geqslant 3, \forall n > m_1$
	$m_1 = \sqrt{n^2 - n + 1} + n - 1$
$MPC_{DPS} < MPC_{OSS} < MPC_{LSS}$	$\forall n \geqslant 2, \forall m > \frac{1}{2}(n^2 + 2n - 2)$
$MPC_{OSS} < MPC_{DPS} < MPC_{LSS}$	$\forall n \geqslant 2, \forall m < \frac{1}{2}(n^2 + 2n - 2)$
$RFC_{OSS} < RFC_{DPS} < RFC_{LSS}$	$(n = 4, \forall m \leqslant 6) \mid\mid (\forall n \geqslant 5, \forall m \geqslant 2)$
$RFC_{OSS} < RFC_{LSS} < RFC_{DPS}$	$(\forall n \geqslant 3, \forall m \geqslant 2) \mid\mid (n = 4, \forall m \geqslant 7)$
$LCOM_{LSS} < LCOM_{DPS} < LCOM_{OSS}$	$\forall n \geqslant 2, \forall m > m_1$
$LCOM_{DPS} < LCOM_{LSS} < LCOM_{OSS}$	$\forall n \geqslant 2, \forall m > m_1$
	$m_1 = \frac{3(1+3n)}{2n} + \frac{\sqrt{3}}{2}\sqrt{\frac{27n^2+26n+3}{n^2}}$
$DAC_{OSS} < DAC_{DPS} < DAC_{LSS}$	$\forall n \geqslant 2, \forall m > n + 1$
$DAC_{DPS} < DAC_{OSS} < DAC_{LSS}$	$\forall n \geqslant 2, \forall m > n + 1$
$WMPC_{LSS} < WMPC_{OSS}$	$\forall n, m \geqslant 2$
$WMPC_{OSS} > WMPC_{DPS}$	$(n = 2, 2 \leqslant m \leqslant 12) \mid\mid (\forall n \geqslant 3, m_1 < m < m_2)$
$WMPC_{OSS} < WMPC_{DPS}$	$(n = 2, \forall m \geqslant 14) \mid\mid (\forall n \geqslant 3,$ $m < m_1) \mid\mid (\forall n \geqslant 3, m > m_1)$
$WMPC_{LSS} < WMPC_{DPS}$	12 cut-off points
	$m_1 = \frac{n^2+3n+4}{2} - \frac{\sqrt{n^4+6n^3+13n^2+8n+12}}{2}$
	$m_2 = \frac{n^2+3n+4}{2} + \frac{\sqrt{n^4+6n^3+13n^2+8n+12}}{2}$
$NOM_{OSS} < NOM_{LSS} < NOM_{DPS}$	$(\forall n \geqslant 3, \forall m \geqslant 2) \mid\mid (n = 2, m = 2)$
$NOM_{LSS} < NOM_{OSS} < NOM_{DPS}$	$n = 2, \forall m \geqslant 3$
$SIZE1_{LSS} < SIZE1_{OSS}$	$\forall n, m \geqslant 2$
$SIZE1_{OSS} < SIZE1_{DPS}$	$\forall n \geqslant 2, m > m_1$
$SIZE1_{OSS} < SIZE1_{DPS}$	$\forall n \geqslant 2, m \leqslant m_1$
$SIZE1_{LSS} < SIZE1_{DPS}$	4 cut-off points
	$m_1 = \frac{n^2+3n+17}{2} + \frac{\sqrt{n^4+6n^3+47n^2+138n+409}}{2}$
$SIZE2_{LSS} < SIZE2_{OSS}$	$n = 2, \forall m \geqslant 2$
$SIZE2_{LSS} > SIZE2_{OSS}$	$\forall n \geqslant 3, \forall m \geqslant 2$
$SIZE2_{OSS} < SIZE2_{DPS}$	$\forall n \geqslant 2, \forall m \geqslant m_1$
$SIZE2_{OSS} > SIZE2_{DPS}$	$\forall n \geqslant 2, \forall m \geqslant m_1$
$SIZE2_{LSS} > SIZE2_{DPS}$	3 cut-off points
	$m_1 = \frac{4-n}{2(2+n)} + \frac{\sqrt{3}\sqrt{3n^2+16n+32}}{(2+n)}$

Section 4. The results that are presented in this paper can be used in various ways. First of all, the mathematical formulas that have been extracted can be used for goal driven decision making. Our study suggests that if a designer is about to apply a pattern, he/she should firstly estimate the number of classes that his system will probably use, then should select the quality attributes that he is most interested in and then select the most fitting design according to the thresholds described in the paper, i.e. he can opt for the design pattern solution or a personalized solution.

In order for the results of the research to have more practical effects and help practitioners that struggle with close deadlines, we have created a decision support tool, called DesignPAD (DESIGN Pattern ADvisor) that aids the developer to choose between the design pattern and the personalized design solution. At this point, the tool provides assistance on decisions concerning the employment of three patterns (Bridge, Factory and Visitor) and several quality attributes. The input of the tool is the pattern under consideration, the estimated system size (either a single number or a range of values) and the goals of the design team with respect to quality attributes. The system based on the results of the study, suggests the most fitting design according to the development team needs. Thus, the tool simulates all the steps of the proposed methodology, for the three patterns and the six alternative designs. Consequently, the developer does not have to deal with complicated mathematical formulas or inequality solving. The tool is available on the web[3]. Screenshots of the tool can be found in Figs. 4–6. In Fig. 4, we see the main screen of the tool where the user can select the pattern he/she wants to investigate. In Fig. 5, the form from where the user selects the metrics he/she is interested in and defines the (n) and (m) for the system is presented. Finally, Fig. 6 depicts the way the results are presented. The first table displays the average metric score for each alternative in the given range of (n) and (m). The second table counts how many times each design alternative produces "best", "medium" and "worse" results.

The results of applying the proposed methodology on three GoF design patterns, are in accordance to the results of the controlled experiments in [59,69], where the authors suggested that patterns are not universally good or bad. Furthermore, an additional factor, i.e. pattern size, other than developers' prior experience with patterns has been highlighted. The results suggested that certain pattern instances (w.r.t. pattern size) cannot produce the best solutions concerning several quality attributes independently of the prior knowledge and expertise of the developer. Thus, even if a pattern fits a certain design problem, the size of the pattern should be taken into account, before applying the pattern.

Concerning the applicability of the methodology to the rest of the design patterns, we believe that the proposed method can be applied to the majority of the rest design patterns. One limitation of the method is the existence of alternative design solutions to design patterns. If the design pattern under consideration has not been linked to any design alternative, the method cannot be applied because there is no set of designs to compare. Another limitation is the existence of axes of change other than "add methods and attributes to any class that participates in the pattern". For this reason the method cannot be applied to patterns that do not involve class hierarchies and client classes, e.g. Singleton. Furthermore, our method might not produce distinguishing results for patterns that can be extended only by adding pattern clients, e.g. Adapter and to patterns that do not seem to have significant effects on software structural quality, e.g. Façade.

Additionally, the method can be applied to any set of metrics or qualitative quality models. Similarly, the method can be applied to any kind of pattern or microstructure, other than the GoF, under the constraint that the pattern produces some kind of code that can be quantitively evaluated. For example, the application of the method is feasible on architectural patterns. In [44], the authors evaluate the use of the *Registry* pattern, i.e. an architectural pattern, by comparing it to a simple OO implementation and with an AOP implementation. Since all design alternatives share some common metric scores, our method can be used in order to formulate the metric scores during all systems' extension.

5.1. Design patterns and maintainability

From the results of Table 2 and AUTH/SWENG/TR-2011-10-01, it is clear that there is no universal answer to the question "Does

[3] http://sweng.csd.auth.gr/~apamp/material/DesignPAD_v2.0.rar.

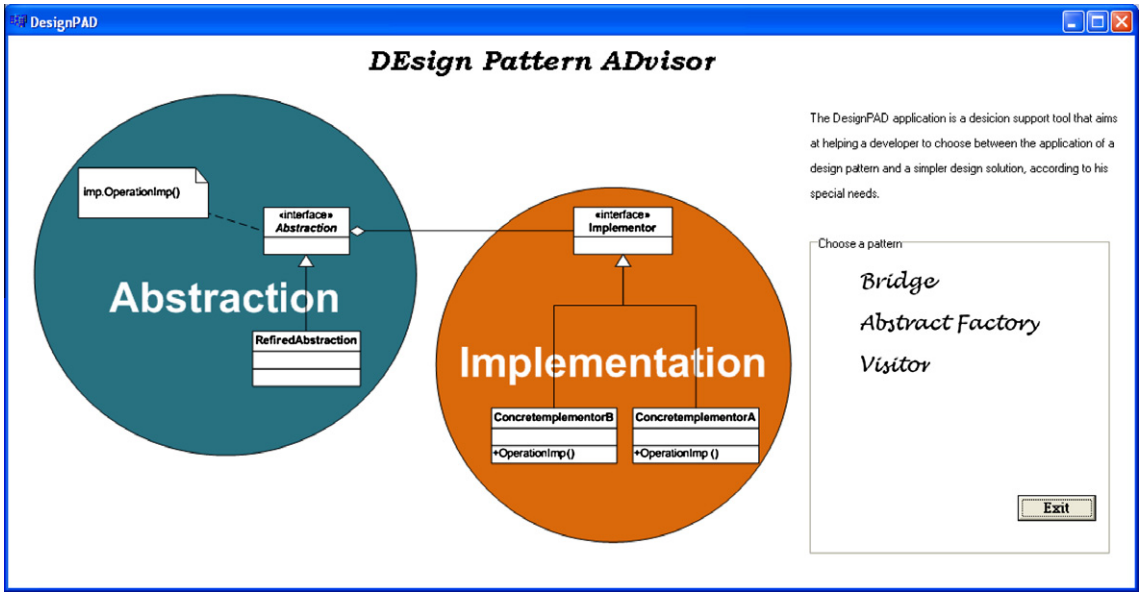


Fig. 4. DePAD – main screen.

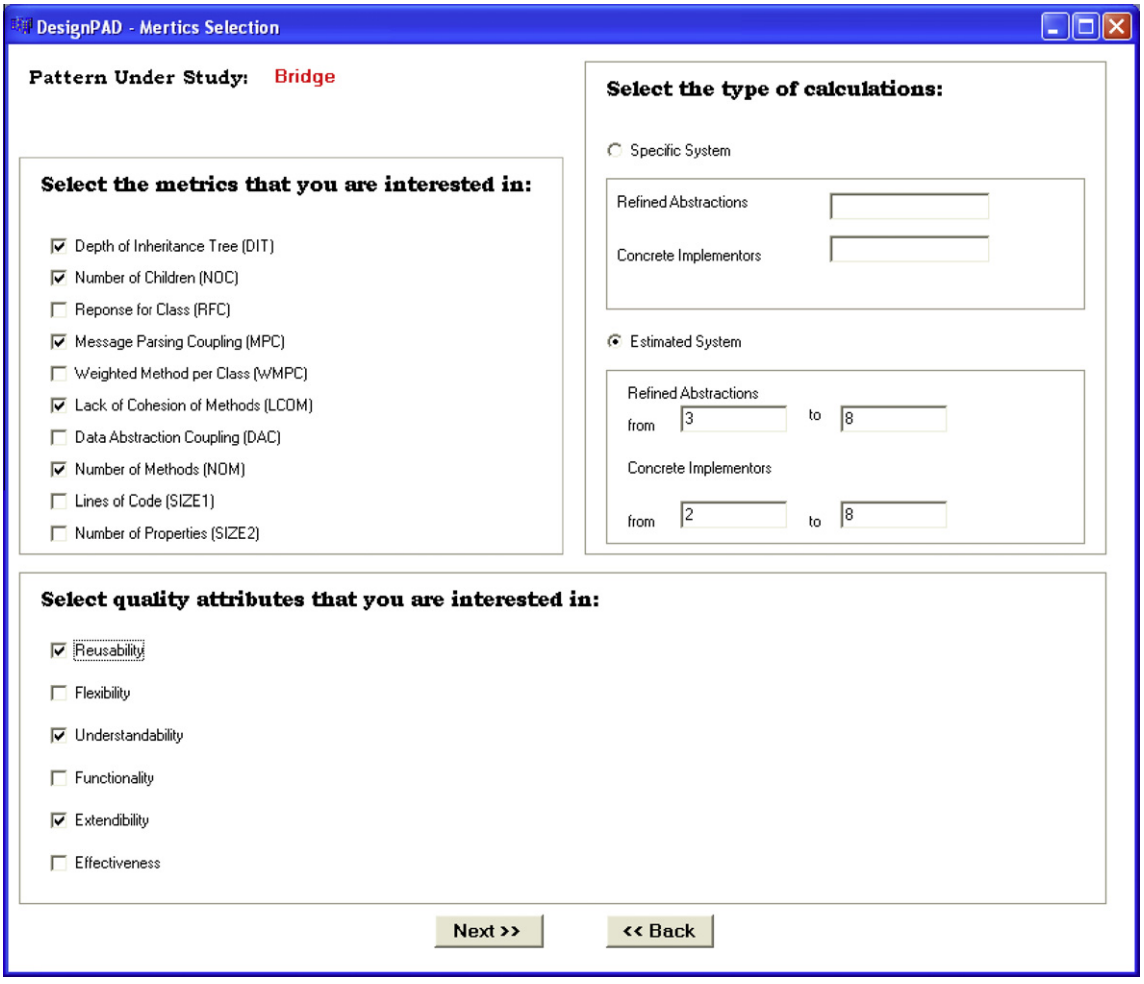
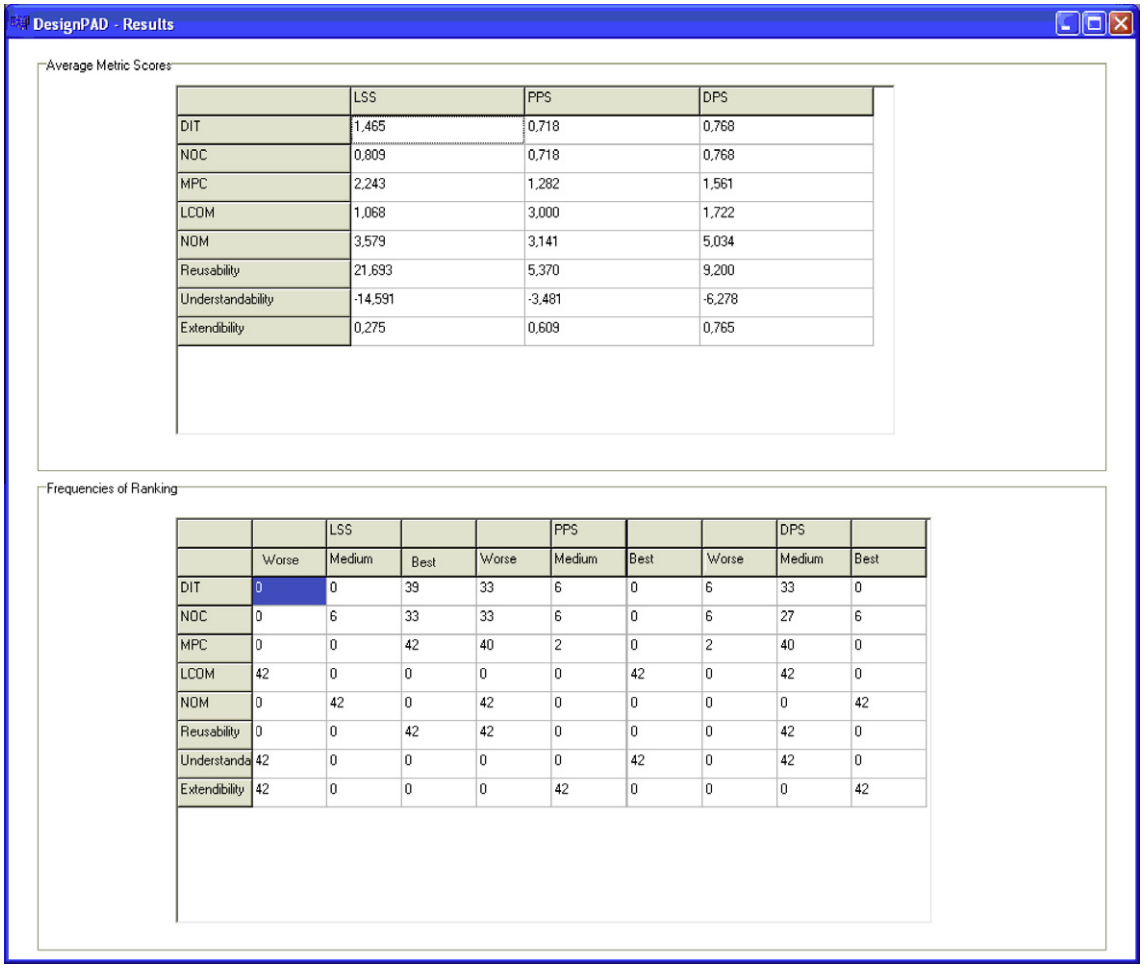


Fig. 5. DePAD – input form.

a design pattern produce more maintainable code than a simpler solution?" According to the above results there are several factors that the designer has to consider, such as pattern size and the most important quality attributes.



	LSS	PPS	DPS
DIT	1,465	0,718	0,768
NOC	0,809	0,718	0,768
MPC	2,243	1,282	1,561
LCOM	1,068	3,000	1,722
NOM	3,579	3,141	5,034
Reusability	21,693	5,370	9,200
Understandability	-14,591	-3,481	-6,278
Extendibility	0,275	0,609	0,765

	LSS			PPS			DPS		
	Worse	Medium	Best	Worse	Medium	Best	Worse	Medium	Best
DIT	0	0	39	33	6	0	6	33	0
NOC	0	6	33	33	6	0	6	27	6
MPC	0	0	42	40	2	0	2	40	0
LCOM	42	0	0	0	0	42	0	42	0
NOM	0	42	0	42	0	0	0	0	42
Reusability	0	0	42	42	0	0	0	42	0
Understanda	42	0	0	0	0	42	0	42	0
Extendibility	42	0	0	0	42	0	0	0	42

Fig. 6. DePAD – results form.

In this section, we describe five scenarios, we present the results of the DesignPAD tool and we discuss the results. The scenarios are summarized in Table 4. The scenarios have been created randomly and the five imaginary case studies aim at the demonstration of the methodology. The results of applying our methodology on the aforementioned scenarios are presented in Table 5. As mentioned in the methodology, a solution is considered better than the other two if achieves minimum values for the majority of the predictor metrics.

From the results of the Table 5 we observe that in three cases the pattern solution provides a more maintainable design, but there are cases, e.g. scenario [a] and [b], where the pattern is not the optimal solution.

Concluding, the results are in accordance with the controlled experiments in [59,69], where the authors suggest that there is no general answer about the effect of patterns on maintainability. Moreover, our results agree with [38] concerning the decoupling that Abstract Factory offers and additionally highlights the thresholds that concern cohesion and message passing metrics. Furthermore, concerning the Bridge pattern, the results on inheritance metrics that are described in [39] cannot be directly compared to those of our study, because in [39] the author has extended the system w.r.t. only one axis of change. However, both studies agree that there are certain cut off points w.r.t. inheritance metrics. Finally, our study has identified further thresholds concerning eight additional metrics than [39].

5.2. Design patterns and structural quality

This section of the paper discusses the findings of our study concerning RQ_3, i.e. the effect of the bridge design pattern on structural quality of a system [19,20]. The results of this case study are presented in AUTH/SWENG/TR-2011-10-01. A graphical overview of the results is presented in Figs. 7 and 8. Each graph depicts three scatter plots in a 3D space. For each scatter plot, the *x*-axis represents the n values, the *z*-axis the *m* values, whereas the *y*-axis represents the corresponding metric score. Each scatter plot represents one design solution. The range of the n and m variables has been set to be [2,40], considering that hierarchies of more than 40 subclasses have low probability to appear in real projects. The results concerning the CF [15] variable are shown in Fig. 7. Additionally, results on the LCOM metric are presented in Fig. 8.

The mathematical statements that are presented in AUTH/SWENG/TR-2011-10-01 can be used for comparing specific instances of designs but they cannot answer questions such as, "Which design produces higher cohesion between methods?" For this reason a set of paired sample *t*-tests has been performed over the abovementioned dataset. The paired-sample *t*-test will suggest whether the mean value of the metric scores of one design is statistically significantly higher or lower than another's. Table 6, presents the mean values of each metric score for all three design solutions. In addition to that, Table 7 presents the statistical signif-

Table 4
Case study scenarios.

Scenario Id	Candidate pattern	Pattern size (*n*,*m*)	Metrics under consideration
[a]	Bridge	(3,5)	All predictors
[b]	Bridge	(3,5)	SIZE1, RFC, DIT, DAC, LCOM
[c]	Bridge	(9,9)	SIZE1, WMPC, NOC, DAC, LCOM
[d]	Abstract Factory[a]	(2,6)	All predictors
[e]	Abstract Factory	(7,2)	NOM, SIZE2, RFC

[a] From the definition of AbstractFactory it becomes clear that the system has two major axes of change, i.e. new types of concrete Products and new types of actions. In the case of the pattern (*n*) represents the number of concrete products (ConcreteProduct1, ConcreteProduct2,....) and (*m*) represents the subcategories of each ConcreteProduct (ConcreteProduct11, ConcreteProduct12,...). In the literature solution [63] (*m*) corresponds to the number of doAction methods and (*n*) is represented by the cascaded if statement within each doAction method. Concerning the open source solution, (*n*) represents the number of concrete products and (*m*) represents the number of doAction methods. For more details see AUTH/SWENG/TR-2011-10-01.

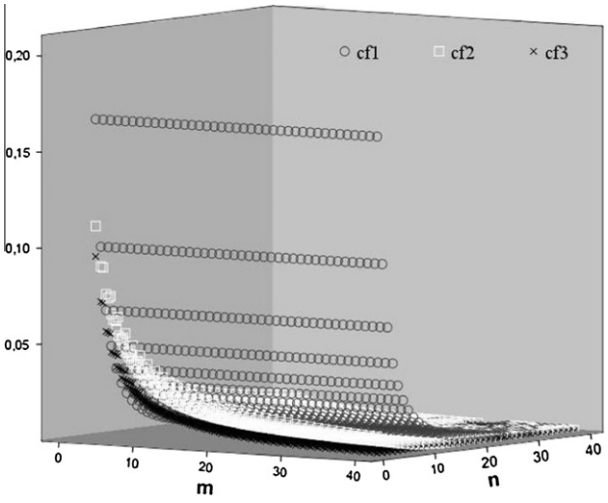


Fig. 7. Graphical representation of the CF scores of the three designs.

icance between the differences in the mean metric scores of the simpler solutions and the design patterns.

As it is observed in Table 7, all significance values equal to 0.000. This fact suggests that the ranking of designs with respect to each metric, as provided in Table 6 is statistically significant. Thus, according to coupling and cohesion, the pattern solution is more probable to produce better results. Concerning complexity, the literature solution provides the best results, closely followed by the pattern solution. With respect to inheritance and size, the open source design alternative is suggested as the best possible solution. Additionally, it is observed that the pattern design is ranked either first or second among the three designs; moreover in two out of three cases, where it lags compared to another design, it is quite close to the best value.

5.3. Design patterns and design quality

In AUTH/SWENG/TR-2011-10-01 we present several inequalities that suggest which of the three designs under study (i.e. Abstract Factory, Visitor and Bridge) is more beneficial w.r.t. each quality attribute. In this section we attempt to interpret and discuss why this design is better, and what this fact may imply. Concerning size (DSC metric), in two out of three patterns the literature simple solution produces the smallest design, with respect to number of classes. This happens because in these designs, several responsibilities are merged in one class, rather than decomposing them to several classes. Of course, such decisions have several trade-offs concerning cohesion and complexity attributes.

Similarly, the low number of classes that is employed in LSS has effect on hierarchies (NOH metric) and abstraction (ANA metric) of the design. On the contrary, we observe that DPS always offers the system the highest amount of hierarchies, which is expected to lead to enhanced extendibility. The encapsulation attribute (DAM metric) is equal in every design since all solutions follow the basic rule of object-oriented programming which suggests that every attribute of a class should be declared as private or protected.

With respect to the cohesion (CAM metric) of the systems, we observe that LSS are more prone to lack of cohesion, since they never produce a "best" design in this field. On the contrary, professional programmers and design patterns seem to spread the functionality of the system among the classes, which leads to better responsibility assignment and thus better cohesion among classes.

Similarly, concerning coupling (DCC metric and MOA metric) LSS appear to lag. The limited use of inheritance in LSS design leads to an increase in the ratio of the use of stronger associations, such as aggregation or composition. Considering PPS and DPS, the results appear divided, with some cut-off points where one solution becomes better than the other.

Additionally, we would expect that DPS would produce "best" results concerning inheritance for the majority of the patterns, but it is observed that the results are divided. A possible explanation is that patterns make extensive use of pure virtual methods, consequently they do not inherit many methods from their super-classes but they override them. However, when a designer chooses to employ an Abstract Factory rather than the simple solution the higher possible gain from inheritance use is benefited.

Table 5
Case study results.

Metric	[a]			[b]			[c]			[d]			[e]		
	LSS	OSS	DPS	LSS	OSS	DPS	LSS	OSS	DPS	LSS	OSS	DPS	LSS	OSS	DPS
DIT	1320	0600	0727	1320	0600	0727	–	–	–	0000	0750	1391	–	–	–
NOC	0720	0600	0727	–	–	–	0878	0818	0800	0000	0750	0870	–	–	–
RFC	5440	4600	5727	5440	4600	5727	–	–	–	3500	3125	1913	11.00	9500	2654
MPC	2080	1400	1455	–	–	–	–	–	–	1000	1000	0478	–	–	–
LCOM	1045	3000	1571	1045	3000	1571	1094	3000	2200	1000	1000	0000	–	–	–
WMPC	3400	6600	5455	–	–	–	3780	5000	3733	7500	2750	1652	–	–	–
DAC	0640	0200	0182	0640	0200	0182	0683	0091	0133	0500	0125	0087	–	–	–
NOM	3360	3200	4273	–	–	–	–	–	–	2500	2125	1435	7500	7250	2000
SIZE1	5360	11.20	6818	5360	11.20	6818	5463	8182	4600	8500	5375	4130	–	–	–
SIZE2	4240	4200	6091	–	–	–	–	–	–	3000	2250	1522	8000	7500	2077
#of MINs	3	6	1	2	2	1	1	0	4	2	0	8	0	0	3

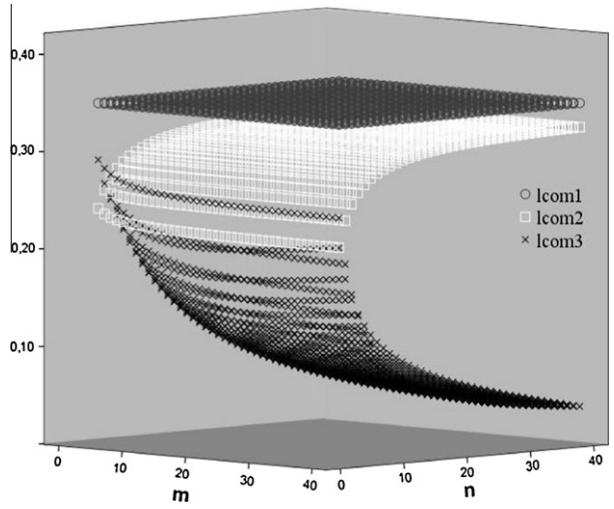


Fig. 8. Graphical representation of the LCOM scores of the three designs.

Table 6
Design solution metric mean value.

Metric	OSS	LSS	DPS
CF	0.158	0.078	0.038
WMPC	7.102	1.002	1.013
LCOM	0.350	0.309	0.098
DIT	1.872	2.781	1.920
NOC	23.000	485.00	45.000

Table 7
Object-oriented metrics considered.

Pair	Mean difference	Sig
CF_{oss}–CF_{dps}	0.0120	0.000
CF_{lss}–CF_{dps}	0.0040	0.000
CF_{oss}–CF_{lss}	0.0080	0.000
DIT_{oss}–DIT_{dps}	−0.0483	0.000
DIT_{lss}–DIT_{dps}	0.8608	0.000
DIT_{oss}–DIT_{lss}	−0.9091	0.000
$LCOM_{oss}$–$LCOM_{dps}$	0.2516	0.000
$LCOM_{lss}$–$LCOM_{dps}$	0.2111	0.000
$LCOM_{oss}$–$LCOM_{lss}$	0.0405	0.000
NOC_{oss}–NOC_{dps}	−22.0000	0.000
NOC_{lss}–NOC_{dps}	440.0000	0.000
NOC_{oss}–NOC_{lss}	−462.0000	0.000
$WMPC_{oss}$–$WMPC_{dps}$	6.0886	0.000
$WMPC_{lss}$–$WMPC_{dps}$	−0.0107	0.000
$WMPC_{oss}$–$WMPC_{lss}$	6.0933	0.000

Furthermore, concerning polymorphism (NOP metric) the LSS design produces the worst results due to the lack of inheritance. This fact obliged the developers to implement the dynamic behavior that is suggested by the requirements, through "cascaded if" and "switch" statements which increase the complexity of the systems.

Finally, professional programmers seem to use the lowest amount of methods (CIS metric), which leads to a small interface size and low complexity (NOM metric). On the contrary, design patterns and LSS employ larger class interfaces, which enhance messaging between classes, but as a side effect the complexity of the design increases.

Next, we describe six design scenarios, we present the results of the DesignPAD tool and we discuss them. The scenarios are summarized in Table 8. The scenarios have been created randomly and the six imaginary case studies aim at the demonstration of the methodology. The results of applying our methodology on the aforementioned scenarios are presented in Table 9. A solution is considered better than the other two if it achieves maximum values for the majority of the design quality attributes.

Table 8
Case study scenarios.

Scenario Id	Candidate pattern	Pattern size (n,m)	Attribute under consideration
[a]	Bridge	(3,5)	All Attributes
[b]	Bridge	(5,2)	Flexibility Extendibility
[c]	Visitor[a]	(9,9)	Understandablity Extendibility Flexibility
[d]	Visitor	(5,8)	Effectiveness Reusability Functionality
[e]	Abstract Factory	(2,6)	All Attributes
[f]	Abstract Factory	(7,2)	Understandablity Extendibility Flexibility

[a] Similarly to the other examples Visitor pattern has two major axes of change, i.e. new visitors and new concrete elements. In the case of the pattern and of the open-source solution (n) represents the number of concrete elements and (m) represents the number of concrete visitors. In the literature solution [18] (m) corresponds to the number of visitorCall methods and (n) represents the number of concrete elements. For more details see AUTH/SWENG/TR-2011-10-01.

From the results of Table 9 we observe that in two cases, i.e. [b] and [e] the pattern solution provides the design of best quality. Additionally, in cases [a] and [c], the pattern also leads the ranking of the designs, but this result is marginal. In case [f] all solutions are considered of equal quality. Finally, in scenario [d] we observe that the LSS solution produces a marginally better result than the other designs. In the cases when all solutions produce an equal count of best attributes, or one solution is marginally better than the others, the developer should consider the trade-offs that every solution offers.

Concluding, the results are in accordance with the controlled experiments in [59,69], where the authors suggest that patterns are not universally good or bad. However, we observe that the use of a pattern is most commonly producing the more extendible design. This means that design pattern solutions should be preferred for systems that are intended to be heavily reused and/or maintained. This is definitely an advantage of design patterns. On the other hand, in every case considered the pattern has never been suggested to be the most understandable solution. The results on other design attributes are divided.

6. Threats to validity

This section of the paper discusses possible threats to the validity of the paper. Firstly, even though the analytical method employed in the metric comparison ensures the accuracy of the procedure, the results on three design patterns cannot be generalized to the rest of the 23 design patterns that are described in [31], or any other type of design pattern. Another threat to the validity of the study is the fact that we have considered only two adhoc solutions. We have tried to adopt alternative solutions that have already been proposed and studied and that represent reasonable design decisions; however, there may be room for further alternative solutions to be examined. Thus, the results cannot be generalized to any other alternative design solution. Additionally, the study has not investigated possible interactions between patterns as mentioned in [55]. It is expected that each pattern coupling type demands different manipulation. Furthermore, the results of the case studies cannot be generalized to pattern variations. In such

Table 9
Case study results.

Attribute	Solution	[a]	[b]	[c]	[d]	[e]	[f]
Reusability	LSS	14.097	–	–	29.162	4.357	–
	OSS	4.135	–	–	5.120	5.098	–
	DPS	7.535	–	–	10.610	12.036	–
Flexibility	LSS	0.567	0.636	0.521	–	0.000	0.000
	OSS	0.500	0.429	0.364	–	1.063	0.389
	DPS	0.568	0.700	0.560	–	0.568	0.370
Understandability	LSS	−9.524	–	−34.901	–	−2.357	−1.056
	OSS	−2.687	–	−4.615	–	−3.786	−3.630
	DPS	−5.154	–	−9.401	–	−8.346	−8.969
Functionality	LSS	6.643	–	–	13.323	2.005	–
	OSS	2.176	–	–	2.582	2.851	–
	DPS	3.993	–	–	5.335	5.937	–
Extendibility	LSS	0.311	0.484	0.076	–	−0.250	−0.250
	OSS	0.550	0.607	0.659	–	0.875	0.444
	DPS	0.661	0.805	0.807	–	0.984	0.847
Effectiveness	LSS	0.737	–	–	0.676	0.300	–
	OSS	0.680	–	–	0.714	0.800	–
	DPS	0.755	–	–	0.779	1.066	–
# OF BEST ATTRIBUTES	LSS	2	0	1	2	1	1
	OSS	1	0	0	0	1	1
	DPS	3	2	2	1	4	1

cases, the pattern implementation will usually not exactly follow the diagram in the GoF book. Hence, the metric calculations will change. Thus the cut-off points extracted from our analysis are accurate only for patterns in their standard form. Moreover, the assumption that using already existing classes in pattern instances produces uniform effects on the quality of all design alternatives, needs to be investigated. Finally, the user of the method should also keep in mind the limitations of the set of metrics and/or quality models that he/she has used to associate design solutions to quality characteristics.

7. Future work

Thus, future work includes a replication of the research with a wider variety of patterns and design alternatives. Additionally, we plan to replicate the methodology by using a quality model that uses code metrics and extends the systems with more ways as described in [56]. Moreover, future work plans also involve the conversion of the DesignPAD tool into an Eclipse plug-in that will automatically receive size information from an already implemented pattern or alternative solution and confirm the correctness of the design solution, or suggest the best solution, with respect to specific design quality attributes. Furthermore we plan to expand our tool so as to support additional design patterns. Finally, we plan to investigate the possibility of employing the method of this study in design patterns that have not been referenced to be implemented with alternative solutions or others that aim at implementing a functional requirement.

8. Conclusions

Concluding, this paper suggested a methodology for exploring designs where design patterns have been implemented, through the mathematical formulation of the relation between design pattern characteristics and well known metrics, and the identification of thresholds for which one design becomes more preferable than another. This approach can assist goal oriented decision making, since it is expected that every design problem demands a specific solution, according to its special needs with respect to quality and its expected size. Our methodology has been used for comparing the quality of systems with and without patterns during their maintenance. Thus, three examples that employ design patterns have been developed, accompanied by alternative designs that solve the same problem. All systems have been extended with respect to their most common axes of change and eleven metric scores have been calculated as functions of extended functionality.

The results of the analysis have identified eight cut-off points concerning the Bridge pattern, three cut-off points concerning Abstract Factory and 29 cut-off points concerning Visitor. In addition to that, a tool that calculates the metric scores has been developed. The tool has been used so as to investigate several random scenarios that suggested that in most cases the pattern solution provides the most extendible design. However, this conclusion cannot be generalized for all patterns, quality attributes or pattern sizes. Thus, multi-criteria decision analysis is necessary, which will weight the importance of each quality attribute according to the designer's experience and provide a non-biased suggestion on the most fitting design decision.

References

[1] P.S.C. Alencar, D.D. Cowan, J. Dong, C.J.P. de Lucena, A pattern-based approach to structural design composition, 23rd International Computer Software and Applications Conference (COMPSAC'99), IEEE, Phoenix, Arizona, 25–26 October 1999, pp. 160–165.
[2] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language – Town, Buildings, Construction, Oxford University Press, New York, 1977.
[3] H. Albin-Amiot, P. Cointe, Y.-G. Gueheneuc, N. Jussien, Instantiating and detecting design patterns: putting bits and pieces together, International Conference on Automated Software Engineering (ASE' 01), ACM, 26–29 November 2001, San Diego, USA, pp. 166–173.
[4] A. Ampatzoglou, A. Chatigeorgiou, "Evaluation of object-oriented design patterns in game development", *Information and Software Technology*, Elsevier, 49 (5), pp. 445–454, 2007.
[5] G. Antoniol, R. Fiutem, L. Cristoforetti, Using metrics toidentify design patterns in object-oriented software, IEEE Proceedings of the 5th International Symposium on Software Metrics (METRICS 1998), IEEE Computer Society, 20–21 March 1998, Maryland, USA, pp. 23–34.
[6] F. Arcelli, S. Spinelli, Impact of refactoring on quality code evaluation, Proceeding of the 4th workshop on Refactoring tools (WRT '11), ACM, 21-28 May 2011, Honolulu, Hawaii.
[7] E.L.A. Baniassad, G.C. Murphy, C. Schwanninger, Design pattern rationale graphs: linking design to source, Proceedings of the 25th International Conference on Software Engineering, IEEE, Portland, Oregon, 03–10 May 2003, pp. 352-362.
[8] J. Bansiya, C. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transaction on Software Engineering 28 (1) (2002) 4–17.

[9] I. Bayley, H. Zhu, Formal specification of the variants and behavioural features of design patterns, Journal of Systems and Software 83 (2) (2010) 209–221.
[10] I. Bayley, H. Zhu, Specifying behavioural features of design patterns in first order logic, Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC '08), IEEE, Turku, Finland, 28 July-01 August 2008, pp. 203–210.
[11] J.M. Bieman, G. Straw, H. Wang, P.W. Munger, R.T. Alexander, Design patterns and change proneness: an examination of five evolving systems, IEEE Proceedings of the 9th International Software Metrics Symposium (METRICS 2003), IEEE Computer Society, 3–5 September 2003, Sydney, Australia, pp.40–49.
[12] J.M. Bieman, D. Jain, H.J. Yang, OO design patterns, design structure, and program changes: an industrial case study, IEEE Proceedings of the 17th International Conference on Software Maintenance (ICSM 2001), 7–9 November 2001, Florence, Italy, pp. 580–589.
[13] A. Birukou, A survey of existing approaches for pattern search and selection, Proceedings of the 2010 European Conference on Pattern Languages of Patterns (EuroPLoP' 10), ACM, 7–11 July 2010, Bavaria, Germany.
[14] A. Blewitt, A. Bundy, I. Stark, Automatic verification of design patterns in Java, Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM, Long Beach, CA, 07–11 November 2005, pp. 224–232.
[15] F. Brito e Abreu, The MOOD metrics set, Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95) – Workshop Metrics, August. 1995.
[16] G.E. Boussaidi, H. Mili, A model driven framework for representing and applying design patterns, 31st Annual International Computer Software and Applications Conference (COMPSAC'07), IEEE, Beijing, China, 24–27 July 2007, pp 97–100.
[17] M. Cartwright, M. Sheppard, An empirical investigation of an object-oriented software system, IEEE Transaction on Software Engineering 26 (8) (2000) 786–796.
[18] A. Chatzigeorgiou, Object-Oriented Design: UML, Principles, Patterns and Heuristics, Klidarithmos, Greece, 2005.
[19] S.R. Chidamber, C.F. Kemmerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[20] S.R. Chidamber, D.P. Darcy, C.F. Kemmerer, Managerial use of metrics for object oriented software: an exploratory analysis, IEEE Transactions on Software Engineering 24 (1) (1998) 629–639.
[21] M. Dagpinar, J.H. Jahnke, Predicting maintainability with object-oriented metrics – an empirical comparison, IEEE Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03), 13–17 November 2003, Victoria, Canada, pp. 155–164.
[22] D.P. Darcy, C.F. Kemmerer, S.A. Slaughter, J.E. Tomayko, The structural complexity of software: an experimental test, IEEE Transactions on Software Engineering 31 (11) (2005) 982–994.
[23] M. Di Penta, L. Cerulo, Y.G. Gueheneuc, G. Antoniol, An empirical study of relationships between design pattern roles and class change proneness, IEEE Proceedings of the 24th International Conference on Software Maintenance (ICSM 2008), 28 September–4 October 2008, Beijing, China, pp. 217–226.
[24] J. Dong, Adding pattern related information in structural and behavioural diagrams, Information and Software Technology 46 (5) (2004) 293–300.
[25] J. Dong, S. Yang, K. Zhang, Visualizing design patterns in their applications and compositions, IEEE Transactions on Software Engineering 33 (7) (2007) 433–453.
[26] J. Deng, E. Kemp, E.G. Todd, Managing UI pattern collections, Proceedings of the 6th ACM SIGCHI New Zealand Chapter's International Conference on Computer-Human Interaction: Making CHI Natural (CHINZ '05), ACM, Auckland, New Zealand, pp. 31–38.
[27] A.H. Eden, A. Yehudai, J. Gil, Precise specification on automatic application of design patterns, Proceedings of the 12th International Conference on Automated Software Engineering, ACM, Lake Tahoe, CA, 02-05 November 1997, pp. 143.
[28] E. Eide, A. Reid, J. Regehr, J. Lepreau, Static and dynamic structure in design patterns, Proceedings of the 24th International Conference on Software Engineering (ICSE'02), IEEE, Orlando, Florida, 19-25 May 2002, pp. 208–218.
[29] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.
[30] R.B. France, D.K. Kim, S. Ghosh, E. Song, A UML-based pattern specification technique, IEEE Transactions on Software Engineering 30 (3) (2004) 193–206.
[31] E. Gamma, R. Helms, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, Reading, MA, 1995.
[32] Y. G. Gueheneuc, H. Saharaoui, F. Zaidi, Fingerprinting design patterns, IEEE Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004), 8–12 November 2004, Delft, Netherlands, pp. 172–181.
[33] Y.G. Gueheneuc, J.Y. Guyomar'h, H. Saharaoui, Improving design pattern identification: a new approach and an exploratory study, Software Quality Journal 18 (1) (2010) 145–174.
[34] P. Gomes, F.C. Pereira, P. Paiva, N. Seco, P. Carreiro, J.L. Ferreira, C. Bento, Using CBR for automation of software design patterns, 6th European Conference on Advances in Case-Based Reasoning, Springer, Aberdeen, Scotland, pp. 534–548.
[35] G. Shu-Hang, L. Yu-Qing, J. Mao-Zhong, G. Jing, L. Hong-Juan, A requirement analysis pattern selection method for e-business project situation, IEEE International Conference on E-Business Engineering (ICEBE'07), IEEE, 24–26 October 2007, pp. 347–350.
[36] B. Henderson-Sellers, L. Constantine, I. Graham, Coupling, cohesion: towards a valid metrics suite for object-oriented analysis and design, Object-Oriented Systems 3 (3) (2002) 143–158.
[37] N.L. Hsueh, J.Y. Kuo, C.C. Lin, Object-oriented design: a goal driven and pattern based approach, Software and Systems Modeling 8 (1) (2009) 67–84.
[38] N.L. Hsueh, P.H. Chu, W. Chu, A quantitative approach for evaluating the quality of design patterns, Journal of Systems and Software 81 (8) (2008) 1430–1439.
[39] B. Huston, The effects of design pattern application on metric scores, Journal of Systems and Software 58 (2001) 261–269.
[40] Y. Kataoka, T. Imai, H. Andou, T. Fukaya, A quantitative evaluation of maintainability enhancement by refactoring, International Conference on Software Maintenance (ICSM'02), 3–6 October 2002, Montreal, Canada, pp. 576–585.
[41] F. Khomh, Y.G. Gueheneuc, Do design patterns impact software quality positively?, IEEE Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), 1–4 April 2008, Athens, Greece, pp.274–278.
[42] F. Khomh, Y.G. Gueheneuc, G. Antoniol, Playing roles in design patterns: an empirical descriptive and analytic study, IEEE Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009), 20–26 September 2009, Edmonton, Alberta, Canada, pp.83–92.
[43] D.K. Kim, R. France, S. Ghosh, E. Song, A role-based metamodeling approach to specifying design patterns, Proceedings of the 27th Annual International Conference on Computer Software and Applications, IEEE, Dallas, Texas, 03–06 November 2003, pp. 452.
[44] K. Kouskouras, A. Chatzigeorgiou, G. Stephanides, Facilitating software extension with design patterns and Aspect-Oriented Programming, Journal of Systems and Software 81 (10) (2008) 1725–1737.
[45] D.C. Kung, H. Bhambhani, R. Shah, G. Pancholi, An expert system for suggesting design patterns: a methodology and a prototype, Series in Engineering and Computer Science: Software Engineering with Computational Intelligence, Kluwer International, 2003.
[46] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer-Verlag, Berlin, Germany, 2006.
[47] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of Systems and Software 23 (2) (1993) 111–122.
[48] M. Lorenz, J. Kidd, Object-Oriented Software Metrics, Prentice Hall, New Jersey, USA, 2004.
[49] J.K.H. Mak, C.S.T. Choy, D.P.K. Lun, Precise modeling of design patterns in UML, Proceedings of the 26th International Conference on Software Engineering (ICSE'04), IEEE, Edinburgh, Scotland, 23–28 May 2004, pp. 252–261.
[50] R.C. Martin, Agile Software Development: Principles, Patterns and Practices, Prentice Hall, Upper Saddle River, NJ, 2003.
[51] T. Mens, T. Tourwe, A declarative evolution framework for object-oriented design patterns, Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), IEEE, Florence, Italy, 07–09 November 2003, pp. 570.
[52] T. Mikkonen, Formalizing design patterns, Proceedings of the 20th International Conference on Software Engineering (ICSE'98), IEEE, Kyoto, Japan, 19–25 April 1998, pp. 115–124.
[53] T. Muraki, M. Saeki, Metrics for applying GOF design patterns in refactoring processes, ACM Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE' 2001), 10–11 September 2001, Vienna, Austria, pp. 27–36.
[54] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (4) (1976) 308–320.
[55] W. McNatt, J. Bieman, Coupling of design patterns: common practices and their benefits, IEEE Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC 2001), 8–12 October 2001, Chicago, USA, pp. 574–579.
[56] T.H. Ng, S.C. Cheung, W.K. Chan, Y.T. Yu, Do maintainers utilize deployed design patterns effectively?, IEEE Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 20–26 May 2007, Washington, USA, pp.168-177.
[57] B.C.d.S. Oliveira, M. Wang, J. Gibbons, The visitor pattern as a reusable, generic, type-safe component, Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA 2008), ACM, Nashville, Tennessee, 19–23 October 2008, pp. 439–456.
[58] N. Pettersson, Measuring precision for static and dynamic design pattern recognition as a function of coverage, Proceedings of the third international workshop on Dynamic analysis (ICSE'05), IEEE, St. Louis, Missouri, 17 May 2005, pp. 1–7.
[59] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, L.G. Votta, A controlled experiment in maintenance comparing design patterns to simpler solutions, IEEE Transactions on Software Engineering 27 (12) (2001) 1134–1144.
[60] M. Riaz, E. Mendes, E. Tempero, A systematic review on software maintainability prediction and metrics, 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09), 15–16 October 2009, Lake Buena Vista, Florida, USA, pp. 367–377.
[61] L. Rising, The Pattern Almanac, Addison-Wesley Longman Publishing, 2000.
[62] L. Samoladas, I. Stamelos, L. Angelis, A. Oikonomou, Open source software development should strive for even greater code maintainability, Communications ACM 47 (10) (2004) 83–87.
[63] A. Shalloway, J.R. Trott, Design Patterns Explained: A New Perspective on Object-oriented Design, Addison-Wesley Professional, Reading, MA, 2005.

[64] N. Soundarajan, J.O. Hallstrom, Responsibilities and rewards: specifying design patterns, Proceedings of the 26th International Conference on Software Engineering (ICSE'04), IEEE, Edinburgh, UK, 23–28 May 2004, pp. 666–675.
[65] T. Taibi, D.C.L. Ngo, Formal specification of design pattern combination using BPSL, Information and Software Technology 45 (3) (2003) 157–170.
[66] N. Tsantalis, A. Chatzigeorgiou, G. Stephanidis, I. Deligiannis, Probabilistic evaluation of object-oriented systems, IEEE Proceedings of the 10th International Symposium on Software Metrics (METRICS'04), 11–17 September 2004, Chicago, Illinois, USA, pp. 26–33.
[67] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, Transactions on Software Engineering 35 (3) (2009) 347–367.
[68] A. Van Koten, A.R. Gray, An application of bayesian network for predicting object-oriented software maintainability, Information and Software Technology 48 (1) (2006) 59–67.
[69] M. Vokác, W. Tichy, D.I.K. Sjøberg, E. Arisholm, M. Aldrin, A controlled experiment comparing the maintainability of programs designed with and without design patterns – a replication in a real programming environment, Empirical Software Engineering 9 (2003) 149–195.
[70] M. Weiss, A. Birukou, Building a pattern repository: benefitting from the open, lightweight, and participative nature of wikis, International Symposium on Wikis (WikiSym), ACM, 21–23 October 2007, Montreal, Canada.
[71] M. Weiss, H. Mouratidis, Selecting security patterns that fulfill security requirements, 16th International Conference on Requirements Engineering (RE'08), IEEE, 8–12 September 2008, Barcelona, Spain, pp 169–172.
[72] L. Welicki, J.M.C. Lovelle, L.J. Aguilar, Patterns meta-specification and cataloging: towards a more dynamic patterns life cycle, International Workshop on Software Patterns, IEEE, 23–27 July 2007, Beijing, China.
[73] T. Winn, P. Calder, Is this a pattern?, IEEE Software 19 (1) (2002) 59–66
[74] Wolfram Mathematica, Reduce–Wolfram Mathematica 7 Documentation. <http://reference.wolfram.com/mathematica/ref/Reduce.html> 2010.
[75] P. Wendorff, Assessment of design patterns during software reengineering: lessons learned from a large professional project, IEEE Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001), 14–16 March 2001, Lisbon, Portugal, pp.77–84.
[76] U. Zdun, P. Alexiou, C. Hentrich, S. Dustdar, Architecting as decision making with patterns and primitives, Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge (ICSE'08), IEEE, Leipzig, Germany, 10–18 May 2008, pp. 11–18.
[77] U. Zdun, Systematic pattern selection using pattern language grammars and design space analysis, Software: Practice & Experience 37 (9) (2007) 983–1016.
[78] Y. Zhou, H. Leung, Predicting object-oriented software maintainability using multivariate adaptive regression splines, Journal of Systems and Software 80 (2007) 1349–1361.
[79] M. Ziane, A transformational viewpoint on design patterns, Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ACM, Grenoble, France, 11–15 September 2000, pp. 273.
[80] O. Zimmermann, U. Zdun, T. Gschwind, F. Leymann, Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method, Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), IEEE, 18–22 February 2008, Vancouver, Canada.