The Journal of Systems and Software 84 (2011) 2265–2283

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

An empirical investigation on the reusability of design patterns and software packages

Apostolos Ampatzoglou*, Apostolos Kritikos, George Kakarontzas, Ioannis Stamelos

Department of Informatics, Aristotle University, Aristotle University Campus, 54124 Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 22 December 2010
Received in revised form 17 June 2011
Accepted 18 June 2011
Available online 24 June 2011

Keywords:
Design patterns
Design
Quality
Reusability
Empirical approach

ABSTRACT

Nowadays open-source software communities are thriving. Successful open-source projects are competitive and the amount of source code that is freely available offers great reuse opportunities to software developers. Thus, it is expected that several requirements can be implemented based on open source software reuse. Additionally, design patterns, i.e. well-known solution to common design problems, are introduced as elements of reuse. This study attempts to empirically investigate the reusability of design patterns, classes and software packages. Thus, the results can help developers to identify the most beneficial starting points for white box reuse, which is quite popular among open source communities. In order to achieve this goal we conducted a case study on one hundred (100) open source projects. More specifically, we identified 27,461 classes that participate in design patterns and compared the reusability of each of these classes with the reusability of the pattern and the package that this class belongs to. In more than 40% of the cases investigated, design pattern based class selection, offers the most reusable starting point for white-box reuse. However there are several cases when package based selection might be preferable. The results suggest that each pattern has different level of reusability.

1. Introduction

The fact that open source software code reuse is being increasingly adopted by software companies and individual developers becomes apparent if we take under consideration the continuous growth of the free libre open source software (FLOSS) community. Reuse of OSS components in other OSS projects is intense: the reuse of code from 1311 leading OSS projects in other OSS projects represents 316,000 staff years and tens of billions of dollars in development costs[1] and OSS components are reused in thousands of projects (e.g. log4j is used in more than 5500 projects).[2] OSS software collectively represents an extremely valuable asset with estimations of the total development cost of OSS software at more than 387 billion dollars.[3] Additionally, in Li et al. (2009) the authors report that in 2007 over half of software developers used a part of open source projects or OSS components off the self (COTS) in their most recent projects.

Although, the Bazaar approach[4] in open-source development and reuse seem to be working pretty well, the more OSS component reuse becomes an established approach, the more its process needs to be analyzed and eventually lean on a concrete definition. In Ajila and Wu (2007), the authors conducted an empirical study which suggested that an organization can have important economic gains in terms of productivity and product quality, if it implements OSS components reuse in a systematic way. Additionally, the need for systematic application of OSS reuse is referenced in Morad and Kuflik (2005). The research of the state of the art on component based software engineering is thoroughly described in Brown and Wallnau (1998), Crnkovic et al. (2006), Crnkovic and Larsson (2002) and Crnkovic et al. (in press).

In the literature software reuse appears in two major forms, systematic and opportunistic reuse (Jansen et al., 2008; Morison et al., 2000). However, the results on the most fitting practice are controversial. Large organizations report on employing more formalized methods and software product lines, whereas small and medium size companies perform more adhoc reuse (Henry and Faller, 1995; Jansen et al., 2008; McConnell, 1996). In fact, when reusing open source code many developers reuse code opportunistically by copying and pasting classes or packages to their own projects. In Chang and Mockus (2008) and Mockus (2007), it is suggested that in FreeBSD, i.e. a well known operating system, about

* Corresponding author.
E-mail addresses: apamp@csd.auth.gr (A. Ampatzoglou), akritiko@csd.auth.gr (A. Kritikos), gkakaron@csd.auth.gr (G. Kakarontzas), stamelos@csd.auth.gr (I. Stamelos).

[1] http://www.blackducksoftware.com/news/releases/2009-03-30.
[2] http://www.blackducksoftware.com/news/releases/2008-12-09.
[3] http://www.blackducksoftware.com/news/releases/2009-04-14.

[4] http://www.catb.org/esr/writings/cathedral-bazaar/cathedral-bazaar/.

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.06.047

43% of the classes have been reused from other projects and that OSS reuse rates are extremely high. More specifically, 53% of the projects have performed reuse activities in 30% of their development process and that 49% of projects have reused more than 80% of their code. Additionally, in Mockus (2007) it is suggested that most reused units have gone through major or minor modifications in order to be adopted in the target project. The fact that the reused artifacts have been modified before being adapted in the target system suggests that white-box reuse techniques have been employed. Although not systematic this form of reuse is very frequent and presents its own challenges.

In this work we study and compare several reuse "chunks", in the context of adhoc reuse, in order to identify the preferable way of selecting groups of classes from the source project, so as to easily adjust them in the target project. The term preferable suggests that the selected set of classes has such structural characteristics that have been identified as positively correlated to white-box reuse. An extended discussion on the selected characteristics is presented in Section 2.2. This approach is one of the first steps of creating reusable components from source code originating from an open-source software project. The authors in McCormack et al. (2008) compared similar systems in size which provided the same (more or less) functionality. To overcome the problem of closed source code unavailability, they used now open source projects, that used to be closed source, and used their first open source release as an approximation of the Closed Source Software (CSS). They reported considerable differences in propagation costs (i.e. probable impact of changes) favoring the OSS projects. This is attributed to the fact that OSS projects are more distributed and therefore the architecture is considerably more modular. To the extent that modularity is important to software reuse it is expected that OSS code will be more reusable. This is supported also by concrete statistics provided by Black Duck that we mentioned earlier. However given that there is a clear incompatibility between the organizational structure of CSS and OSS projects which is reflected in the software architecture with impressive differences in the propagation costs of functionally similar systems (Mockus, 2007), it is an open research question if the OSS code can be reused in CSS projects without significant re-modularization and/or re-architecting. Although, white-box reuse is considered an inferior type of reuse than black-box reuse, the abovementioned observation and the extent of white-box reuse in OSS, enforce the significance of the white-box reuse research for OSS software reuse and validates the approach taken in this work for looking at alternative reuse granules (classes vs. packages vs. patterns) as opposed to using by default the package reuse granule which is the main architectural element in OO systems.

Another state-of-the-art software technique that provides solutions to common design problems is design patterns. By the employment of design patterns, the quality of the system under development is expected to improve while, at the same time, the whole architecture of the system becomes more adaptable and extensible. Design pattern reusability can be perceived in two ways, (a) reusing the idea of a pattern and (b) reusing the source code of a design pattern instance. Reusing the idea of patterns can be employed in component development, when the development team wants to use a pattern in order to solve a common design problem through a well documented design solution, while they develop a component from scratch. However, when reusing pattern instances the idea is not to reuse the rationale of the pattern, but the code that instantiates the solution. Of course, in such cases the pattern instance has to also (at least partially) fulfill the functional requirements of the target system. Developers are not particularly interested in reusing code that applies a pattern more than they are in reusing any code that fulfills their functional needs. However, it has been suggested that code reuse often entails adaptation (Bosch, 1999; Hölzle, 1993). In addition, recent empirical studies reveal that maintenance (including adaptive maintenance) is improved by the identification of design patterns (Prechelt et al., 2002; Scanniello et al., 2010), since developers recognize the roles that the different objects play in a complex interaction. Since (a) maintenance is improved with the identification of design patterns and (b) reusing code often requires adaptation, as in adaptive maintenance, it is natural to assume that reusing unfamiliar code is also improved with design pattern identification. But this only establishes a positive relation between design pattern identification and code reuse at the cognitive level. Reusers have a shorter cognitive distance to cover if design patterns are identified. Our work asks the complementary and currently unanswered question: "Besides comprehending, is it also easier to reuse the design pattern code at a technical level than it is to reuse alternative granules and more specifically packages or classes?" In this work we try to answer this question from a purely technical standpoint by (a) statically analyzing the source code of the alternative reuse granules (i.e. design patterns, packages and classes), (b) assessing their reusability in accordance with a well-established reusability assessment model (Bansiya and Davis, 2002), and (c) comparing the reuse granules' reusability assessments. Therefore the research question that we examine applies *after* a relevant class has been identified which provides the required functionality and concerns the reuse granule so that the reusability of the selected granule is improved in relation to the selection of the isolated class.

In Crnkovic et al. (2002) it is suggested that design patterns can be used as pre-existing components, in cases that the functionality of the pattern instance is relevant to the desired functionality of the target system. In many real cases, the attempt to identify a reuse chunk, points to a class that provides part of the desired functionality. If this class participates in a design pattern in the source project, then the reuser has three major reuse alternatives, to reuse the class, to reuse the pattern or to reuse the package where the class belongs to. Although, functionality is the key decider for reuse in the first step of the process, i.e. the identification of the reusable unit, the selection according to some quality attributes, such as reusability, is a key decider for selecting a component among functionally equivalent reuse candidates. In order to conduct an empirical study in a holistic way, we compared the reusability of the classes participating on design patterns with the reusability of the pattern itself (i.e. the collaboration of classes that implement a design pattern) and the reusability of the package to which the classes are included.

The rest of the paper is organized as follows. In Section 2 we provide background information for the basic terms discussed in this work. In Section 3 we analyze the methodology we followed in order to be able to answer the research questions we present in the same section. In Section 4 we provide the statistical analysis conducted to the data we collected. In Section 5 we discuss the results of the statistical analysis of the previous section. In Section 6 we speculate on threats to validity. Finally, in Section 7 we conclude our work by summarizing our findings and we refer to possible future work.

2. Background information

This section of the paper deals with presenting an overview of the research state of the art on component based software development, on measurements of software reusability, on component selection strategies, and finally on design patterns.

2.1. Component based software engineering

Component based software engineering (CBSE) focuses on the development of components in order to enable their reuse in more systems rather than only to the original one for which they

have been implemented in the first place (i.e. development for reuse) and the development of new systems with reusable components (i.e. development with reuse). In Ajila and Wu (2007) the authors suggest that reuse can occur in many levels on granularity, which could be a few lines of code, methods, classes or whole systems. Outside systems built on a certain component-based technology, component is understood as a general term and in the literature components have been related to packages, patterns and objects.

In Franch and Carvallo (2003), components are referred as packages; the authors make clear that these packages are essential in commercial off-the-self software (COTS software). This realization strengthens the position that components are very important in the software development process and as such, measuring their reusability can lead to faster and effective development of higher quality software.

Additionally, in Crnkovic et al. (2002) and Szyperski (1997) a component is described as a unit of composition that can be deployed independently and be adapted in a different system. A very important characteristic of a "useful" component is the decoupling of component interface from component implementation. In Crnkovic et al. (2002) the authors suggest that design patterns can be considered in CBSE with two perspectives (a) design patterns can be used in CBSE design, when reusable units should be identified as pre-existing components, and (b) develop components based on design patterns in order to adapt the pattern mechanism so as to increase component cohesion and decrease component internal coupling. Design pattern size is usually smaller than this of traditional components. However, the term "component", both in literature and in practice, is often used to denote any software part and not necessarily an architectural unit. For example in component models such as JavaBeans and Enterprise Java Beans the component is just a class. In Component Object Model (COM) and CORBA Component Model (CCM) a component is an object, whereas in SOFA, PECOS and Pin it is an architectural unit (Lau and Wang, 2005). Design patterns considering size, as a collaboration of classes, are larger than classes/objects and smaller than architectural components. Therefore they can be considered as a starting point for the derivation of architectural components in the context of white-box reuse. The fact that pattern application increases cohesion and decreases coupling is supported by several studies (Ampatzoglou and Chatzigeorgiou, 2007; Geuheneuc et al., 2004; Hsueh et al., 2008; Huston, 2001; Kouskouras et al., 2008). However, there are patterns, such as *Visitor* and *Observer*, which might introduce additional coupling. On the other hand, such patterns have a positive impact on other important quality attributes like reuntime flexibility.

Furthermore, the term class and component are often considered synonymous or very similar in existing component models (e.g. Java Beans and Enterprise Java Beans) (Lau and Wang, 2005). In his seminal work however, Szyperski (Szyperski, 1997) carefully makes the distinction between classes and components. Typically, a component consists of one or more classes, it can be however also implemented in a completely different technology as long as it is an independent unit of deployment which provides its services through a contractual interface and has explicit context dependencies only. Furthermore, a component may contain several more elements, other than classes even when it is implemented in an Object-Oriented language, such as global variables, images, html files and in general all artifacts that are useful for the component's provided services. Components therefore, are not only development artifacts. Components (and their connectors) exist as such during the execution of the system. Furthermore components can be versioned independently, with the same component existing side-by-side with other components and even with another version of itself if this is required by the installed applications (i.e. side-by-side versioning). Finally components can be upgraded dynamically during the system operation. Thus, classes and components are at different lifecycle levels, since components are deployment units, whereas classes are development artifacts and objects are notions of instantiation (Szyperski, 1997).

2.2. Software reusability

The selection of the group of classes as a component off-the-self requires the evaluation of several aspects of candidate components (Franch and Carvallo, 2003; Kontio, 2006). In Andreou and Tziakouris (2007), Cho et al. (2001), Fahmi and Choi (2008) and Yu et al. (2009) the authors suggest that one prominent way to select packages is the selection according to the packages' quality characteristics. In this study we selected to investigate several class selection alternatives with respect to their reusability. According to Bansyia et al. "Software reusability reflects the presence of object-oriented characteristics that allow a system to be reapplied to a new problem without significant effort" (Bansiya and Davis, 2002). After reviewing the literature we identified several ways to assess the reusability of a class or a system. In Table 1, we can see an overview of structural quality attributes that are reported to be important concerning the reusability of a system.

In Bansiya and Davis (2002), Bansyia et al. proposes a model (QMOOD) for calculating software reusability from low level quality metrics, at an early design stage. More specifically, the authors propose linear equations that can predict several high level quality attributes. The proposed hierarchical model is validated through an experiment with professional software evaluators.

Additionally, in Barnard (1998) the author provides thresholds on both code and styling attributes which, when surpassed, the reusability of the system becomes more difficult. However, the automatic application of the model, although rigorously validated, is not possible because of several abstract metrics, such as "meaningful attribute name". In Gui and Scott (2007) the correlation between component reusability and various coupling metrics is discussed. Furthermore the authors of Andreou and Tziakouris (2007) propose the quality evaluation of components based on ISO 9126, but the approach they suggest is qualitative and therefore not easily automated.

In Washizaki et al. (2003), Washizaki et al. suggests that several heuristics, such as existence of meta-information and component observability, could prove useful in measuring the reusability of software components. Similarly to (Barnard, 1998), using this approach to automatically evaluate system reusability is not possible. Finally, Sandlhu et al. (2009) proposes the combination of several metrics through a neural network, in order to assess the reusability of object-oriented software systems. However, this approach is not useful for the nature of our study, which needs numerical data.

Concluding, in our study we selected to use the QMOOD model (Bansiya and Davis, 2002) because it appears to be thoroughly validated, it assesses software reusability from metric scores that can be automatically calculated and it does not involve subjective parameters. Additionally, the QMOOD model is based on well studied metrics used in many published works, in well known software engineering journals (Counsell et al., 2006; Etzkorn et al., 2004; Genero et al., 2007; Hsueh et al., 2008; Khomh and Gueheneuc, 2008; Marcus et al., 2008; Plague et al., 2007). Additionally, the reusability as defined and calculated in QMOOD, takes into account structural quality characteristics such as coupling and cohesion that are very important when applying white-box reuse. A decoupled and highly coherent component is expected to be more maintainable and easier to adapt. According to QMOOD software reusability

Table 1
Structural quality characteristics influencing reusability.

Attribute	Effect	Studies
Coupling	–	Bansiya and Davis (2002), Barnard (1998), Gui and Scott (2007) and Sandlhu et al. (2009)
Cohesion	+	Bansiya and Davis (2002) and Sandlhu et al. (2009)
Messaging	+	Bansiya and Davis (2002)
Size	+	Bansiya and Davis (2002)
Inheritance	–	Barnard (1998) and Sandlhu et al. (2009)
Complexity	–	Barnard (1998) and Sandlhu et al. (2009)

is calculated as follows:

$$\text{reusability} = 0.25 \times \text{cohesion} + 0.5 \times \text{messaging} + 0.5 \times \text{size} - 0.25 \times \text{coupling} \quad (1)$$

At component level, we calculate reusability as the average reusability of all classes that participate in the component.

$$\text{reusability} = \frac{\sum_{i=1}^{NOC} \text{reusability\_of\_class\_}i}{NOC} \quad (2)$$

2.3. Design patterns

Design patterns have been introduced in software engineering literature as elements of reuse (Gamma et al., 1995). The notion of patterns in software development represents a collection of well-known design solutions to common design problems. In this paper we investigate whether selecting a group of classes that are based on design patterns offers enhanced white-box reuse opportunities rather than software packages and classes. Thus, we investigate if a reuser should select a group of classes that participate in a design pattern, alter them and produce a reusable component, rather than attempting to componentize complete software packages or classes.

Furthermore Conway's law suggests that "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" (Conway, 1968). Although Conway's law was not verified at the time of its publishing it was heavily cited for decades and recent studies from Harvard Business School (McCormack et al., 2008) and Microsoft research (Agape et al., 2008) confirm it. In the context of reuse this has significant implications for selecting software packages as reuse granules, since packages reflect the structure of a software system, and according to Conway's law this reflects the structure of the organization that produced the system. Reuse however occurs at a different organization with a different structure. Consequently using packages as reuse granules may be inappropriate for external reuse of software in different organizations and alternative reuse granules should be considered.

The possibility of creating software components from design patterns was introduced in 2006 by Meyer et al. (Arnout and Meyer, 2006; Meyer and Arnout, 2006). In Meyer and Arnout (2006) the authors suggest that only two design patterns are not componentizable. The componentization of the Visitor and the Factory Method patterns are thoroughly discussed in Brown and Wallnau (1998) and Arnout and Meyer (2006), respectively. Additionally, design pattern componentization offers faithfulness, completeness, simplicity, usability, ease of learning, type safety and system performance (Meyer and Arnout, 2006).

Furthermore, in Khomh and Gueheneuc (2008) Khom et al. discuss the effect of design pattern application in several external qualities attributes, through a survey conducted on professional software engineers. The results of the study imply that the employment of eleven design patterns has a positive effect on systems reusability, whereas twelve suggest the opposite. More specifically, the patterns that are reported to be beneficial appear to be Abstract Factory, Factory Method, Prototype, Adapter, Composite, Proxy, Chain of Responsibility, Interpreter, Iterator, Observer and Template Method. A possible weakness of this survey is that neutral opinions are considered as negative and therefore marginal results should be cautiously adopted. On the contrary, in Wydaeghe et al. (1998) the authors underline that Bridge and Façade have a very positive impact on system reusability; Observer, Visitor and Iterator also have positive impact while Chain of Responsibility has no noticeable effect on software reusability.

3. Methodology applied in the empirical study

The aim of this study is to compare the reusability of different reuse granules: classes vs. patterns vs. packages.[5] In order to achieve this goal, we have conducted a case study according to the guidelines described in Kitchenham et al. (1995). More specifically, the suggested steps are listed below:

- Define Research Questions
- Select Projects
- Identify the Method of Comparison
- Minimize the Effect of Confounding Factor
- Plan the Case Study
- Monitor the Case Study Against Plan
- Analyze and Report the Results

The research questions of this study are defined in Section 3.1. In Section 3.2, we describe the case study plan; we discuss the selection of projects and the confounding factors of this research. Furthermore, Section 3.3 deals with the description of the dataset and the methods for comparison. As mentioned before, Section 6 presents possible threats to validity that arose from monitoring the case study against the research plan. Finally, in Section 4 we present the procedure and the results of the statistical analysis and in Section 5 we discuss the results, with respect to the research questions.

3.1. Research questions

The main motivation of our study is to compare the reusability of classes, patterns and packages. The first research question (***RQ1***) that the paper attempts to answer can be described by the following scenario: "A developer wants to implement a specific requirement. He identifies a class that provides the main functionality that he wants to implement. This class happens to participate in a design pattern. Which classes should be selected, modified and reused in the final project?" In our research we investigated four alternatives for the reuser:

1. Select only the class that he is interested in [further reference as: *Alternative A_1-Class Based Selection*].

[5] By the term package we refer to a set of classes that is created by developers, in order to group collaborating classes.

2. Select the pattern that the class belongs to (i.e. all collaborating classes in the context of the pattern implementation) [further reference as: *Alternative A_2-Pattern Based Selection*].
3. Select the package that the class belongs to [further reference as: *Alternative A_3-Package Based Selection*].
4. Select all packages to which every pattern participant[6] belongs to [further reference as: *Alternative A_4-Multi-Package Based Selection*].

For example, let us suppose that 4 classes, i.e. A, B, C and D, participate in one design pattern. Class A (cA) belongs to package pA, class B (cB) belongs to package pB, class C (cC) belongs to package pC and class D (cD), belongs to package pA. For this pattern we would investigate the scenarios of Table 2.

The four selection alternatives represent choices available to the developer in relation to two distinct axes: (a) The developer is aware of a pattern existing in the structure of the system connecting the class he is interested in with other classes (i.e. the pattern's participants), or (b) the developer is not aware of such a pattern. The developer in the first case would either choose the class or the package in which the class belongs to, i.e. alternatives A_1 and A_3. In the pattern awareness case the developer could choose all four alternatives since the existence of the pattern now represents an interesting alternative. So besides the class and the package of the class we also examine the pattern and the "all packages of the pattern participants" choices. In cases when two solutions are identical, e.g. a case when a design pattern consists of classes which are placed in one package and that package includes classes only from that pattern; it is obvious that the reusability of the two alternatives is equal. Thus, in such cases the alternatives cannot be distinguished.

Additionally, another alternative would be to select all classes that are statically dependent to the selected class. However, such an approach might create class sets that do not clearly represent OO structural units and such a selection would lead to a set of classes that are completely different in size and that are not easily described. Thus, the practical benefits from such a procedure would be limited. Furthermore, the dependencies among classes are considered in the model. The problem with static dependencies in large Object-Oriented systems is that they are many and they are indistinguishable in the sense that important dependencies look the same as unimportant ones. Also for reuse it is important to select with one class other classes which are essential for its correct usage. In fact what we need here is the "uses" relation which describes the "Uses Style" of the module view (Clements, 2002). In this style classes use other classes if they need them for their operation. The authors observe that a class may use another and not depend on it statically, e.g. the other class provides a value in a file that the first class needs, and/or a class may have a static dependency on another class that does not need for its operation. For example a class may call on a logging service but does not need the logging service to provide its own services. Finally, the proposed method in this study is applicable during the coding of a software system and concerns reuse at the implementation level. However the selection of reuse granules is also affected by other concerns such as independent versioning and upgrades, which are considered during other development lifecycle phases, mainly during the architecture development.

Consequently, we preferred to limit our study in classes, patterns and packages that are well defined structural OO units. Additionally, the fact that units, which are strongly dependent to other units are not preferable for reuse, will be taken into account because the selected reusability model considers coupling metrics in the calculation of reusability. At this point it is necessary to clarify that, with the design pattern reuse statement we do not refer to the reuse of pattern rationale, but to the reuse of pattern implementation, i.e. the classes that implement a particular pattern. RQ_1 is summarized as follows:

RQ1: Which structural unit is more reusable (a class, a pattern or a package)?

Furthermore, our study attempts to answer three research questions (RQ):

RQ2: Is the selection of the most reusable set of classes correlated to the pattern type?
RQ3: Is the selection of the most reusable set of classes correlated to the number of the pattern's participants?
RQ4: Is the selection of the most reusable set of classes correlated to the number of packages that are involved in the pattern?

In order to explore RQ1 the following null hypotheses have been stated and investigated, according to the aforementioned definitions of alternatives.

$H_{0(1)}$: Alternative A_1 offers the most reusable selection of classes
$H_{0(2)}$: Alternative A_2 offers the most reusable selection of classes
$H_{0(3)}$: Alternative A_3 offers the most reusable selection of classes
$H_{0(4)}$: Alternative A_4 offers the most reusable selection of classes

3.2. Case study plan

In Basili et al. (1986), the authors suggest that before conducting an empirical study, the research team should prepare a thorough study plan. In this case study the plan involved a seven (7) step procedure:

a) choose some open source projects
b) perform pattern detection for every selected project
c) for every pattern find all the classes that participate in it
d) for every class that participates in each pattern create a pool of available set of classes that include it, according to the four alternatives mentioned in Section 3.1
e) for every available set of classes calculate its reusability according to the QMOOD model mentioned in Section 2.1
f) tabulate data
g) analyze data with respect to the research questions

The subjects of this research are one hundred of the most successful open source projects. The selected projects had to fulfill two criteria in order to automate the execution of steps (b)–(e), due to limitations of pattern detection tool (Tsantalis et al., 2006). The software (a) had to be written in java and (b) provide a jar executable file. The projects that have been investigated in our study are presented in the web.[7]

In any empirical study, factors, other than the independent variables, which influence the value of the dependent variable, are characterized as confounding factors. A possible confounding factor that is expected to affect the reusability of any set of class is its functionality, in the sense that groups of classes that implement certain reusable functional requirements are more likely to being reused. However, in the current scenario, we assume that

[6] As pattern participant we mean every class that plays a specific role in a design pattern (Fahmi and Choi, 2008). A definition of the pattern participants is given in (Harrison and Avgeriou, 2010).

[7] http://sweng.csd.auth.gr/apamp/material/jss_projects.doc.

Table 2
Class selection alternatives.

Class selection alternatives	Reused class			
	A	B	C	D
Alternative A_1	cA	cB	cC	cD
Alternative A_2	cA, cB, cC,cD	cA, cB, cC,cD	cA, cB, cC,cD	cA, cB, cC,cD
Alternative A_3	pA	pB	pC	pA
Alternative A_4	pA, pB, pC	pA, pB, pC	pA, pB, pC	pA, pB, pC

a developer has identified a class that fits the major functionality that he wants to reuse and he desires to evaluate all possible structural units that this class belongs to, in order to find and use the most reusable set of classes. Therefore our results take into account reusability issues, other than the functional fitness of the selected class, because all structural units that this class belongs to, offer the desired functionality.

3.3. Data analysis method

The dataset that has been tabulated after step (f) of the research plan consists of 27,461 rows, one for every pattern participant that can be reused, and eight (8) columns. More specifically, for every reuse candidate class the following data have been recorded:

- class name
- pattern name
- number of classes participating in the pattern (NOFparticipants)
- number of packages where the pattern is spread into (NOFpackageSet)
- reusability of *Class Based Selection* Alternative A_1 (R-class)
- reusability of *Pattern Based Selection* Alternative A_2 (R-pattern)
- reusability of *Package Based Selection* Alternative A_3 (R-package)
- reusability of *Multi-Package Based Selection* Alternative A_4 (R-packageSet)

In the data analysis phase we have used several statistical tests, descriptive statistics and graphs. More specifically, concerning RQ_1, we performed hypothesis testing in order to investigate $H_{0(1)}$–$H_{0(4)}$. In order to explore RQ_2–RQ_4, we produced additional hypothesis testing. In order for this goal to be feasible, we created two categorical variables. The variable transformation rules have been selected according to the histograms of Figs. 1 and 2 and the quartiles presented in Table 3.

4. Statistical analysis

One of the first steps in analyzing the dataset in a statistical analysis requires the data reduction phase (Wohlin et al., 2000). More specifically, one of our concerns is to eliminate all outliers that derive from extreme reusability index values, both high and low. In order to inspect the existence of outliers we have created boxplots for the R-class, R-pattern, R-package and R-packageSet variables. The boxplots are presented in Fig. 3. As it is shown in Fig. 3, the dataset has several outliers that have been omitted from further statistical analysis. On the completion of this process, the dataset consisted of 23,931 rows. The final dataset of the study is available in the web.[8]

The descriptive statistics on the reusability of each class selection alternative are presented in Table 4. In order to draw safer conclusions on the differences presented in the mean reusability values, hypotheses testing have been considered. According to

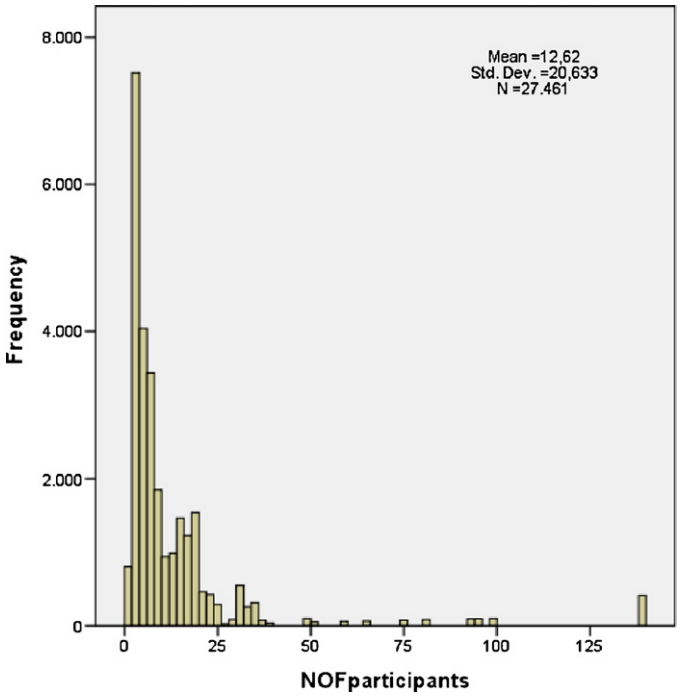

Fig. 1. Distribution of NOFparticipants variable.

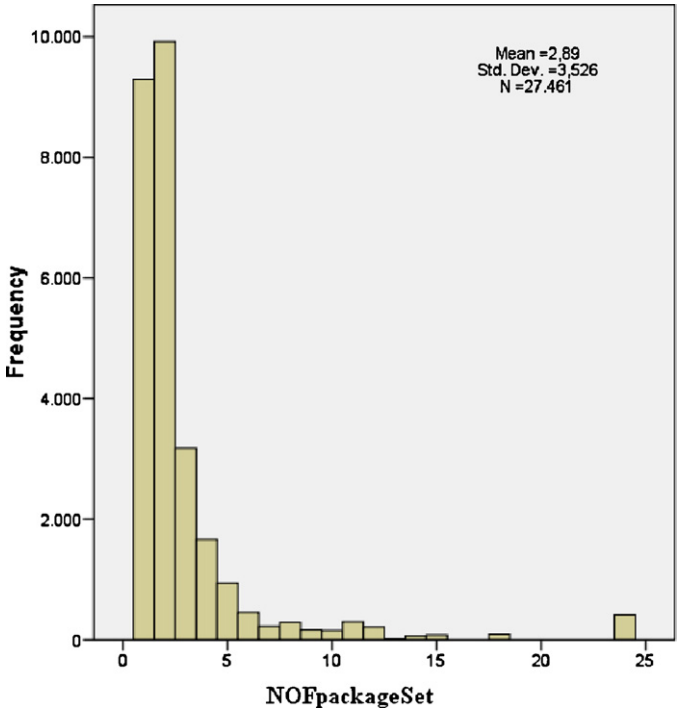

Fig. 2. Distribution of NOFpackageSet variable.

[8] http://sweng.csd.auth.gr/apamp/material/metrics_jss.xls or http://sweng.csd.auth.gr/apamp/material/metrics_jss.csv.

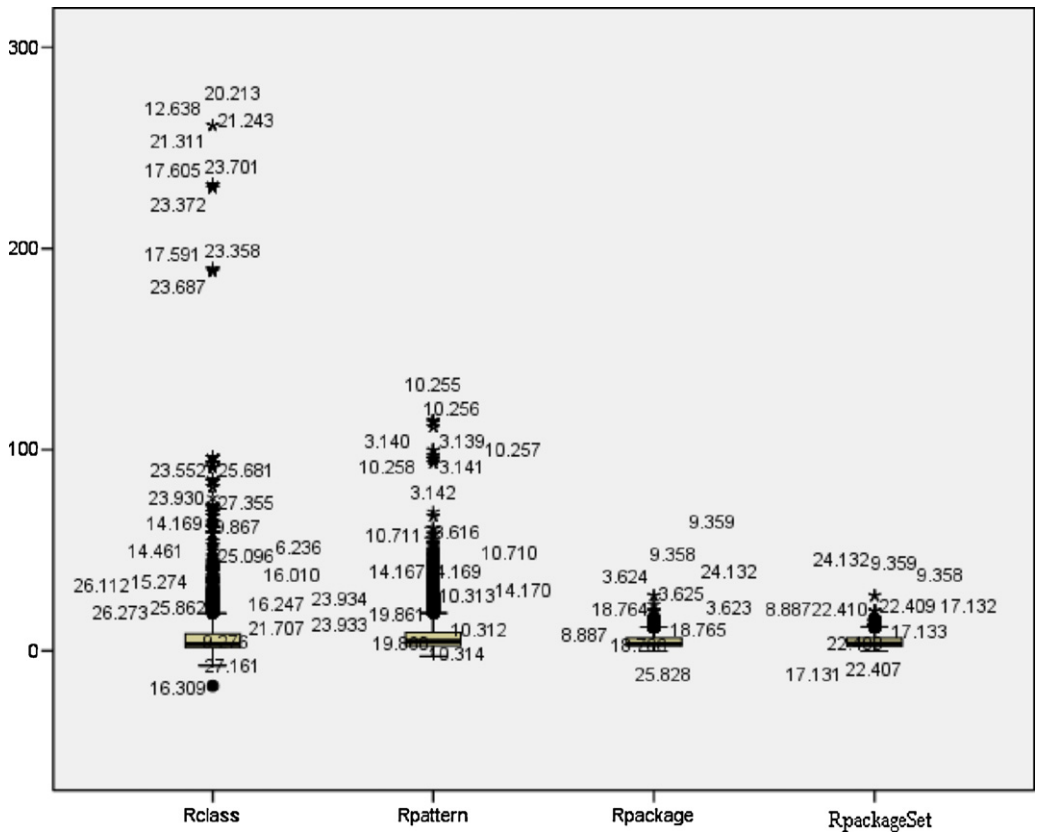

Fig. 3. Boxplots on R-class, R-pattern, R-package and R-packageSet variables.

Table 3
Variable transformation range.

Variable	Range	Categorical value
NOFparticipants	0–3	Low
NOFparticipants	4–6	Medium
NOFparticipants	7–15	High
NOFparticipants	>16	Very high
NOFpackageSet	1	Low
NOFpackageSet	2	Medium
NOFpackageSet	>3	High

Table 4
Descriptive statistics on class selection alternatives.

	N	Min	Max	Mean	Std. Deviation
R-pattern	23,931	−2.58	41.26	5.21	3632
R-class	23,931	−7.25	18.77	4.55	4237
R-package	23,931	−0.19	12.53	4.05	2203
R-packageSet	23,931	−0.19	11.53	3.99	2135

Wohlin et al. (2000), there are two ways for selecting the best performing method, among methods that have been tested on the same sample, i.e. paired sample *t*-test and Wilcoxon Signed Rank test. In the case of our dataset, since data does not follow the normal distribution, we had to employ a non-parametric hypothesis testing technique, i.e. Wilcoxon Signed Rank test. The results on the comparison of the four class selection alternatives are presented in Tables 5 and 6.

The results of Table 5, suggest that the reusability of the pattern alternative is higher than the reusability of the class alternative in 59.1% of the cases, the reusability of class is higher than the pattern's in 37.6% and the two alternatives tie in the rest 3.3% of the cases. If a developer chooses to select a design pattern as a starting point for white-box reuse, he gains a statistically significant change in the selected granule reusability rather than if he selects to start from a single class ($Z = 34.290$, sig = 0.000).

In order to investigate RQ_2–RQ_4, we performed the Crosstabs procedure with the categorical variables described in Section 3.3, i.e. variables on the number of pattern participating classes, number of packages that participate in the pattern, pattern name and best class selection alternative. The crosstabs procedure is often used to record and analyze the relation between two or more categorical variables. It displays the (multivariate) frequency distribution of the variables in a matrix format. For simplicity in our tables we present the frequency as a percentage and not as an absolute value.

The results of the Crosstabs are presented in Tables 6–8. The significance level of all Pearson x^2-test that derived from the Crosstabs equals sig. = 0.00 and therefore pattern type, number of participants, number of packages are correlated to the selection of the best practice.

Finally, in order to further explore the way that the variables of RQ_2–RQ_4 influence the selection of the best class selection practice we performed Wilcoxon Singed Rank tests in several sub-datasets. For example, in order to investigate if the difference between R-package and R-pattern for the Composite pattern is statistically significant, we filtered the dataset, so as to isolate only the rows that correspond to instances of the Composite pattern and replicated the procedure of Tables 4 and 5. The results of this procedure are presented in Appendix A.

5. Discussion

In this section of the paper we discuss the findings of our study with respect to the four research questions that have been stated in the case study plan. The motivation of our research dealt with the selection of the set of classes that are going to be reused. More specifically, we investigated four class selection alternatives as a

Table 5
Wilcoxon Signed Rank tests on class selection alternatives.

	N	Mean rank	Sum of ranks	*Z*	sig
(R-pattern)–(R-class)					
Negative ranks	8939	11082.10303	99,062,919	34.290	0.000
Positive ranks	14,205	11881.09968	168,771,021		
Ties	787				
Total	23,931				
(R-package)–(R-class)					
Negative ranks	10,312	14071.60745	145,106,416	−1.871	0.061
Positive ranks	13,613	10365.70624	141,108,359		
Ties	6				
Total	23,931				
(R-packageSet)–(R-class)					
Negative ranks	10,248	13960.0523	143,062,616	0.042	0.967
Positive ranks	13,677	10466.63442	143,152,159		
Ties	6				
Total	23,931				
(R-package)–(R-pattern)					
Negative ranks	15,236	13728.85482	209,172,832	−62.233	0.000
Positive ranks	8663	8821.449613	76,420,218		
Ties	32				
Total	23,931				
(R-packageSet)–(R-pattern)					
Negative ranks	14545	13985.25775	203,415,574	−57.027	0.000
Positive ranks	9341	8764.25083	81,866,867		
Ties	45				
Total	23,931				
(R-packageSet)–(R-package)					
Negative ranks	8597	8888.882866	76,417,726	15.233	0.000
Positive ranks	10,130	9767.216387	98,941,902		
Ties	5204				
Total	23,931				

Table 6
Crosstabs (best approach—design pattern).

	Alternative A_1	Alternative A_2	Alternative A_3	Alternative A_4
(Object) Adapter-Command	38.85%	18.55%	38.48%	4.12%
Composite	20.49%	41.95%	33.17%	4.39%
Decorator	25.03%	25.35%	28.00%	21.62%
Factory Method	31.87%	18.50%	32.51%	17.12%
Observer	26.41%	6.60%	64.06%	2.93%
Prototype	26.18%	9.31%	43.76%	20.74%
Proxy	44.31%	3.92%	41.57%	10.20%
Proxy2	38.89%	0.00%	27.78%	33.33%
Singleton	36.21%	63.79%	0.00%	0.00%
State-Strategy	31.13%	15.35%	39.21%	14.31%
Template Method	22.16%	25.43%	20.79%	31.61%
Visitor	25.00%	75.00%	0.00%	0.00%

Table 7
Crosstabs (best approach—NOFParticipants).

	Alternative A_1	Alternative A_2	Alternative A_3	Alternative A_4
Low	36.43%	24.53%	32.25%	6.79%
Medium	27.89%	17.57%	37.98%	16.56%
High	27.53%	14.14%	39.65%	18.69%
Very high	27.05%	13.43%	36.20%	23.31%

starting point for white-box reuse, namely *(Alternative A_1) "select only the class that implements the desired functionality", (Alternative A_2) "select all the classes that participate in the pattern that the class which implements the desired functionality participates in", (Alternative A_3) "select all the classes that participate in the package that the class which implements the desired functionality belongs to" and (Alternative A_4) "select all classes that belong to all packages that are involved in the pattern to which the class that implements the desired functionality participates in".*

Table 8
Crosstabs (Best Approach – NOFPackageSet).

	Alternative A_1	Alternative A_2	Alternative A_3	Alternative A_4
Low	29.87%	24.76%	29.63%	15.75%
Medium	31.10%	15.75%	38.67%	14.47%
High	29.61%	12.35%	41.30%	16.74%

5.1. RQ_1—which is the most reusable unit?

Taking under consideration all cases, i.e. not filtering pattern instances, design pattern size, i.e. number of classes that participate in the design pattern, and number of packages involved in the pattern, the results suggest that employing the pattern-based selection approach (*Alternative A_2*), provides statistically significant more reusable groups of classes than the other alternatives. More specifically, the pattern provides the best approach in about 40% of the cases.

5.2. RQ_2—does design pattern type affect reusability?

In order to investigate the remaining cases and identify rules that help the developer decide the best selection approach, we investigated the correlation among the best selection practise, design pattern size, pattern type and the number of packages that the pattern is spread into. According to our analysis, all the

aforementioned variables appeared to be correlated. By taking into account the results of Table 5 and Appendix A, we were able to rank the best selection alternatives according to design pattern type. The results are summarized in Table 9. The columns of the table represent design patterns. For every pattern the alternatives are ranked from higher to lower, with respect to their reusability. More specifically, we ranked the alternatives to levels from 1 to 4. In order to do so we performed the following steps:

1. we ranked the alternatives in a descending order according to their average reusability
2. we demarcated levels in cases when the mean reusability of one alternative is statistically significantlly higher than the reusability of the other alternative.

For example, concerning the Adapter pattern, the alternative with the higher reusability metric score is A2, followed by A1. From the results presented in Appendix A, we observe that the difference between class selection and pattern selection is statistically significant. Therefore, A2 and A1 belong to different reusability levels.

Table 9 suggests that a developer should select the design pattern, rather than any other solution, if the class he is interested in, participates in an Adapter, Factory Method, Prototype or State design pattern. On the other hand, if the class participates in a Composite or Decorator pattern, it is suggested that the developers should reuse the whole package where the class belongs to. The majority of these results comply with the opinion of professional software engineers as stated in Khomh and Gueheneuc (2008). On the contrary, our results differ in three points, namely (a) Composite pattern reusability, (b) State pattern reusability and (c) Template Method pattern reusability. Although a direct comparison of our results to Khomh et al. is not possible, because of the different nature of the two studies, we provide a discussion on the differences of the results.

In Khomh and Gueheneuc (2008) the authors suggest that the use of *Composite* pattern improves the reusability of a system, in contrast to the use of *Decorator* pattern, which diminishes the reusability of the system. However, the two patterns are very similar in their structure and the expected result would be that these patterns should exhibit similar quality characteristics. From our study both patterns appear to have similar reusability effects, which was the expected conclusion. Additionally, concerning the *State*/*Strategy* pattern, our approach suggested that the pattern-based alternative appears to be the optimal class selection scenario, which is in contrast to the results of Khomh and Gueheneuc (2008), where the authors suggested that both State and Strategy employment have a negative effect on software reusability. However, these patterns are very easy to use, since they only use a polymorphism mechanism, which enhances extendibility and understandability. Finally, concerning Template Method, which according to Khomh and Gueheneuc (2008) has a positive effect on reusability, the result of our empirical study suggests that the template pattern is not the best way to select a reusable group of classes. If a developer needs to reuse a class that participates in an instance of a Template, he should reuse all packages that are involved in the pattern instance. A closer analysis of the results of our study indicates that in most cases the Template pattern instances, that have been identified, are spread into different packages, and that the classes that participate in it present high coupling, to other classes. The classes that comprise the Template Method pattern create the skeleton of an algorithm. Thus, in order for the algorithm to access all data that it is interested in, these classes have to be highly coupled with other classes. This fact suggests that Template might not be properly used by open-source developers, since proper application of the pattern suggests that the algorithm uses local class data. As a conclusion, the results on the Template Method pattern should be cautiously adopted. In addition, Template pattern reusability probably needs further investigation.

5.3. RQ_3 and RQ_4—does number of pattern participants or the number of packages that are involved in the pattern affect reusability?

Similarly, in order to present the correlations among the best class selection alternative, the design pattern's size and the number of packages involved in the pattern, we used the results of Tables 7–8 and Appendix A. Thus, we were able to rank the best selection alternatives according to design pattern size and distribution of pattern among packages. The results are summarized in Table 10. The columns of the table represent NOFparticipants and NOFpackageSet categories. For every category the alternatives are ranked from higher to lower, with respect to their reusability. The results of this procedure were not very helpful in identifying rules for selecting the optimal class selection scenario, since they are similar for all variable values.

5.4. Illustrative example

As an example, in Fig. 4 we present a part of the *org.jfree.chart.block* package of the *jFreeChart* project. This package employs five *State/Strategy* instances. From these patterns, two involve only classes from the aforementioned package, whereas the rest three pattern instances are spread into two packages.

According to our work, if a class is identified to participate in a State/Strategy pattern, the reuser should investigate the number of packages that are involved in the pattern. If the pattern is completely instantiated in the same package, then the pattern is the most reusable unit, but this result is marginal (35.9% patterns vs. 35.5% package). However, if the state pattern is spread in more than one package, the pattern is the more reusable unit. The reusability of each class selection alternative, for each pattern is presented in Table 11.

The results that are suggested from Table 11 and from the empirical data of our dataset are intuitively valid. If a developer wants to reuse the functionality of the *Arrangement* hierarchy he will probably need to use all the classes of *Block*, *AbstractBlock* and *BlockFrame* hierarchies, which is almost the whole package. On the contrary, if the reuser is interested in the functionality of *BlockFrame* hierarchy, or the *EntityBlockParams* hierarchy, only a small portion of the package is needed, because these classes are almost self-sufficient. Similarly, if a reuser is interested in the functionality of the *Block* hierarchy, *AbstractBlock* and *BlockFrame* hierarchies are needed but *Arrangement* hierarchy is optional. In the case that Arrangement is needed the optimal selection strategy should be the package set.

The above scenario and discussion, comply with the empirical data of our research. Thus, a reuser could select the optimal reusability unit according to the results of our study without performing extensive dependency analysis. This fact suggests that an inexperienced developer/re-user, who could be mislead by his intuition, can be guided from the results of this study and make the optimal decision on class selection strategies.

5.5. Practical consideration

This section deals with the practical benefits that derive from this work. Firstly, we present an approach of how practitioners can use our results. The approach is summarized in the next steps:

- Let us assume that a reuser identifies a piece of code that provides some kind of desired functionality.
- The reuser examines if the class/classes that he is interested in, are involved in a design pattern.

Table 9
Best class selection alternative rankings.

Componentization alternatives ranking												
Level-1	A2	A3	A3	A2	A2	A2	A2 A1	A4 A1 A2 A3	A3 A4	A2	A4	A4 A1 A2 A3
Level-2	A1	A4	A4	A3	A3	A1 A3 A4	A3 A4		A1 A2	A4	A3	
Level-3	A3	A2	A2	A4	A1 A4					A1 A3	A2	
Level-4	A4	A1	A1	A1							A1	
	Adapter	Composite	Decorator	Factory	Observer	Prototype	Proxy	Proxy2	Singleton	State	Template	Visitor

Table 10
Best class selection alternative ranking.

Componentization alternatives ranking							
Level-1	A2	A2	A2	A2	A2	A2	A2
Level-2	A3 A4	A4	A3 A4	A4	A3 A4	A3 A4 A1	A4
Level-3	A1	A3	A1	A3	A1		A3
Level-4		A1		A1			A1
	0–3 classes	4–6 classes	7–15 classes	>15 classes	1 package	2 packages	>2 packages

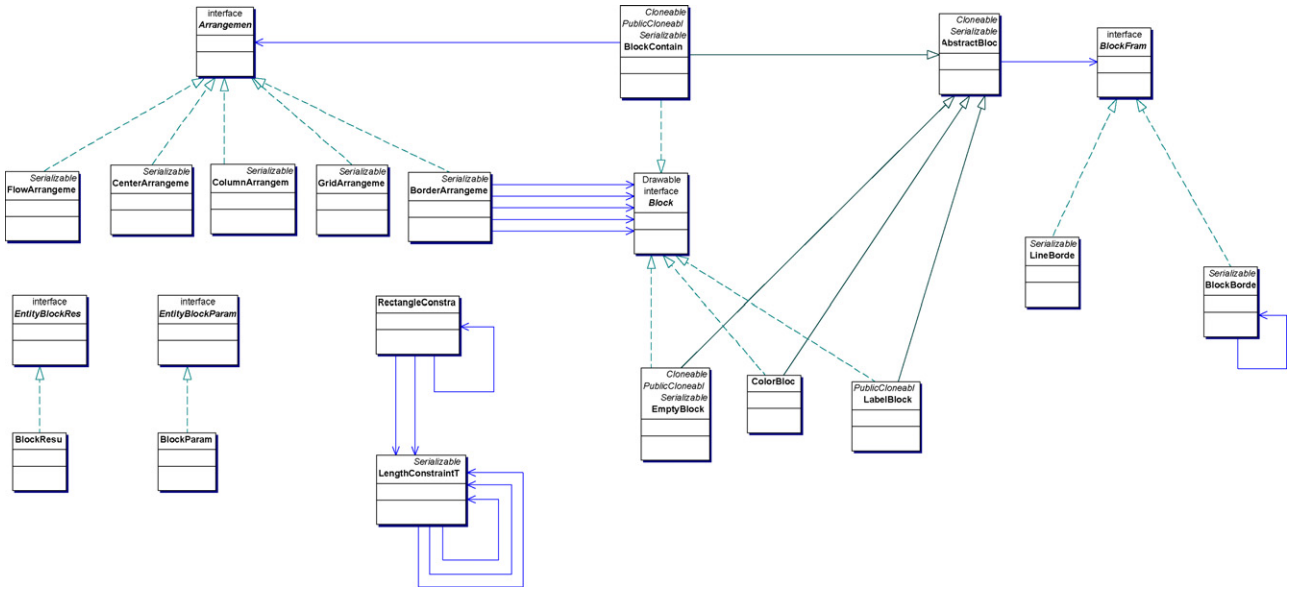

Fig. 4. Class diagram from jFreeChart.org.jfree.chart.block package.

- If they are not, according to our results, the reuser should pick the package that the class/classes belong to and reuse it/them.
- If the class/classes participate in the pattern:

- The reuser identifies, pattern type, pattern size (number of pattern participants) and the number of packages that the pattern is spread into.
- The reuser searches the complete results of our study, and identifies the most profitable reuse granule (class/pattern/package/package set).

The above steps can be implemented in software engineering recommendation tool (Robillard et al., 2010) that would (a) identify automatically the pattern participation of an interesting class and (b) use the results of this study to recommend the most appropriate reuse granule to the developer relieving him from the manual inspection of the results. The development of such a tool is in progress and its effectiveness will be validated in the authors' future research.

6. Threats to validity

This section of the paper deals with presenting the case study's internal and external threats to validity. To begin with, since the case study subjects are open-source projects, the results may not apply in black-box reuse scenarios where the reuser has no access to the source code. Concerning the empirical study internal validity, the existence of confounding factors is analyzed in Section 3.2. Additionally, the dataset consisted of only Java projects, since the tool we used was able to detect design patterns only in binary java files form. Moreover, only one repository, namely Sourceforge, has been mined. However, the size of the dataset and the statistical significance of the results, suggest that concerning white-box reuse of open-source java code, the results are quite safe.

Furthermore, even though the size of our dataset is sufficient, the results on ten design patterns cannot be generalized to the rest of the 23 design patterns that are described in Gamma et al. (1995). Additionally, neither the results nor the method of the paper can be safely applied in design patterns of other pattern cat-

Table 11
Reusability example (from *jFreeChart.org.jfree.chart.block*).

State/strategy	Context	Subclasses	R-pattern	R-package	R-packageSet	NOFparticipants	NOFpackageSet
Arrangement	BlockContainer	BorderArrangement CenterArrangement ColumnArrangement FlowArrangement GridArrangement	3.4	4.0	4.0	7	1
BlockFrame	AbstractBlock	BlockBorder LineBorder	5.4	4.0	4.0	4	1
BlockFrame	LegendTitle	BlockBorder LineBorder	5.4	4.0	5.3	4	2
Block	BorderArrangement	BlockBorder BlockContainer BlockFrame BlockParams BlockResults ColorBlock EmptyBlock LabelBlock LineBorder LegendGraphic LegendItemBlock Title	5.3	4.0	5.3	14	2
EntityBlockParams	LegendTitle	BlockParams	5.6	4.0	5.3	3	2

egories, such as architectural patterns and game design patterns. The results and conclusions of our research depend strongly on the selection of the QMOOD model (Bansiya and Davis, 2002). It is possible that selecting a different reusability model might provide slightly altered results. The employed reusability model is solidly validated and widely used in the area of software engineering. Thus, the results on the ten design patterns that have been investigated can be considered safe. Additionally, our method cannot be employed in cases when the reuse candidates are not providing similar functionality, because the QMOOD definition of reusability does not take into account functionality which is the key decider in calculating reusability of a component. However, this threat had no negative effect in our study since the alternative reuse granules provide similar functionality.

Moreover, the practical considerations presented in Section 5.5 have not been evaluated with professional or open-source developers. Therefore, the effectiveness and the easiness to use of the proposed methodology need to be evaluated in order to safely assess the applicability of the method.

Finally, the results of this study are interpreted from a design pattern based perspective. One might suggest that cases when a package based selection is preferable than the pattern based selection might occur because of some characteristics of the package. However, the packages that are investigated in our research are so heterogeneous with respect to their size, source project, domain, etc., that their distribution among design patterns does not bias these results.

7. Conclusions—future work

CBSE is a promising technique for enhancing the software development process. In component based (CB) systems the components are explicitly specified, either as component interfaces, or interfaces/abstract classes when using OO languages. In CB systems the easiest way of reuse, is to reuse components, i.e. the classes that implement a particular interface. In open-source software some systems are CB, but not all, and in such cases, both classes that might build a component and components themselves, are being reused. In our study we investigated a scenario where a desired requirement is implemented as a design pattern. The research question that this work explores is which classes should be used as a starting point for white-box reuse, in order for the reusability of the selected classes to be optimized. Thus, we investigated which is the most reusable unit, a class a pattern or a package?

In order to achieve this goal we performed a multi-project case study on about 23,000 classes that could be reused as pattern-based components. For each case, we investigated four alternatives, namely, (a) reuse the class, (b) reuse the pattern that the class belongs to, (c) reuse the package that the class is included in and (d) reuse all packages that include at least one class which participates in the pattern. The results of the study suggest that in most of the cases the alternative to reuse the design pattern offers the optimal selection option. However, there are cases where the package alternative leads to a more reusable set of classes. These scenarios are thoroughly discussed in this paper and are compared to previous work results.

As future work, we plan to replicate our case study on projects written in various object-oriented programming languages and employ different reusability models. Additionally, the aforementioned procedure is intended to investigate all 23 GoF design patterns (Gamma et al., 1995). However, this process will be a difficult task because of the lack of pattern detection tools. Additionally, future research plans include the replication of the study by taking into account domain specific characteristics, which could influence the structural quality of software packages.

Finally, we plan to empirically validate the practical considerations described in Section 5.5, by an experiment. In that study we will ask professional developers to reuse pieces of code that are based on design patterns and pieces of code that are not. This way we will evaluate the correctness and the needed time for component adaptation, in both cases, and compare the results. Additionally, we will ask the developers to evaluate the usefulness of the results of this study by assessing a recommendation tool that would assist them selecting the best reuse granule based on quality characteristics.

Acknowledgements

This work is partially funded by the European Commission in the context of the OPEN-SME 'Open-Source Software Reuse Services for SMEs' project, under the grant agreement no. FP7-SME-2008-2/243768. The authors would like to acknowledge many valuable suggestions made by the anonymous reviewers with regard to the discussion on game requirements, game project management and game maintenance.

Appendix A.

Wilcoxon Signed Rank Test results.

Variable	Value	Solution comparison		*N*	Sum of ranks	*Z*	Sig.
Pattern	(Object) Adapter-Command	(R-pattern)–(R-class)	Negative Ranks	1502	2265208.50	4.694	0.000
			Positive Ranks	1664	2748152.50		
			Ties	10			
			Total	3176			
		(R-package)–(R-class)	Negative Ranks	1670	3184930.00	−12.817	0.000
			Positive Ranks	1506	1860146.00		
			Ties	0			
			Total	3176			
		(R-packageSet)–(R-class)	Negative Ranks	1670	3184930.00	−12.817	0.000
			Positive Ranks	1506	1860146.00		
			Ties	0			
			Total	3176			
		(R-package)–(R-pattern)	Negative Ranks	2285	4117865.00	−30.869	0.000
			Positive Ranks	891	927211.00		
			Ties	0			
			Total	3176			
		(R-packageSet)–(R-pattern)	Negative Ranks	2285	4117865.00	−30.869	0.000
			Positive Ranks	891	927211.00		
			Ties	0			
			Total	3176			
		(R-packageSet)–(R-package)	Negative Ranks	632	359277.00	−5.426	0.000
			Positive Ranks	467	245173.00		
			Ties	2077			
			Total	3176			
Pattern	Composite	(R-pattern)–(R-class)	Negative Ranks	49	6220.00	5.111	0.000
			Positive Ranks	156	14895.00		
			Ties	0			
			Total	205			
		(R-package)–(R-class)	Negative Ranks	64	9408.00	1.354	0.176
			Positive Ranks	141	11707.00		
			Ties	0			
			Total	205			
		(R-packageSet)–(R-class)	Negative Ranks	66	9445.00	1.311	0.190
			Positive Ranks	139	11670.00		
			Ties	0			
			Total	205			
		(R-package)–(R-pattern)	Negative Ranks	104	14852.00	−5.091	0.000
			Positive Ranks	101	6263.00		
			Ties	0			
			Total	205			
		(R-packageSet)–(R-pattern)	Negative Ranks	104	12656.00	−2.488	0.013
			Positive Ranks	101	8459.00		
			Ties	0			
			Total	205			
		(R-packageSet)–(R-package)	Negative Ranks	137	13863.00	−3.919	0.000
			Positive Ranks	68	7252.00		
			Ties	0			
			Total	205			
		(R-pattern)–(R-class)	Negative Ranks	686	508931.50	6.371	0.000
			Positive Ranks	894	740058.50		
			Ties	2			
			Total	1582			
		(R-package)–(R-class)	Negative Ranks	530	525371.50	5.465	0.000
			Positive Ranks	1050	723618.50		
			Ties	2			
			Total	1582			
		(R-packageSet)–(R-class)	Negative Ranks	538	530771.50	5.167	0.000
			Positive Ranks	1042	718218.50		
			Ties	2			
			Total	1582			
		(R-package)–(R-pattern)	Negative Ranks	753	691612.00	−3.795	0.000
			Positive Ranks	825	554219.00		
			Ties	4			
			Total	1582			
		(R-packageSet)–(R-pattern)	Negative Ranks	752	690357.00	−3.725	0.000
			Positive Ranks	826	555474.00		
			Ties	4			
			Total	1582			
		(R-packageSet)–(R-package)	Negative Ranks	748p	558397.00	−3.940	0.000
			Positive Ranks	663q	437769.00		
			Ties	171r			
			Total	1582			

Appendix A (*Continued*)

Variable	Value	Solution comparison		*N*	Sum of ranks	*Z*	Sig.
Pattern	Factory Method	(R-pattern)–(R-class)	Negative Ranks	450	226613.50	6.056	0.000
			Positive Ranks	623	349587.50		
			Ties	19			
			Total	1092			
		(R-package)–(R-class)	Negative Ranks	481	296662.00	0.166	0.868
			Positive Ranks	611	300116.00		
			Ties	0			
			Total	1092			
		(R-packageSet)–(R-class)	Negative Ranks	494	302476.00	−0.392	0.695
			Positive Ranks	598	294302.00		
			Ties	0			
			Total	1092			
		(R-package)–(R-pattern)	Negative Ranks	620	399931.00	−9.939	0.000
			Positive Ranks	469	193574.00		
			Ties	3			
			Total	1092			
		(R-packageSet)–(R-pattern)	Negative Ranks	617	403775.00	−10.310	0.000
			Positive Ranks	472	189730.00		
			Ties	3			
			Total	1092			
		(R-packageSet)–(R-package)	Negative Ranks	490	242293.50	−6.917	0.000
			Positive Ranks	383	139207.50		
			Ties	219			
			Total	1092			
Pattern	Observer	(R-pattern)–(R-class)	Negative Ranks	110	21995.00	8.330	0.000
			Positive Ranks	299	61850.00		
			Ties	0			
			Total	409			
		(R-package)–(R-class)	Negative Ranks	167	36979.00	2.067	0.039
			Positive Ranks	242	46866.00		
			Ties	0			
			Total	409			
		(R-packageSet)–(R-class)	Negative Ranks	172	39348.00	1.076	0.282
			Positive Ranks	237	44497.00		
			Ties	0			
			Total	409			
		(R-package)–(R-pattern)	Negative Ranks	346	77556.00	−15.474	0.000
			Positive Ranks	59	4659.00		
			Ties	4			
			Total	409			
		(R-packageSet)–(R-pattern)	Negative Ranks	350	79279.00	−16.206	0.000
			Positive Ranks	55	2936.00		
			Ties	4			
			Total	409			
		(R-packageSet)–(R-package)	Negative Ranks	337	70171.00	−12.340	0.000
			Positive Ranks	68	12044.00		
			Ties	4			
			Total	409			
		(R-pattern)–(R-class)	Negative Ranks	1380	2674240.50	22.232	0.000
			Positive Ranks	2827	6177287.50		
			Ties	2			
			Total	4209			
		(R-package)–(R-class)	Negative Ranks	1890	4603737.00	−2.204	0.028
			Positive Ranks	2319	4256208.00		
			Ties	0			
			Total	4209			
		(R-packageSet)–(R-class)	Negative Ranks	1810	4317824.00	1.422	0.155
			Positive Ranks	2399	4542121.00		
			Ties	0			
			Total	4209			
		(R-package)–(R-pattern)	Negative Ranks	2954	7242394.00	−35.672	0.000
			Positive Ranks	1255	1617551.00		
			Ties	0			
			Total	4209			
		(R-packageSet)–(R-pattern)	Negative Ranks	2749	6647665.00	−28.129	0.000
			Positive Ranks	1460	2212280.00		
			Ties	0			
			Total	4209			
		(R-packageSet)–(R-package)	Negative Ranks	1322	1949920.00	25.181	0.000
			Positive Ranks	2512	5401775.00		

Appendix A (*Continued*)

Variable	Value	Solution comparison		*N*	Sum of ranks	*Z*	Sig.
			Ties	375			
			Total	4209			
			Negative Ranks	123	15238.00		
		(R-pattern)–(R-class)	Positive Ranks	128	16388.00	0.499	0.617
			Ties	4			
			Total	255			
			Negative Ranks	191	28217.00		
		(R-package)–(R-class)	Positive Ranks	64	4423.00	−10.091	0.000
			Ties	0			
			Total	255			
			Negative Ranks	191	28217.00		
		(R-packageSet)–(R-class)	Positive Ranks	64	4423.00	−10.091	0.000
			Ties	0			
			Total	255			
Pattern	Proxy		Negative Ranks	207	29736.00		
		(R-package)–(R-pattern)	Positive Ranks	48	2904.00	−11.380	0.000
			Ties	0			
			Total	255			
			Negative Ranks	207	29736.00		
		(R-packageSet)–(R-pattern)	Positive Ranks	48	2904.00	−11.380	0.000
			Ties	0			
			Total	255			
			Negative Ranks	41	3371.00		
		(R-packageSet)–(R-package)	Positive Ranks	84	4504.00	1.405	0.160
			Ties	130			
			Total	255			
			Negative Ranks	30	784.00		
		(R-pattern)–(R-class)	Positive Ranks	24	701.00	−0.357	0.721
			Ties	0			
			Total	54			
			Negative Ranks	25	872.00		
		(R-package)–(R-class)	Positive Ranks	29	613.00	−1.115	0.265
			Ties	0			
			Total	54			
			Negative Ranks	25	881.00		
		(R-packageSet)–(R-class)	Positive Ranks	29	604.00	−1.193	0.233
			Ties	0			
Pattern	Proxy2		Total	54			
			Negative Ranks	33	903.00		
		(R-package)–(R-pattern)	Positive Ranks	21	582.00	−1.384	0.166
			Ties	0			
			Total	54			
			Negative Ranks	33	903.00		
		(R-packageSet)–(R-pattern)	Positive Ranks	21	582.00	−1.384	0.166
			Ties	0			
			Total	54			
			Negative Ranks	9	315.00		
		(R-packageSet)–(R-package)	Positive Ranks	30	465.00	1.053	0.292
			Ties	15			
			Total	54			
			Negative Ranks	0	0.00		
		(R-pattern)–(R-class)	Positive Ranks	0	0.00	0.000	1.000
			Ties	729			
			Total	729			
			Negative Ranks	260	100551.00		
		(R-package)–(R-class)	Positive Ranks	465	162624.00	5.502	0.000
			Ties	4			
			Total	729			
			Negative Ranks	260	100551.00		
		(R-packageSet)–(R-class)	Positive Ranks	465	162624.00	5.502	0.000
			Ties	4			
Pattern	Singleton		Total	729			
			Negative Ranks	260	100551.00		
		(R-package)–(R-pattern)	Positive Ranks	465	162624.00	5.502	0.000
			Ties	4			
			Total	729			
			Negative Ranks	260	100551.00		
		(R-packageSet)–(R-pattern)	Positive Ranks	465	162624.00	5.502	0.000
			Ties	4			
			Total	729			
			Negative Ranks	0	0.00		
		(R-packageSet)–(R-package)	Positive Ranks	0	0.00	0.000	1.000
			Ties	729			
			Total	729			

Appendix A (*Continued*)

Variable	Value	Solution comparison		*N*	Sum of ranks	*Z*	Sig.
Pattern	State—Strategy	(R-pattern)–(R-class)	Negative Ranks	3813	18400427.00	22.185	0.000
			Positive Ranks	6138	31115749.00		
			Ties	0			
			Total	9951			
		(R-package)–(R-class)	Negative Ranks	4361	25106938.00	−1.217	0.223
			Positive Ranks	5590	24409238.00		
			Ties	0			
			Total	9951			
		(R-packageSet)–(R-class)	Negative Ranks	4368	25068241.00	−1.082	0.279
			Positive Ranks	5583	24447935.00		
			Ties	0			
			Total	9951			
		(R-package)–(R-pattern)	Negative Ranks	6653	37510401.00	−44.999	0.000
			Positive Ranks	3298	12005775.00		
			Ties	0			
			Total	9951			
		(R-packageSet)–(R-pattern)	Negative Ranks	6475	37133474.00	−43.183	0.000
			Positive Ranks	3476	12382702.00		
			Ties	0			
			Total	9951			
		(R-packageSet)–(R-package)	Negative Ranks	4105	18506109.00	4.161	0.000
			Positive Ranks	4727	20500419.00		
			Ties	1119			
			Total	9951			
	Template Method	(R-pattern)–(R-class)	Negative Ranks	795	890807.00	12.009	0.000
			Positive Ranks	1449	1628083.00		
			Ties	21			
			Total	2265			
		(R-package)–(R-class)	Negative Ranks	672	900456.50	12.293	0.000
			Positive Ranks	1593	1665788.50		
			Ties	0			
			Total	2265			
		(R-packageSet)–(R-class)	Negative Ranks	653	848396.50	13.966	0.000
			Positive Ranks	1612	1717848.50		
			Ties	0			
			Total	2265			
		(R-package)–(R-pattern)	Negative Ranks	1021	1149341.00	3.723	0.000
			Positive Ranks	1227	1378535.00		
			Ties	17			
			Total	2265			
		(R-packageSet)–(R-pattern)	Negative Ranks	713	977395.00	8.914	0.000
			Positive Ranks	1522	1521335.00		
			Ties	30			
			Total	2265			
		(R-packageSet)–(R-package)	Negative Ranks	772	575696.50	13.685	0.000
			Positive Ranks	1128	1230253.50		
			Ties	365			
			Total	2265			
	Visitor	(R-pattern)–(R-class)	Negative Ranks	1	4.00	0.365	0.715
			Positive Ranks	3	6.00		
			Ties	0			
			Total	4			
		(R-package)–(R-class)	Negative Ranks	1	4.00	0.365	0.715
			Positive Ranks	3	6.00		
			Ties	0			
			Total	4			
		(R-packageSet)–(R-class)	Negative Ranks	1	4.00	0.365	0.715
			Positive Ranks	3	6.00		
			Ties	0			
			Total	4			
		(R-package)–(R-pattern)	Negative Ranks	0	0.00	2.000	0.046
			Positive Ranks	4	10.00		
			Ties	0			
			Total	4			
		(R-packageSet)–(R-pattern)	Negative Ranks	0	0.00	2.000	0.046
			Positive Ranks	4	10.00		
			Ties	0			
			Total	4			
		(R-packageSet)–(R-package)	Negative Ranks	4	10.00	2.000	0.046
			Positive Ranks	0	0.00		
			Ties	0			
			Total	4			

Appendix A (*Continued*)

Variable	Value	Solution comparison		*N*	Sum of ranks	*Z*	Sig.
Number of classes participating in the pattern	0–3 classes	(R-pattern)–(R-class)	Negative Ranks	2983	9164047.50	8.944	0.000
			Positive Ranks	3501	11860322.50		
			Ties	781			
			Total	7265			
		(R-package)–(R-class)	Negative Ranks	3545	15157998.50	−11.060	0.000
			Positive Ranks	3716	11206692.50		
			Ties	4			
			Total	7265			
		(R-packageSet)–(R-class)	Negative Ranks	3543	15124213.50	−10.871	0.000
			Positive Ranks	3718	11240477.50		
			Ties	4			
			Total	7265			
		(R-package)–(R-pattern)	Negative Ranks	4589	19213349.00	−33.982	0.000
			Positive Ranks	2664	7093282.00		
			Ties	12			
			Total	7265			
		(R-packageSet)–(R-pattern)	Negative Ranks	4592	19161279.00	−33.690	0.000
			Positive Ranks	2661	7145352.00		
			Ties	12			
			Total	7265			
		(R-packageSet)–(R-package)	Negative Ranks	1559	2253036.50	1.140	0.254
			Positive Ranks	1479	2363204.50		
			Ties	4227			
			Total	7265			
Number of classes participating in the pattern	4–6 classes	(R-pattern)–(R-class)	Negative Ranks	1924	4566435.50	19.153	0.000
			Positive Ranks	3215	8640794.50		
			Ties	6			
			Total	5145			
		(R-package)–(R-class)	Negative Ranks	2145	6341547.50	2.558	0.011
			Positive Ranks	2998	6886248.50		
			Ties	2			
			Total	5145			
		(R-packageSet)–(R-class)	Negative Ranks	2124	6213732.50	3.758	0.000
			Positive Ranks	3019	7014063.50		
			Ties	2			
			Total	5145			
		(R-package)–(R-pattern)	Negative Ranks	3266	9492957.00	−27.358	0.000
			Positive Ranks	1867	3683454.00		
			Ties	12			
			Total	5145			
		(R-packageSet)–(R-pattern)	Negative Ranks	3074	9175048.00	−24.552	0.000
			Positive Ranks	2053	3970580.00		
			Ties	18			
			Total	5145			
		(R-packageSet)–(R-package)	Negative Ranks	2238	4881580.00	5.002	0.000
			Positive Ranks	2381	5788310.00		
			Ties	526			
			Total	5145			
Number of classes participating in the pattern	7–15 classes	(R-pattern)–(R-class)	Negative Ranks	2339	6942098.00	21.177	0.000
			Positive Ranks	3992	13101848.00		
			Ties	0			
			Total	6331			
		(R-package)–(R-class)	Negative Ranks	2620	9456430.00	3.889	0.000
			Positive Ranks	3711	10587516.00		
			Ties	0			
			Total	6331			
		(R-packageSet)–(R-class)	Negative Ranks	2577	9319996.00	4.827	0.000
			Positive Ranks	3754	10723950.00		
			Ties	0			
			Total	6331			
		(R-package)–(R-pattern)	Negative Ranks	4113	14302829.00	−29.665	0.000
			Positive Ranks	2210	5690497.00		
			Ties	8			
			Total	6331			
		(R-packageSet)–(R-pattern)	Negative Ranks	3909	13835576.00	−26.643	0.000
			Positive Ranks	2407	6113510.00		
			Ties	15			
			Total	6331			
		(R-packageSet)–(R-package)	Negative Ranks	2990	8723665.50	0.903	0.367
			Positive Ranks	2957	8962712.50		
			Ties	384			
			Total	6331			

Appendix A (*Continued*)

Variable	Value	Solution comparison		*N*	Sum of ranks	*Z*	Sig.
Number of classes participating in the pattern	>15 classes	(R-pattern)–(R-class)	Negative Ranks	1693	4550844.00	20.236	0.000
			Positive Ranks	3497	8919801.00		
			Ties	0			
			Total	5190			
		(R-package)–(R-class)	Negative Ranks	2002	6435015.00	2.782	0.005
			Positive Ranks	3188	7035630.00		
			Ties	0			
			Total	5190			
		(R-packageSet)–(R-class)	Negative Ranks	2004	6286977.00	4.153	0.000
			Positive Ranks	3186	7183668.00		
			Ties	0			
			Total	5190			
		(R-package)–(R-pattern)	Negative Ranks	3268	10445233.00	−34.368	0.000
			Positive Ranks	1922	3025412.00		
			Ties	0			
			Total	5190			
		(R-packageSet)–(R-pattern)	Negative Ranks	2970	9818659.00	−28.563	0.000
			Positive Ranks	2220	3651986.00		
			Ties	0			
			Total	5190			
		(R-packageSet)–(R-package)	Negative Ranks	1810	4388884.00	20.533	0.000
			Positive Ranks	3313	8736242.00		
			Ties	67			
			Total	5190			
Number of packages participating in the pattern	1 package	(R-pattern)–(R-class)	Negative Ranks	3019	11467741.00	16.400	0.000
			Positive Ranks	4632	17804985.00		
			Ties	783			
			Total	8434			
		(R-package)–(R-class)	Negative Ranks	3366	16186830.00	7.042	0.000
			Positive Ranks	5062	19332976.00		
			Ties	6			
			Total	8434			
		(R-packageSet)–(R-class)	Negative Ranks	3366	16186780.00	7.043	0.000
			Positive Ranks	5062	19333026.00		
			Ties	6			
			Total	8434			
		(R-package)–(R-pattern)	Negative Ranks	4364	21389478.00	−16.816	0.000
			Positive Ranks	4038	13911525.00		
			Ties	32			
			Total	8434			
		(R-packageSet)–(R-pattern)	Negative Ranks	4351	21332824.00	16.846	0.000
			Positive Ranks	4038	13859031.00		
			Ties	45			
			Total	8434			
		(R-packageSet)–(R-package)	Negative Ranks	3058	8928257.50	0.590	0.555
			Positive Ranks	2944	9086745.50		
			Ties	2432			
			Total	8434			
Number of packages participating in the pattern	2 packages	(R-pattern)–(R-class)	Negative Ranks	3189	11766131.50	22.160	0.000
			Positive Ranks	4919	21107754.50		
			Ties	4			
			Total	8112			
		(R-package)–(R-class)	Negative Ranks	3705	17275300.00	−3.898	0.000
			Positive Ranks	4407	15631028.00		
			Ties	0			
			Total	8112			
		(R-packageSet)–(R-class)	Negative Ranks	3633	16654314.50	−0.954	0.340
			Positive Ranks	4479	16252013.50		
			Ties	0			
			Total	8112			
		(R-package)–(R-pattern)	Negative Ranks	5676	25613506.00	−43.428	0.000
			Positive Ranks	2436	7292822.00		
			Ties	0			
			Total	8112			
		(R-packageSet)–(R-pattern)	Negative Ranks	5318	24360603.00	−37.488	0.000
			Positive Ranks	2794	8545725.00		
			Ties	0			
			Total	8112			
		(R-packageSet)–(R-package)	Negative Ranks	2541	6850500.00	12.805	0.000
			Positive Ranks	3288	10141035.00		
			Ties	2283			
			Total	8112			

Appendix A (*Continued*)

Variable	Value	Solution comparison		*N*	Sum of ranks	*Z*	Sig.
Number of packages participating in the pattern	>2 package	(R-pattern)–(R-class)	Negative Ranks	2731	9899817.50	20.394	0.000
			Positive Ranks	4654	17372987.50		
			Ties	0			
			Total	7385			
		(R-package)–(R-class)	Negative Ranks	3241	14835608.00	−6.545	0.000
			Positive Ranks	4144	12437197.00		
			Ties	0			
			Total	7385			
		(R-packageSet)–(R-class)	Negative Ranks	3249	14748619.00	−6.070	0.000
			Positive Ranks	4136	12524186.00		
			Ties	0			
			Total	7385			
		(R-package)–(R-pattern)	Negative Ranks	5196	22104667.00	−46.219	0.000
			Positive Ranks	2189	5168138.00		
			Ties	0			
			Total	7385			
		(R-packageSet)–(R-pattern)	Negative Ranks	4876	21561870.00	−43.256	0.000
			Positive Ranks	2509	5710935.00		
			Ties	0			
			Total	7385			
		(R-packageSet)–(R-package)	Negative Ranks	2998	10488056.00	8.482	0.000
			Positive Ranks	3898	13292800.00		
			Ties	489			
			Total	7385			

References

Ajila, S.A., Wu, D., 2007. Empirical study of the effects of open source adoption on software development economics. Journal of Systems and Software 80 (September (9)), 1517–1529, Elsevier.

Ampatzoglou, A., Chatzigeorgiou, A., 2007. Evaluation of object-oriented design patterns in game development. Information and Software Technology 49 (May (5)), 445–454, Elsevier.

Andreou, A., Tziakouris, M., 2007. A quality framework for developing and evaluating original software components. Information and Software Technology 49 (February (2)), 122–141, Elsevier.

Arnout, K., Meyer, B., 2006. Pattern componentization: the factory example. Innovations in Systems and Software Engineering 2 (2), 65–79, Springer.

Basili, V.R., Selby, R.W., Hutchens, D.H., 1986. Experimentation in software engineering, transactions on software engineering. IEEE Computer Society 12 (7), 733–743, July.

Bansiya, J., Davis, C., 2002. A hierarchical model for object-oriented design quality assessment. Transaction on Software Engineering, IEEE Computer Society 28 (1), 4–17.

Barnard, J., 1998. A new reusability metric for object-oriented software. Software Quality Journal 7 (March (1)), 35–50, Springer.

Bosch, J., 1999. Superimposition: a component adaptation technique. Information & Software Technology 41 (5), 257–273.

Brown, A., Wallnau, K., 1998. The current state of CBSE, software. IEEE Computer Society 15 (September/October (5)), 37–46.

Chang, H.F., Mockus, A., 2008. Evaluation of source code copy detection methods on FreeBSD. In: 2008 International Working Conference on Mining Software Repositories (MSR'08), Association of Computing Machinery , Leipzig, Germany, May 10–11, pp. 61–66.

Cho, E.S., Kim, M.S., Kim, S.D., 2001. Component metrics to measure component quality. In: Proceedings of the 8th Asia-Pasific Software Engineering Conference (APSEC' 01) , IEEE Computer Society, Seoul, South Korea, December 4–7, pp. 419–426.

Clements, P. "Documenting Software Architectures: Views and Beyond". Addison-Wesley Professional, 2nd ed., October 2002.

Conway, M.E., 1968. How do committees invent? Datamation Magazine 14 (April (5)), 28–31.

Counsell, S., Swift, S., Crampton, J., 2006. The interpretation and utility of three cohesion metrics for object-oriented design. Transactions on Software Engineering and Methodology, Association of Computing Machinery 15 (April (2)), 123–149.

Crnkovic, I., Chaudron, M., Larsson, S., 2006. Component-based development process and component lifecycle. In: Proceedings of the 2006 International Conference on Software Engineering Advances (ICSEA' 06) , IEEE Computer Society, Tahiti, French Polynesia, 29 October–03 November 2006, pp. 44–53.

Crnkovic, I., Larsson, M., 2002. Challenges of component-based development. Journal of Systems and Software 61 (April (3)), 201–212, Elsevier.

Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V. "A Classification Framework for Software Component Models," Transactions on Software Engineering, IEEE Computer Society, in press, http://doi.ieeecomputersociety.org/10.1109/TSE.2010.83.

Crnkovic, I., Hnich, B., Johnson, T., Kiziltan, Z., 2002. Specification, implementation, and deployment of components. Communications, Association of Computing Machinery 45 (October (10)), 35–40.

Etzkorn, L.H., Gholston, S.E., Fortune, J.L., Stein, C.E., Utley, D., Farrington, P.A., Cox, G.W., 2004. A comparison of cohesion metrics for object-oriented systems. Information and Software Technology 46 (August (10)), 677–687.

Fahmi, S.A., Choi, H.J., 2008. Life cycles for component based software development. In: Proceedings of the 8th Conference on Computer and Information Technology , IEEE Computer Society, Sydney, Australia, July 8–11, pp. 637–642.

Franch, X., Carvallo, J.P., 2003. Using quality models in software package selection. Software, IEEE Computer Society 20 (January/February (1)), 34–41.

Gamma, E., Helms, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading, MA.

Genero, M., Manso, E., Visaggio, A., Canfora, G., Mario Piattini, 2007. Building measure-based prediction models for UML class diagram maintainability. Empirical Software Engineering 12 (October (5)), 517–549, Springer.

Geuheneuc, Y.G., Sahraoui, H., Zaidi, F., 2004. Fingerprinting Design Patterns. In: Proceedings of the 11th Working Conference on Reverse Engineering , Delft, The Netherlands, November 08–12, pp. 172–181.

Gui, G., Scott, P.D., 2007. Ranking reusability of software components using coupling metrics. Journal of Systems and Software 80 (September (9)), 1450–1459, Elsevier.

Harrison, N.B., Avgeriou, P., 2010. How do architecture patterns and tactics interact? A model and annotation. Journal of Systems and Software 83 (October (10)), 1735–1758, Elsevier.

Hsueh, N.L., Chu, P.H., Chu, W., 2008. A quantitative approach for evaluating the quality of design patterns. Journal of Systems and Software 81 (August (8)), 1430–1439, Elsevier.

Henry, E., Faller, B., 1995. Large-scale industrial reuse to reduce cost and cycle time. Software, IEEE Computer Society 12 (September (5)), 47–53.

Hölzle, U., 1993. Integrating Independently-Developed Components in Object-Oriented Languages. In: Proceedings of the 7th European Conference on Object-Oriented Programming , LNCS 707 Springer, pp. 36–56.

Huston, B., 2001. The effects of design pattern application on metric scores. Journal of Systems and Software 58 (September (3)), 261–269, Elsevier.

Jansen, S., Brinkkemper, S., Hununk, I., Demir, C., 2008. Pragmatic and opportunistic reuse in innovative start-up companies. Software, IEEE Computer Society 25 (November/December (6)), 2–9.

Khomh, F., Gueheneuc, Y.G., 2008. Do design patterns impact software quality positively. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering , IEEE Computer Society, Athens, Greece, April 01–04, pp. 274–278.

Kitchenham, B., Pickard, L., Pfleeger, S.L., 1995. Case studies for method and tool evaluation. Software, IEEE Computer Society 12 (July (4)), 52–62.

Kontio, J., 2006. A case study in applying a systematic method for COTS selection. In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06) , Association of Computing Machinery, Shanghai, China, May 20–28, pp. 201–209.

Kouskouras, K., Chatzigeorgiou, A., Stephanides, G., 2008. Facilitating software extension with design patterns and Aspect-Oriented Programming. Journal of Systems and Software 81 (October (10)), 1725–1737, Elsevier.

Lau, K.K., Wang, Z., 2005. A taxonomy of software component models. In: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA) , IEEE, pp. 88–95.

Li, J., Conradi, R., Slyngstad, O.P.N., Bunse, C., Torchiano, M., Morisio, M., 2009. Development with off-the self components: 10 facts. Software, IEEE Computer Society 26 (March/April (2)), 807–887.
Marcus, A., Poshyvanyk, D., Ferenc, R., 2008. Using the conceptual cohesion of classes for fault prediction in Object-Oriented Systems. Transaction on Software Engineering, IEEE Computer Society 34 (March/April (2)), 287–300.
Meyer, B., Arnout, K., 2006. Componentization: the visitor example. Computer, IEEE Computer Society 39 (July (7)), 23–30.
McConnell, S., 1996. Rapid Development: Taming Wild Software Schedules. Microsoft Press, Redmond, Washington, USA.
McCormack, A.D., Rusnak, J., Baldwin, C.Y., 2008. Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis. In: Working Paper 08-039, Harvard Business School.
Mockus, A., 2007. Large-scale code reuse in open-source software. In: 1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07) , IEEE Computer Society, Minesota, USA, May 20–26, pp. 7–12.
Morad, S., Kuflik, T., 2005. Conventional and open source software reuse at orbotech—an industrial experience. In: International Conference on Software Science, Technology, and Engineering (SwSTE'05) , IEEE Computer Society, Harelip, Israel, February 22–23, pp. 110–117.
Morison, M., Tully, C., Ezra, M., 2000. Diversity in reuse processes. Software, IEEE Computer Society 17 (July/August (4)), 56–63.
Agape, N., Murphy, B., Basili, V., 2008. The influence of organizational structure on software quality: an empirical case study. In: Proceedings of the 30th international conference on Software engineering (ICSE '08) , Association of Computing Machinery, Leipzig, Germany, May 10–18, pp. 521–530.
Plague, H.M., Etzkorn, L.H., Gholston, S., Quattlebaum, S., 2007. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. Transactions on Software Engineering 33 (June (6)), 402–419, IEEE Computer Society.
Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.F., 2002. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Transactions on Software Engineering 28 (June (6)), 595–606.
Robillard, M., Walker, R., Zimmermann, T., 2010. Recommendation systems for software engineering. IEEE Software 27 (July/August (4)), 80–86.
Sandlhu, P.S., Kaur, H., Singh, A., 2009. Modeling reusability of object-oriented software systems. World Academy of Sciences Engineering and Technology 56 (August), 162–165, Academic Science Research.
Scanniello, G., Gravino, C., Risi, M., Tortora, G., 2010. A controlled experiment for assessing the contribution of design pattern documentation on software maintenance. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10), ACM.
Szyperski, C., 1997. Component Software: Beyond Object-Oriented Programming. Addison-Wesley International, Massachusetts, USA.
Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T., 2006. Design pattern detection using similarity scoring. Transaction of Software Engineering 32 (November (11)), 896–909, IEEE Computer Society.
Washizaki, H., Yamamoto, H., Fukazawa, Y., 2003. A metric suite for measuring reusability of software components. In: Proceedings of the 9th International Software Metrics Symposium (METRICS'03) , IEEE Computer Society, Sydney, Australia, September 03–05, pp. 211–223.
Wydaeghe, B., Verschaeve, K., Michiels, B., Damme, B.V., Arckens, E., Jonckers, V., 1998. Building an OMT-editor using design patterns: An experience report. In: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS'98) , IEEE Computer Society, Santa Barbara, California August 3–7, pp. 20–33.
Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A., 2000. Experimentation in Software Engineering, 1st ed. Kluwer Academic Publishers, Boston/Dordrecht/London.
Yu, L., Chen, K., Ramaswamy, S., 2009. Multiple-parameter coupling metrics for layered component based software. Software Quality Journal 17 (March (1)), 5–24, Springer.

Ampatzoglou Apostolos is a PhD candidate in the Department of Informatics, Aristotle University of Thessaloniki, Greece and a laboratory associate at the Technological Education Institute of Thessaloniki, Greece. He holds a BS in Informatics from Technological Education Institute of Thessaloniki and an MSc in Computer Science from the University of Macedonia. His research interests include design patterns, software metrics and computer games.

Kritikos Apostolos is a PhD candidate in the Department of Informatics, Aristotle University of Thessaloniki, Greece and a laboratory associate at the Technological Education Institute of Serres, Greece. He holds a BS in Informatics from Aristotle University of Thessaloniki and an MSc in Information Systems from Aristotle University of Thessaloniki. His research interests include software reuse and software architecture.

Kakarontzas George is a PhD candidate in the Department of Informatics, Aristotle University of Thessaloniki, Greece and a lecturer at the Technological Education Institute of Larissa, Greece. He holds a BS in Informatics from the Athens University of Economics and Business and an MSc in Object-Oriented Software Technology from the University of Brighton. His research interests include component-based software engineering and grid computing.

Dr. Ioannis Stamelos is an Associate Professor at the Department of Informatics of the Aristotle University of Thessaloniki, where he carries out research and teaching in the area of software engineering. He holds a diploma of Electrical Engineering (1983) and a PhD in Computer Science by the Aristotle University of Thessaloniki (1988). His current research interests are focused on open source software engineering, software project management and software education. He has published more than 100 articles in international journals and conferences. He is/was the scientific coordinator or principal investigator for his University in over 20 research and development projects in Information & Communication Technologies with funding from national and international organizations.