Investigating the Use of Object-Oriented Design Patterns in Open-Source Software: A Case Study*

Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos

Aristotle University of Thessaloniki, Thessaloniki, Greece
{apamp,stamelos}@csd.auth.gr

Abstract. During the last decade open source software communities are thriving. Nowadays, several open source projects are so popular that are considered as a standard in their domain. Additionally, the amount of source code that is freely available to developers, offer great reuse opportunities. One of the main concerns of the reuser is the quality of the code that is being reused. Design patterns are well known solutions that are expected to enhance software quality. In this paper we investigate the extent to which object-oriented design patterns are used in open-source software, across domains.

Keywords: Open source software, Design patterns, Empirical study.

1 Introduction

Open source software (OSS), a term introduced in 1998 [9], has been expanding rapidly in recent years. There exist several successful projects developed as open source software, such as Linux, Mozila Firefox and Apache Server.

Collaboration is the basis of the development of an open source project. A first version of the project is developed by a single developer or a group of developers and is released over the internet, freely available, so that the members of the open source community can extend and maintain the project. In open source software development, there are both advantages and disadvantages. One disadvantage of open source software is that there is no documentation and technical support. On the other hand, the advantages of this software development type are its low cost, its reliability and the availability of the source code in order to customize the project according to once special needs [18].

Moreover, another feature of open source software is the potential reuse of the source code, which is freely available to the open source developers. A code segment should have certain characteristics, such as understandability, maintainability and flexibility, in order to be easily and successfully reused in another project.

Gamma et.al have introduced, in 1995, design patterns as common solutions to common design problems [10]. The main incentive to introduce patterns was the creation of a common vocabulary for developers, which provide flexible, reusable and

* This paper is an extended and revised version of the paper entitled "An Empirical Study on Design Pattern Usage on Open-Source Software", published in ENASE 2010.

maintainable design solutions. Furthermore, Meyer et.al explains how object-oriented design patterns can be transformed to reusable components [16].

In literature, many empirical studies have attempted to examine how design pattern application affects software quality. The main conclusion of these studies is that object oriented design patterns can not be considered as universally good or bad. In section 2, we provide a more detailed presentation of the current state of the art, discussing the effect of design pattern use on software quality.

This paper is an extended and revised version on authors' previous work [1] that aims at examining the application of object-oriented design patterns in open source software. More specifically, an empirical study has been performed, in order to investigate which patterns are more frequently used in open source software, which differences exist within software domains and the size of design patterns. The main extension and revision points are concluded below:

- The number of case study subjects is increased
- Added two software categories and revised two others by exploring broader software categories (i.e. replaced e-commerce applications with business applications)
- One more research question dealing with design pattern size has been added.

In the next section of the paper, a literature review on design patterns influence on software quality is provided, In section 3 we present the methodology of our work, i.e. research questions, case study process and data analysis methods. In section 4, the findings of our empirical study are presented, while in section 5 we provide a discussion on the results, categorized according to the research question they address to. Finally, at the end of the paper, possible threats to validity, future work and conclusions are presented.

2 Design Patterns

In this section of the paper we present the findings of a literature review on the influence of design pattern application on software quality. A common division of software quality is between internal and external quality [4]. Software internal quality is measurable and estimates software features such as complexity, cohesion, coupling, inheritance etc. that are not easy to understand for the end-user or the developer. External software quality can not be easily measured, but it is closer to the end-user's and the developer's sense. Functionality, reliability, usability, efficiency, maintainability and portability, are the best known external quality characteristics, as described in ISO/IEC 9126.

The effect of design pattern application on software internal quality has been examined by Ampatzoglou et.al [2] and Huston [12]. According to Huston, the application of the Mediator pattern reduces coupling, the Bridge pattern reduces size and inheritance metrics and finally the use of the Visitor pattern reduces the project's complexity with respect to number of methods [12]. Ampatzoglou et.al suggests that the application of the State and the Bridge pattern reduces coupling and complexity, with respect to cyclomatic complexity and increases cohesion among methods. As a side-effect, the project size concerning the number of classes increases [2].

Furthermore, the effect of design patterns on external quality has been investigated in several studies. The influence of design patterns i.e. Abstract Factory, Observer, Decorator, Composite and Visitor to software maintainability has been investigated by Vokac et.al and Prechelt et.al [17 and 21], by conducting controlled experiments. According to the results of the experiment, the employment of a design pattern is usually more useful than the simpler solution. The software engineer has to choose between applying a design pattern or a simple solution in line with common sense. Besides, Hsueh et.al investigate how design patterns impact on one quality attribute, which is the most obvious attribute that the pattern affects [11]. The selection of the quality attribute is made according to the pattern's non functional requirements, whereas the metric is selected according to [4].

Wendorff presents an industrial case study, where inappropriate pattern use has caused severe maintainability problems. The reasons of inappropriate design pattern use is classified into two categories (1) software engineers have not understood the reasoning behind the patterns that they have employed (2) the patterns that they have applied have not fulfilled the project's requirements. Moreover, the paper emphasizes the need for documenting design pattern application and that pattern removal leads to extreme cost [22]. In [13], an analysis on software maintenance, with professional engineers, is performed. According to the empirical study, design patterns do not always have positive impact on software quality. In particular, it is concluded that when patterns are applied, the simplicity, the learnability and the understandability are negatively affected.

In [6], an industrial case study is conducted, in order to examine the correlation among code changes, reusability, design patterns, and class size. On the report of the results of the study, the number of changes is highly correlated to class size. Additionally, classes that play roles in design patterns or that are reused through inheritance are more change prone than others. Despite the study's good structure and validation, it investigates an individual maintainability aspect, change proneness, and does not mention maintainability issues such as change effort and design quality.

In [8], the authors present the investigation of correlations among class change proneness, the role that a class holds in a pattern and the kind of change that occurs. They use three open source projects in order to perform the empirical study. Concerning the majority of design patterns, the results of the study comply with common sense. However, in some cases, the conclusions differ from those expected.

3 Methodology

Wholin et.al suggests that there are three major empirical investigation approaches, surveys, case studies and experiments [23]. In this paper we have conducted a case study, exploiting the plethora of open source. On the contrary, surveys are not suitable for our research because in this case we would miss the patterns that were employed without intention by programmers. Finally, an experiment with open-source programmers would decrease the number of subjects in our research. In this section of the paper we describe the methodology of our case study. The case study of our research was based on the guidelines described in [14], and consisted of the following steps:

(a) Define hypothesis
(b) Select projects
(c) Method of comparison selection
(d) Minimization of confounding factors
(e) Planning the case study
(f) Monitoring the case study and
(g) Analyze and report the results

The hypotheses, i.e. step (a), are defined in section 3.1. Steps (b) and (d) which deal with project selection protocol and minimizing confounding factors are presented in section 3.2, accompanied with step (e). The methods used in analyzing the data, i.e. step (c), is presented in section 3.3, step (f), described in [14], is discussed in section 6. Finally, concerning step (g), we report the results on section 4 and discuss them in section 5.

3.1 Research Questions

In this section of the paper we state the research questions that are investigated in our study.

RQ1: Which is the frequency of design pattern application?
RQ2: Are there any differences in pattern application within the software categories under study?
RQ3: Are there any differences in the number of pattern participant classes across pattern types and software categories?

3.2 Case Study Plan

In this section of the paper we present the case study plan. According to [5] planning a case study is an important step for the validity of the study. Our plan involved a five step procedure described below:

(a) choose open source project categories
(b) identify a number of projects that fulfil certain selection criteria, for each a category
(c) perform pattern detection for every selected project. The pattern detection was conducted with an automated tool [20] that identifies instances of eleven (11) patterns of all GoF pattern categories (i.e. *Creational*, *Behavioural* and *Structural*)
(d) tabulate data
(e) analyze data with respect to the research questions

In this study the OSS project categories that have been considered are development tools, office/business applications, internet application, databases and computer games. These categories have been selected as highly active topics in open source communities [19]. From these categories we have selected projects that fulfilled the following criteria:

(a) Software written in java, due to limitations of pattern detection tool [20]. However, java is probably the most widely used programming language.
(b) software that provides binary code, due to limitations of pattern detection tool.
(c) software should be ranked in the fifty most successful projects of their category, according to sourceforge.net rating.
(d) software binary size should be larger than 100KB, in order not to be considered trivial.

In case studies, factors, other than the independent variables, which influence the value of the dependent variable, are considered confounding factors. The most important confounding factors in our research are considered to be the experience of the developer on design patterns and object-oriented programming in general. In our study we limit our analysis to automatically collected data. On the other hand, it is expected that in a random developer sample of a large developers' community, the distribution of skill and experience are closely near to the distribution of the population.

3.3 Data Analysis Methods

The dataset that has been created after design pattern detection consisted mainly of numerical data. On the completion of the pre-processing phase each project was characterized by 28 variables:

- name
- category
- number of downloads
- number of factory method instances
- number of prototype instances
- number of singleton instances
- number of creational pattern instances
- number of adapter instances
- number of composite instances
- number of decorator instances
- number of proxy instances
- number of structural pattern instances
- number of observer instances
- number of state-strategy instances
- number of template method instances
- number of visitor instances
- number of behavioural pattern instances
- average number of pattern participants per pattern (11 variables)

The analysis phase of our study has employed descriptive statistics, independent sample t-test and paired sample t-test. Concerning RQ_1, we have employed descriptive statistics and paired sample t-tests so as to compare the mean number of instances for each design pattern. In the investigation of RQ_2 and RQ_3, for similar reasons we have used descriptive statistics and independent sample t-tests. The statistical analysis has been performed with SPSS$^©$.

According to [23], one of the first steps during statistical analysis of the dataset is the elimination of outliers. In our study we identified and erased seventeen outliers. In most cases the observed extreme values where identified as maximum values, that is software that exhibit a very large number of pattern instances.

4 Results

In Table 1, the mean number of design pattern instances is presented. The data refer to the whole data set without discrimination across software categories. In addition to that, standard deviation of each variable is presented.

Table 1. Average Number of Pattern Instances

	Mean	**Std. Deviation**
Factory	3.21	7.21
Prototype	5.80	18.98
Singleton	13.99	19.16
Creational	*23.01*	*36.55*
Adapter	34.71	53.66
Composite	0.48	2.22
Decorator	2.53	6.50
Proxy	1.58	4.50
Structural	*39.30*	*61.17*
Observer	1.44	2.55
State	37.70	58.96
Template	5.93	8.52
Visitor	0.50	2.50
Behavioural	*45.65*	*66.19*

The results of Table 1 provide indications on the employment rate of each pattern in OSS. In order to be able to compare the mean values of each variable in a more elaborate way, we have performed 55 paired sample t-tests, i.e. one test for every possible pair of design patterns. The results of a t-test between two variables are interpreted by two numbers, the mean difference (diff) and the t-test significance (sig). The *diff* variable represents the difference of subtracting the mean value of the second variable, from the mean value of the first. Whereas, *sig* represents the possibility, that *diff* is not statistically significant. In Table 2, we present the statistically significant differences in pattern application.

Table 2. Significant paired sample t-tests on pattern employment difference

	diff	sig		diff	sig
Factory – Singleton	-10.78	0.00	*Decorator – Template*	-3.40	0.00
Factory – Adapter	-31.50	0.00	*Decorator – Visitor*	2.03	0.00
Factory – Composite	2.74	0.00	*Proxy – State*	-36.10	0.00
Factory - Proxy	1.64	0.03	*Proxy – Template*	-4.36	0.00
Factory - Observer	1.78	0.01	*Proxy - Visitor*	1.08	0.04
Factory – State	-34.45	0.00	*Observer – State*	-36.25	0.00
Factory - Template	-2.72	0.00	*Observer – Template*	-4.50	0.00
Factory – Visitor	2.71	0.00	*Observer - Visitor*	0.94	0.01
Prototype – Singleton	-8.19	0.00	*State – Template*	31.71	0.00
Prototype – Adapter	-28.91	0.00	*State – Visitor*	37.19	0.00
Prototype – Composite	5.33	0.00	*Template – Visitor*	5.43	0.00
Prototype - Decorator	3.27	0.04	*Adapter – Composite*	34.23	0.00
Prototype - Proxy	4.22	0.01	*Adapter – Decorator*	32.18	0.00
Prototype – Observer	4.36	0.02	*Adapter – Proxy*	33.13	0.00
Prototype – State	-31.84	0.00	*Adapter – Observer*	33.27	0.00
Prototype – Visitor	5.30	0.01	*Adapter - State*	-2.66	0.00
Singleton – Adapter	-20.72	0.00	*Adapter – Template*	28.77	0.00
Singleton – Composite	13.51	0.00	*Adapter – Visitor*	34.21	0.00
Singleton – Decorator	11.46	0.00	*Composite – Decorator*	-2.06	0.00
Singleton – Proxy	12.41	0.00	*Composite - Proxy*	-1.10	0.01
Singleton – Observer	12.55	0.00	*Composite – Observer*	-0.96	0.00
Singleton – State	-23.58	0.00	*Composite – State*	-37.22	0.00
Singleton – Template	8.06	0.00	*Composite – Template*	-5.46	0.00
Singleton – Visitor	13.49	0.00	*Decorator – State*	-35.14	0.00

In Table 3, the mean numbers of instances of each design patterns within the software categories under study are presented.

In order to statistically validate the results of the above table, we performed 42 independent sample t-tests, i.e. one test for each pattern for all the possible pairs of software categories. In Table 4, we provide the statistically significant results on comparing pattern application between software categories. The results are presented similarly to those of Table 2.

Table 3. Average Number of Pattern Instances among Software Categories

	Office / Business	**Internet**	**Development Tools**	**Database**	**Games**
Factory	9.24	3.00	1.00	2.64	0.50
Prototype	20.05	1.85	2.45	3.45	1.58
Singleton	29.43	13.55	8.30	7.36	11.67
Creational	*58.71*	*18.40*	*11.75*	*13.45*	*13.75*
Adapter	91.95	18.30	19.80	24.77	19.83
Composite	1.52	0.15	0.10	0.32	0.29
Decorator	6.95	1.50	1.50	2.14	0.75
Proxy	4.95	0.25	0.20	0.73	1.67
Structural	*105.38*	*20.20*	*21.60*	*27.95*	*22.54*
Observer	2.57	1.20	1.35	0.77	1.33
State	94.57	29.25	23.84	27.14	15.63
Template	11.71	5.80	4.10	6.05	2.42
Visitor	0.00	1.70	0.25	0.27	0.37
Behavioral	*108.86*	*37.95*	*29.84*	*34.23*	*19.75*

Table 4. Significant independent sample t-tests

	Pattern	**diff**	**sig**
Office/Business - Internet	Factory	6.24	0.05
Office/Business - Internet	Prototype	18.20	0.04
Office/Business - Internet	Singleton	15.88	0.03
Office/Business - Internet	Adapter	73.65	0.00
Office/Business - Internet	Decorator	5.45	0.06
Office/Business - Internet	Proxy	4.70	0.02
Office/Business - Internet	State	65.32	0.01
Office/Business - Internet	Template	5.91	0.08
Office/Business – Development Tools	Factory	8.24	0.01
Office/Business – Development Tools	Prototype	17.60	0.05
Office/Business – Development Tools	Singleton	21.13	0.01
Office/Business – Development Tools	Adapter	72.15	0.00
Office/Business – Development Tools	Decorator	5.45	0.06
Office/Business – Development Tools	Proxy	4.75	0.02
Office/Business – Development Tools	State	70.73	0.01
Office/Business – Development Tools	Template	7.61	0.01
Office/Business – Database	Factory	6.60	0.04
Office/Business – Database	Prototype	16.59	0.07
Office/Business – Database	Singleton	22.07	0.00
Office/Business – Database	Adapter	67.18	0.00
Office/Business – Database	Proxy	4.23	0.03
Office/Business – Database	Observer	1.80	0.05
Office/Business – Database	State	67.44	0.01

Table 4. (*Continued*)

	Pattern	**diff**	**sig**
Office/Business – Database	Template	5.67	0.08
Office/Business – Games	Factory	8.74	0.01
Office/Business – Games	Prototype	18.46	0.04
Office/Business – Games	Singleton	17.76	0.02
Office/Business – Games	Adapter	72.12	0.00
Office/Business – Games	Decorator	6.20	0.03
Office/Business – Games	State	78.95	0.02
Office/Business – Games	Template	9.30	0.00
Internet – Development Tools	Factory	2.00	0.09
Internet - Database	Singleton	6.19	0.09
Internet – Games	Factory	2.50	0.03
Internet – Games	State	13.63	0.09
Database – Games	Factory	2.14	0.07
Database – Games	Template	3.63	0.07

In Table 5, we present the mean numbers of classes that participate in each design patterns within the software categories under study. Additionally, Table 6 presents the statistically significant differences between the mean values of number of classes that participate in design patterns, among software categories.

Table 5. Average Number of Pattern Participating Classes among Software Categories

	Office / Business	**Internet**	**Development Tools**	**Database**	**Games**	**Overall**
Factory	6.35	5.93	6.77	6.93	6.21	6.46
Prototype	7.84	7.69	10.80	7.23	8.74	8.16
Singleton	1.00	1.00	1.00	1.00	1.00	1.00
Adapter	2.00	2.00	2.00	2.00	2.00	2.00
Composite	7.42	20.50	12.00	6.20	9.33	9.84
Decorator	11.77	17.12	10.33	11.93	17.23	13.46
Proxy	2.20	2.29	2.07	3.20	2.11	2.32
Observer	10.39	11.90	11.00	6.65	10.67	10.33
State	7.15	9.04	6.92	6.40	8.68	7.47
Template	6.83	5.79	5.82	6.35	7.88	6.45
Visitor	0.00	7.15	3.24	4.72	2.00	5.21

Table 6. Significant independent sample t-tests

	Pattern	**diff**	**sig**
Office/Business – Development Tools	Prototype	-2.96	0.00
Internet – Development Tools	Prototype	-3.11	0.00
Databases – Development Tools	Prototype	-3.57	0.00
Office/Business – Internet	Composite	-13.08	0.01
Databases – Development Tools	Composite	-5.8	0.00
Databases – Internet	Composite	-14.3	0.02
Internet – Games	Composite	11.17	0.03
Office/Business – Internet	Decorator	-5.35	0.00
Office/Business – Games	Decorator	-5.46	0.00
Internet – Development Tools	Decorator	6.79	0.00
Internet – Databases	Decorator	5.19	0.00
Games – Development Tools	Decorator	6.9	0.00
Games – Databases	Decorator	5.3	0.00
Internet – Development Tools	Proxy	0.22	0.04
Databases – Office/Business	Observer	-3.74	0.01
Databases – Internet	Observer	-5.25	0.00
Databases – Development Tools	Observer	-4.35	0.00
Databases – Games	Observer	-4.02	0.00
Office/Business – Internet	State	-1.89	0.00
Internet – Development Tools	State	2.12	0.00
Databases – Office/Business	State	-0.75	0.00
Databases – Internet	State	-2.64	0.00
Databases – Development Tools	State	-0.52	0.01
Office/Business – Games	State	-1.53	0.00
Games – Development Tools	State	1.76	0.00
Games – Databases	State	2.28	0.00
Games – Development Tools	Template	2.06	0.01
Games – Databases	Template	1.53	0.01
Internet – Games	Visitor	5.15	0.00
Games – Databases	Visitor	-2.72	0.00
Games – Development Tools	Visitor	-1.24	0.00
Databases – Internet	Visitor	-2.53	0.00
Databases – Development Tools	Visitor	1.03	0.00
Internet – Development Tools	Visitor	3.91	0.00

5 Discussion

This section of the paper discusses the results of our case study. The discussion is organized in subsections according to the research questions that have been introduced in the beginning of the paper. Thus, section 5.1 discusses which design patterns are more frequently used in open source software development, section 5.2 discusses the usage of each design pattern on three software categories and section 5.3 discusses the size of the design patterns used in open source software in general.

5.1 Design Pattern Application

The results of Table 1, clearly suggest that some patterns are more frequently applied in open source than others. In addition to that, Table 2 suggests that pattern usage intensity classifies patterns in seven categories as shown in Figure 1. Patterns on the top of Figure 1 are statistically significantly employed more times in open source software projects than those closer to the bottom of Figure 1.

Some of the results that are presented in Figure 1 are reasonable, whereas some findings are surprising. As one would expect, the *Adapter* pattern is frequently used, because reusing classes of others is a common practice in open source software communities. In such cases, adapter provides a mechanism for adopting the new class in the existing system without modifying the existing code. In addition to that, the Adapter's rationale is akin to the basic concepts of object - oriented programming and thus it might be explicitly used by the developers. Furthermore, the *State* pattern as expected ranks high, because its background requires just the proper use of inheritance. Finally, more difficult to understand patterns, according to authors' opinion, such as *Visitor* and *Observer*, are not often employed by open source developers.

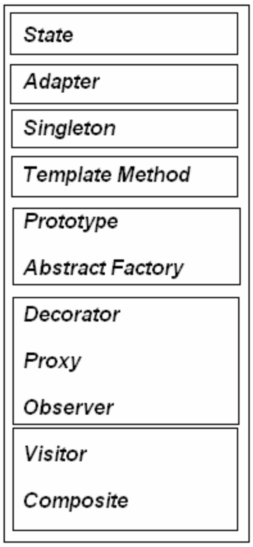

Fig. 1. Design Pattern Usage Levels

On the contrary, although the *Singleton* pattern is quite complex in its structure [7] and it was expected not to be as popular, it is ranked as the 3rd most used pattern. A possible reason for this is the limitation of the case study subject to the Java languages, where Singleton is implemented by a simple instantiation mechanism. Another bizarre observation is that the *Decorator* pattern is more frequently used than the *Composite* pattern. The Composite pattern is the base of the Decorator pattern and therefore it was expected to be more frequently employed. Summing up the above, open source developers employ easy to understand patterns more than more elaborate ones. A possible reason for this is that typically there are no detailed formal design activities before programming in open source.

5.2 Design Patterns and Software Categories

As it is observed in Table 3, design pattern usage within every category follows similar distribution as in open source software development in general. However, comparing pattern application across software categories, the results suggest that some patterns are more frequently applied in one category, than another. From Tables 3 and 4, we observe that *Office/Business* applications employ statistically significantly more patterns than any other category. Furthermore, rather surprising is the fact that in general *Development Tools* employ a relative limited number of pattern instances w.r.t the other software categories. One would expect that developers of this category would be familiar with patterns and use them. Figure 2 presents the ranking of pattern usage among software categories.

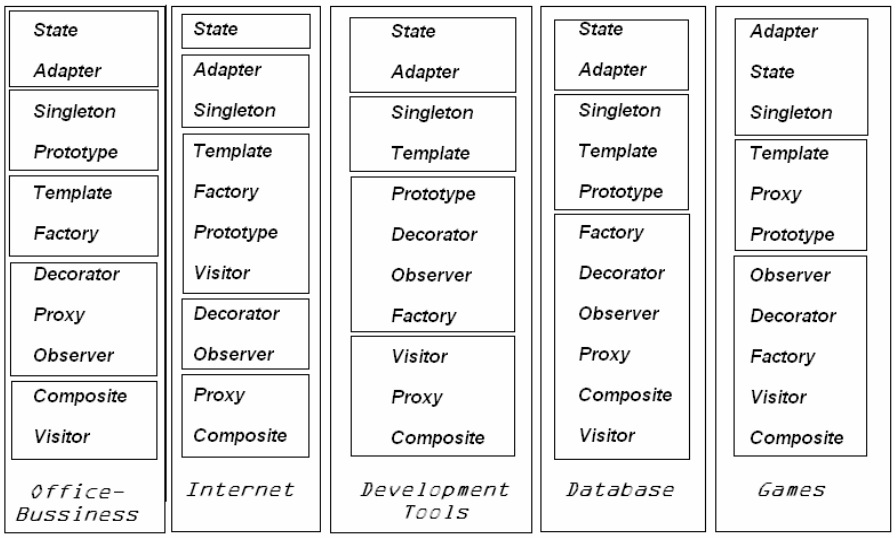

Fig. 2. Design Pattern Usage Levels across Categories

From Figures 1 and 2, it is suggested that *Decorator and Observer* patterns are more highly ranked in *Development Tools* than in the other categories. This fact can be justified by the expectation that developers of this category are more likely to be aware of the pattern, which is not easily applied by chance. In addition to that, the *Adapter* pattern is the most frequently employed pattern in the *Games* category. This fact suggests that game developers might perform more "as is" reuse activities than other programmers. This observation is interesting and deserves further investigation.

Additionally, the *Visitor* pattern appears to be more applicable in *Internet* application and the *Proxy* pattern more applicable in *Games*. Thus, we can assume that domain specific requirements (functional or non-functional) of this category might be implemented with the use of these patterns.

5.3 Design Pattern Size among Software Categories

This section of the paper discusses the most important findings on the variation of the size of deign patterns among software categories. The "largest" patterns, with respect to number of classes appear to be *Decorator* and *Observer*, whereas the pattern with the least pattern, apart from Singleton and Adapter that employ a standard number of classes, appears to be *Visitor*.

Within software categories, we found that when the *Prototype* pattern is applied in *Development Tools*, it appears to employ statistically significant more classes than when applied in any other software category. Similarly, the *Composite* pattern instances in *Internet* applications are larger than the Composite instances in other software categories. Concerning *Decorator*, we identified that the larger pattern instances can be found in *Games* and *Internet* applications. Finally, the smallest *Observer* instances can be identified in *Database* applications.

These findings can be used in studies that investigate pattern effect on software quality, with respect to their size, the role that each class plays in a pattern and for case study construction.

6 Threats to Validity

This section of the paper presents the internal and external threats to the validity of our case study. Firstly, since the subjects have been open-source projects, the results may not apply to closed source software. Concerning the empirical study internal validity, the existence of confounding factors is analyzed in section 3.4. The most confounding factor is that the study cannot take into account the knowledge of developer's on design patterns, but it can be reasonably assumed that the familiarity degree with pattern knowledge across different application domains, corresponds to the distribution of the population.

In addition to that, the sample size is quite small with respect to the total number of open source software and generalizing the results from the sample to the population is risky. In addition, the dataset consisted only from Java projects, since the tool we used was able to detect design patterns only in binary java files. Moreover, only one repository, namely Sourceforge, has been mined.

7 Conclusions

This study is an extension of a previous work of the authors. It empirically investigates the usage of object oriented design patterns in open source software development. For this reason the authors have explored 129 open source software from five categories, i.e. development tools, business/office application, internet applications, database applications and computer games.

The results of the study confirm that "easy to use" design patterns, such as *Adapter*, *State* and *Singleton* are more frequently applied in open source. More elaborate patterns such as *Visitor* and *Observer* are more frequently employed by development tool programmers, most probably due to their better understanding and knowledge on software engineering issues. Additionally, the frequent application of the *Adapter* pattern in computer games might indicate higher reuse levels in this type of software applications. Finally, the results suggested that among software categories, *Office/Business* application employ statistically significantly more design patterns than other categories and that the size of design patterns vary among software genres.

As future work we are about to create a web repository on the findings of the design pattern detection process, so as to enhance design pattern reuse opportunities. In addition to that, we are going to explore projects written in other programming languages, such as C++. More software categories and open source projects are going to be investigated. Finally, the most important findings of the study, such as the reuse increased reuse opportunities in games, the limited number of pattern instances in development tools and the factors that influence design pattern usage are going to be investigated.

References

1. Ampatzoglou, A., Charalampidou, S., Savva, K., Stamelos, I.: An empirical study on design pattern employment in open-source software. In: 5th Working Conference on the Evaluation of Novel Approaches in Software Engineering, pp. 275–284. INSTICC, Athens (2010)
2. Ampatzoglou, A., Chatzigeorgiou, A.: Evaluation of object-oriented design patterns in game development. Information and Software Technology 49(5), 445–454 (2007)
3. Arnout, K., Meyer, B.: Pattern componentization: the factory example. Innovations in Systems and Software Technology 2(2), 65–79 (2006)
4. Bansiya, J., Davis, C.: A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Transaction on Software Engineering 28(1), 4–17 (2002)
5. Basili, V.R., Selby, R.W., Hutchens, D.H.: Experimentation in Software Engineering. IEEE Transactions on Software Engineering 12(7), 733–743 (1986)
6. Bieman, J.M., Jain, D., Yang, H.J.: OO design patterns, design structure, and program changes: an industrial case study. In: 17th International Conference on Software Maintenance, ICSM 2001, pp. 580–591. IEEE Computer Society, Florence (2001)
7. Chatzigeorgiou, A.: Object-Oriented Design: UML, Principles, Patterns and Heuristics, 1st edn. Kleidarithmos, Athens (2005)
8. Di Penta, M., Cerulo, L., Gueheneuc, Y.G., Antoniol, G.: An Empirical Study of Relationships between Design Pattern Roles and Class Change Proneness. In: 24th International Conference on Software Maintenance, ICSM 2008, pp. 217–226. IEEE Computer Society, Beijing (2008)
9. Feller, J., Fitzgerald, B.: Understanding open source software development, 1st edn. Addison-Wesley Longman, Boston (2002)

10. Gamma, E., Helms, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Professional, Reading (1995)
11. Hsueh, N.L., Chu, P.H., Chu, W.: A quantitative approach for evaluating the quality of design patterns. Journal of Systems and Software 81(8), 1430–1439 (2008)
12. Huston, B.: The effects of design pattern application on metric scores. Journal of Systems and Software 58(3), 261–269 (2001)
13. Khomh, F., Gueheneuc, Y.G.: Do design patterns impact software quality positively? In: 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, pp. 274–278. IEEE Computer Society, Athens (2008)
14. Kitchenham, B., Pickard, L., Pfleeger, S.L.: Case Studies for Method and Tool Evaluation. IEEE Software 12(4), 52–62 (1995)
15. McShaffry, M.: Game Coding Complete. Paraglyph Press, Arizona (2003)
16. Meyer, B., Arnout, K.: Componentization: The Visitor Example. IEEE Computer 39(7), 23–30 (2006)
17. Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G.: A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering 27(12), 1134–1144 (2001)
18. Samoladas, I., Stamelos, I., Angelis, L., Oikonomou, A.: Open source software development should strive for even greater code maintainability. Communications of the ACM 47(12), 83–87 (2004)
19. Sowe, S.K., Angelis, L., Stamelos, I., Manolopoulos, Y.: Using Repository of Repositories (RoRs) to Study the Growth of F/OSS Projects: A Meta-Analysis Research Approach. In: OSS 2007, Open Source Software Conference, pp. 147–160. Springer, Limerick (2007)
20. Tsantalis, N., Chatzigeorgiou, V., Stephanides, G., Halkidis, S.T.: Design Pattern Detection using Similarity Scoring. IEEE Transaction on Software Engineering 32(11), 896–909 (2006)
21. Vokác, M., Tichy, W., Sjøberg, D.I.K., Arisholm, E., Aldrin, M.: A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns - A Replication in a Real Programming Environment. Empirical Software Engineering 9(3), 149–195 (2003)
22. Wendorff, P.: Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In: 5th European Conference on Software Maintenance and Reengineering, CSMR 2001, pp. 77–84. IEEE Computer Society, Lisbon (2001)
23. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering, 1st edn. Kluwer Academic Publishers, Boston (2000)