2017 43rd Euromicro Conference on Software Engineering and Advanced Applications

How the use of design patterns affects the quality of software systems: a preliminary investigation

Carmine Gravino
Department of Computer Science
University of Salerno
Fisciano (SA), Italy
Email: gravino@unisa.it

Michele Risi
Department of Computer Science
University of Salerno
Fisciano (SA), Italy
Email: mrisi@unisa.it

***Abstract*—In this paper we analyze at the class level the quality of the software portions including classes participating in design patterns instances (DP classes) with respect to the remaining software portions (NoDP classes). The performed study is based on 10 software systems from which information about design pattern instances and CK (Chidamber and Kemerer) metrics were obtained by exploiting repositories of pattern instances and the tool Understand, respectively. The analysis revealed that the use of design patterns impacts on the quality of the software.**

I. Introduction

Design patterns [1] can be used both in the process of designing a software system, and in reverse engineering to review and improve an existing system through code refactoring and maintenance operations. Indeed, some studies have also shown that the availability of documentation explaining the use of design pattern instances in software systems allows to better comprehend the source code of those systems, and thus facilitating their maintenance (see e.g., [2]).

In this paper we present the results of a study carried out to analyze at the class level the quality of the software portions including classes participating in design patterns instances (DP classes) with respect to the remaining software portions (NoDP classes). To this aim, we exploited some (CK) metrics proposed by Chidamber and Kemerer [3], which provide an indication of the software quality with respect to crucial properties, e.g., cohesion and coupling [4]. The study is based on 10 software systems from which we obtained information on design pattern instances and CK metrics.

Structure of the paper. Related work is presented in Section II. Section III presents the design of the performed study, while the results are discussed in Section IV. Conclusion concludes the paper.

II. Related Work

Several studies have highlighted that the design based on patterns improves the quality of object-oriented software systems [1], [5], e.g., reducing the development time and program quality, producing reusable software, improving the documentation, and increasing flexibility and elegance of the developed software system. Among them, Scanniello *et al.* [2] have studied the importance of documenting design pattern instances and how it can affect the comprehensibility of source code. To this aim, they conducted a family of four controlled experiments with participants having different experience (i.e., professionals and Bachelor, Master, and PhD students). Design patterns were graphically documented with UML class diagrams, or textually documented as comments in the source code. The results indicated that documenting design-pattern instances yields an improvement in correctness of understanding source code for those participants with an adequate level of experience. Prechelt *et al.* [5] have performed a series of experiments to test whether pattern comment lines help during maintenance phase when they are well documented in the source code. The results highlighted that the quality is improved by reducing the time required for performing changes. Replications of this experiment were executed in order to examine the importance of design patterns in source code comprehension [6]. Aversano *et al.* [7] have investigated the frequency and scope of modifications of design patterns by examining three open source systems. The results have pointed out that design patterns closely related to the application purpose are modified more frequently than others. Gustafsson *et al.* [8] have presented a tool that allows to assure software quality by computing various kinds of design metrics from the system architecture, by automatically exploring instances of design patterns and anti-patterns from the architecture, and by reporting potential quality problem to the designers. Khomh *et al.* [9] assessed the impact that specific design pattern roles play on the quality of classes, analyzed in terms of CK measures as done in our study. They found a significant increase in many metric values and a significant increase in the frequencies and numbers of changes when classes play two roles in design pattern instances. Our study can be considered a further assessment with respect to [9], exploiting more design patterns (20 vs. 6) and software systems (10 vs. 6).

III. Study design

We defined the following research questions RQ_X: *Are the classes participating in design pattern instances characterized by better values of the CK measure X with respect to the classes that are not involved in design pattern instances?* where X can be one of the measures [3]: WMC (Weighted Methods per class), LCOM (Lack of Cohesion in Methods), AMS (AvgLineCode), CLOC (CountLineCom-

978-1-5386-2141-7/17 $31.00 © 2017 IEEE
DOI 10.1109/SEAA.2017.32

ment), LOC (CountLineCode), CLOC/NL (CountLineComment/CountLine). As for the measures DIT (Depth of Inheritance Tree) and CBO (Coupling Between Object classes), we have analyzed the values characterizing the classes that are involved in the design pattern instances. Indeed, due to the definitions of these measures we cannot fairly compare the values obtained for the DP classes with ones obtained for the NoDP classes. Regarding the design patterns, we took into account those proposed by Gamma *et al.* [1].

The software systems we considered are publicly available: QuickUML 2001, Lexi 0.1.1 alpha, Refactory 2.6.24, NetBeans 1.0.x, JUnit 3.7, JHotDraw 5.1, JHotDraw 6.0, MapperXML 1.9.7, Nutch 0.4, and PMD 1.8. They have been used in several studies to assess the accuracy of design pattern recovery approaches (e.g., [10], [11], [12], [13], [14]).

As for the design pattern instances to consider in our study, we exploited P-MARt [15] which is a repository of micro-architectures similar to design motifs, where a design motif represents the solution of a design pattern. Furthermore, we employed instances recovered by the following tools, and made available on the web by the researchers working on them: DPD [12], ePAD [16], RM [13], DPF [14]. In particular, we first considered the instances present in P-MARt since they cover all the GoF patterns. Then, we added those recovered by the above tools and not present in P-MARt. To obtain CK metrics values we used the tool Understand [17], which is an integrated and interactive development environment conceived to support software developers in documenting, comprehending, and maintaining their source code (e.g., C, C++, C#, Java).

Regarding data analysis, we employed simple descriptive statistics and the Mann-Whitney test [18] to verify whether there is statistical significant difference among CK metrics values of DP and NoDP classes. The results were intended as statistically significant at $\alpha = 0.05$. Furthermore, we employed the Cliff's d non-parametric effect size measure in order to provide information about the magnitude of the difference between two distributions [19]. As for the magnitude of the effect size we considered the following classification: negligible ($d < 0.147$), small ($0.147 \leq d < 0.33$), medium ($0.33 \leq d \leq 0.474$), and large ($d > 0.474$).

As for the threats that could affect our study, we tried to mitigate errors in the identification of design pattern instances by exploiting public repositories and results made available by other researchers who assessed their data through empirical validations (e.g., [10], [11], [12], [13], [14]). We employed software systems analyzed in previous studies. Similarly, among the proposals provided to assess the quality of software systems we used CK measures since they have been widely employed in previous studies (e.g., [4], [20]). Finally, we carefully applied the statistical tests performed by verifying all the required assumptions.

IV. Results and Discussion

In order to have an idea of the portions of code interested by the design pattern instances, in Table I, we have reported for each software system in our study the number of pattern instances and the number of DP and NoDP classes. To better highlight the use of the design patterns, we have also determined the percentage of DP classes with respect to the total number of classes. We can observe that the number of NoDP classes are greater than the number of DP classes, for all the considered software systems, except for JHotDraw5.1. This is not a surprise since JHotDraw5.1 has been developed with the intent of using design patterns by the people that introduce them. By analyzing the % of Table I, we can note that among the remaining systems, JHotDraw 6.0, JUnit, JRefactory, QuickUML, and MapperXML are characterized by values greater than 20, i.e., more than 1/5 of the classes are involved in the design pattern instances. The systems PMD, Nutch, NetBeans, and Lexi have a percentage less than 15%. This confirms the level of variability of the considered systems, also from the point of view of design pattern use.

TABLE I
pattern instances and # of DP and NoDP classes (and their % with respect to the total # of classes), for each system

System	# pattern instances	# DP classes (and %)	# NoDP classes (and %)
HotDraw 5.1	68	119 (71)	49 (29)
JHotDraw 6.0	82	76 (23)	253 (77)
JUnit	14	25 (27)	68 (73)
JRefactory	41	198 (34)	392 (66)
QuickUML	18	54 (25)	162 (75)
PMD	14	52 (11)	440 (89)
Nutch	15	40 (13)	260 (87)
MapperXML	25	60 (24)	186 (76)
NetBeans	26	200 (4)	5347 (96)
Lexi	5	8 (6)	117 (94)

In Table II, the entry "DP" (NoDP, respectively) for the cell (i,j) means that the set of DP classes in the system *i* are characterized by a mean value for the measure *j* better than the one calculated for the NoDP (DP, respectively) classes. The entry "=" means that the mean values of the measure *j* calculated for the set of the DP and NoDP classes are comparable. For CBO and DIT, we have reported the values of the mean for the set of DP classes and the standard deviation (St.Dev) since we cannot compare the distributions for the set of DP and NoDP classes, due to the definition of these two measures. So, we will discuss only the distributions.

In the following we discuss the results for each CK measure.

RQ$_{AMS}$: the ratio between the numbers of source code lines and the number of methods is higher for NoDP classes in 8 systems (see Table II) for the majority of the patterns (Factory Method, Decorator, Composite, Prototype, Template Method, Visitor, State, Command, Singleton, Observer, Builder, Abstract Factory, Iterator, Adapter, Facade, Strategy, Proxy, Memento). Differently, for Nutch, the AMS mean value for DP classes is higher than the one for NoDP classes for 5 (Iterator, Memento, Command, Strategy and Template Method) of the 8 patterns present. For JHotDraw5.1 AMS is greater for DP classes in the case of 5 patterns (Singleton, Adapter, Observer, State, Null Object) with respect to the value for NoDp classes, while for other 5 patterns (Factory Method, Decorator, Composite, Prototype, Template Method) is the contrary. For this reason, in Table II, the entry "=" is reported. The Mann-Whitney test revealed that there is significant difference in

TABLE II
SUMMARY OF RESULTS CONSIDERING THE MEAN OF CK MEASURE VALUES FOR EACH SOFTWARE SYSTEMS

System	AMS	LOC	CBO	CLOC	DIT	LCOM	WMC	CLOC/NL
JHotDraw 5.1	=	DP	Mean(DP classes)= 3.98; Dev.St.(DP classes) = 1.56	DP	Mean(DP classes) = 3; Dev.St.(DP classes) = 0.65	DP	DP	=
JHotDraw 6.0	NoDP	DP	Mean(DP classes) = 10.87; Dev.St.(DP classes) = 5.74	DP	Mean(DP classes) = 2; Dev.St.(DP classes) = 0.5	DP	DP	=
JUnit	NoDP	DP	Mean(DP classes) = 2.5; Dev.St.(DP classes) = 2.55	DP	Mean(DP classes) = 2; Dev.St.(DP classes) = 0.48	DP	DP	DP
JRefactory	NoDP	NoDP	Mean(DP classes) = 7.35; Dev.St.(DP classes) = 4.24	=	Mean(DP classes) = 2; Dev.St.(DP classes) = 0.33	DP	DP	=
QuickUML	NoDP	DP	Mean(DP classes) = 5; Dev.St.(DP classes) = 4	DP	Mean(DP classes) = 2; Dev.St.(DP classes)= 0.86	DP	=	DP
Lexi	NoDP	DP	Mean(DP classes) = 2.5; Dev.St.(DP classes) = 6.64	DP	Mean(DP classes) = 2; Dev.St.(DP classes) = 0.55	NoDP	NoDP	DP
MapperXML	NoDP	NoDP	Mean(DP classes) = 4.11; Dev.St.(DP classes) = 1.89	DP	Mean(DP classes) = 3; Dev.St.(DP classes) = 1.3	DP	DP	DP
PMD	NoDP	DP	Mean(DP classes) = 15.38; Dev.St.(DP classes) = 14.6	DP	Mean(DP classes) = 1; Dev.St.(DP classes) = 0.45	DP	DP	=
NetBeans	NoDP	DP	Mean(DP classes) = 5.33; Dev.St.(DP classes) = 2.83	DP	Mean(DP classes) = 2; Dev.St.(DP classes) = 0.79	DP	DP	DP

the case of only 3 systems (JHotDraw6.0, JRefactory, and MapperXML) with a medium (in few cases large) effect size. Thus, *the DP classes are not characterized by better AMS values with respect to the NoDP classes*.

RQ$_{LOC}$: the mean number of LOC characterizing the DP classes is greater than the one of NoDP classes, for the majority of design patterns (Factory Method, Decorator, Composite, Adapter, Prototype, Observer, State, Strategy, Template Method, Visitor, Abstract Factory, Iterator, Command, Facade, Singleton, Builder, Proxy) in 7 systems. For the remaining 3 systems (JRefactory, MapperXML, Nutch) we have a different situation. Indeed, in JRefactory for just 2 design patterns (State and Builder) of the 8 design patterns present, DP classes are characterized by a greater mean number of LOC, while for the remaining design patterns (Singleton, Adapter, Visitor, Template Method, Observer e Factory Method) the NoDP classes have a greater mean number of LOC. Similarly, in MapperXML the mean number of LOC is grater for the DP classes for 3 design patterns (Facade, Singleton e Strategy), while for the remaining 6 design patterns that can be found in the code, the mean number of LOC is greater for the NoDP classes. For the system Nutch in 3 cases (Bridge, Singleton, and Adapter) DP classes are characterized by a greater mean number of LOC, while for the remaining design patterns (Iterator, Memento, Command, Strategy, and Template Method) the NoDP classes have a greater value. The performed statistical test revealed that there is significant difference between LOC characterizing the DP and NoDP classes in all the systems with a medium (in few cases large) effect size, except for QuickUML, Nutch, JRefcatory, and MapperXML. Thus, *the DP classes are characterized by a greater number of LOC*.

RQ$_{CLOC}$: the mean number of comments results to be greater for the DP classes in 8 systems for the majority of the design patterns (Factory Method, Decorator, Singleton, Composite, Adapter, Prototype, Command, Observer, State, Strategy, Template Method, Builder, Visitor, Abstract Factory, Iterator, Facade). Just in 2 systems (JRefactory and Nutch) DP and NoDP classes are characterized by comparable values of the mean of CLOC. Indeed, for 4 design patterns (Builder, Visitor, Template Method, and Factory Method) DP classes have a greater value, while for the remaining Singleton, Adapter, State, and Observer patterns the mean number of comments is greater in the case of NoDP classes. Similarly, for Nutch, and the design patterns Iterator, Memento, Command and Strategy, DP classes are characterized by a greater mean number of CLOC, while for the remaining design patterns (Bridge, Singleton, Adapter, and Template Method) NoDP classes have a grater value. The performed statistical test revealed that there is significant difference between CLOC characterizing the DP and NoDP classes in all the systems with a medium (in few cases large) effect size. Thus, *the DP classes are characterized by a greater number of CLOC*.

RQ$_{LCOM}$: the mean value for the LCOM measure is greater for the DP classes in 8 systems for the majority of the design patterns (Singleton, State, Builder, Template Method, Factory Method, Proxy, Iterator, Composite, Adapter, Observer, Strategy, Abstract Factory, Command, Prototype, Facade, Visitor, Decorator, Null Object). The opposite is verified in the two remaining systems: Lexi and Nutch. In the case of Lexi for all the design patterns NoDP classes are characterized by higher mean with respect to the DP classes. Differently, for Nutch and 3 design patterns (Iterator, Singleton and Strategy) DP classes have a greater mean value. The performed statistical test revealed that there is significant difference between LCOM characterizing the DP and NoDP classes in all the systems with a medium (in few cases large) effect size, except for JRefcatory, MapperXML, Nutch, and PMD. Thus, *the DP classes are characterized by a better result in terms of LCOM with respect to the NoDP classes*.

RQ$_{WMC}$: the mean value of the method count for a class results to be greater for the DP classes in 7 systems for the majority of the patterns (Singleton, Observer, Builder, Factory Method, Decorator, Composite, Adapter, Prototype, Command, State, Strategy, Template Method, Visitor, Abstract Factory, Iterator, Facade, Proxy). Differently, in the case of the system Lexi the mean value is greater for the NoDP classes, while for the remaining 2 systems, QuickUML and Nutch, the mean value characterizing DP and NoDP classes are comparable. Indeed, in the case of QuickUML the DP classes result to be more complex (i.e., mean value of WMC greater) for the patterns Singleton, Builder, Template Method, Strategy, Prototype, while for the remaining patterns (Abstract Factory, Composite, Command, Observer, State) less complex. The statistical test revealed that there is significant difference between WMC characterizing the DP and NoDP classes in all the systems with a medium (in few cases large) effect size, except for JRefcatory, MapperXML, Nutch, and QuickUML. Thus, *the DP classes are more complex than the NoDP classes*.

RQ$_{CLOC/NL}$: in mean the percentage of lines of comments is greater for the DP classes in 5 systems for the majority of the

patterns (Observer, Builder, Singleton, Null Object, Template Method, Visitor, Iterator, Adapter, Facade, Strategy, Abstract Factory, Composite, Observer, State, Decorator, Bridge, Command). For the remaining 5 software systems the mean values of CLOC/NL for DP and NoDP classes are comparable. The statistical test revealed that there is significant difference between CLOC/NL characterizing the DP and NoDP classes in all the systems with a medium (in few cases large) effect size, except for PMD. Thus, *the DP classes are characterized by a better result with respect to the NoDP*.

CBO: overall, the mean values for CBO are in the range 1-4. We have considered values in the range 1-2 "low", while "medium" has been associated to the range 3-4. Values equal or higher than 5 have been considered "high". From Table II, we can observe that the coupling level in the case of DP classes is high for the majority of the systems (QuickUML, JRefactory, NetBeans, JHotDraw 6.0, Nutch, PMD). For JHotDraw5.1 and MapperXML the values for CBO can be considered medium, while for Lexi e JUnit the coupling level can be classified as low. The statistical test revealed that there is significant difference between CBO characterizing the DP and NoDP classes in all the systems with a medium (in few cases large) effect size, except for PMD. Thus, *the DP classes are characterized by a high level of coupling*. This can be considered an expected results due to the definitions of the design patterns, that are based on the use of relationships such as hierarchy, delegations, use, etc.

DIT. Overall, the mean values for DIT are in the range 1-4. The values in the range 0-1 are classified as "low", while higher values are considered "medium-high"". Table II suggests that the mean values for DIT in the case of the DP classes are in the range 1-3, except for Nutch and PMD that are characterized by low values. Thus, *the classes involved in pattern instances are characterized by medium-high values of the maximum inheritance path from the class to the root class*. The Mann-Whitney test revealed that there is significant difference between DIT characterizing the DP and NoDP classes in all the systems with a medium (in few cases large) effect size, except for JRefactory, Nutch, and QuickUML.

V. Conclusion

The above analysis has revealed that for the DP classes more lines of comments have been written by developers. This can be considered an expected result, since developers tend to document pattern instances when they knowingly decide to use them [2]. The analysis about AMS has shown that the methods of DP classes are characterized by a less number of lines of code with respect to the NoDP classes. This should improve the comprehensibility of the source code [20]. The results about LOC measure have also shown that in general the developers wrote more lines of code for a class in the set of DP classes with respect to NoDP classes. As an unexpected result, we have observed that overall the DP classes are characterized by higher values of LCOM (and so worse) with respect to NoDP classes. This means that the level of cohesion seems to be lower for DP classes. Furthermore, the analysis of distributions of WMC values has highlighted that the DP classes are also more complex. The DP classes are also characterized by a high level of coupling. So, complex code. These can be considered interesting results that deserve further analysis in the future. Indeed, it is widely known that the use of design patterns improve the comprehensibility of the code, when the patterns are well documented [2]. Thus, we are in the case that we structure our code in a more complex way but this choice allow us to re-use available and tested solutions that can also improve comprehensibility of the code if the pattern instances are well documented. This point deserves further investigation and analysis in our future researches.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] G. Scanniello, C. Gravino, M. Risi, G. Tortora, and G. Dodero, "Documenting design-pattern instances: A family of experiments on source-code comprehensibility," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 14:1–14:35, 2015.

[3] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[4] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies, "Class level fault prediction using software clustering," in *Conf. on Aut. Softw. Eng.*, 2013, pp. 640–645.

[5] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 595–606, 2002.

[6] M. Vokác, W. F. Tichy, D. I. K. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns-a replication in a real programming environment," *Em. So. En.*, vol. 9, no. 3, pp. 149–195, 2004.

[7] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," in *European Softw. Eng. Conf. and Foundations of Softw. Eng.*, 2007, pp. 385–394.

[8] J. Gustafsson, J. Paakki, L. Nenonen, and A. I. Verkamo, "Architecture-centric software evolution by software metrics and design patterns," in *European Conf. on Softw. Mainten. and Reeng.*, 2002, pp. 108–115.

[9] F. Khomh, Y. G. Gueheneuc, and G. Antoniol, "Playing roles in design patterns: An empirical descriptive and analytic study," in *2009 IEEE International Conference on Software Maintenance*, 2009, pp. 83–92.

[10] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *J. of Systems and Softw.*, vol. 82, no. 7, pp. 1177–1193, 2009.

[11] ——, "Improving behavioral design pattern detection through model checking," in *European Conf. on Softw. Maintenance and Reengineering*. IEEE Computer Society, 2010, pp. 176–185.

[12] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, 2006.

[13] G. Rasool and P. Mäder, "Flexible design pattern detection based on feature types." in *Conf. on Aut. Softw. Eng.*, 2011, pp. 243–252.

[14] M. L. Bernardi, M. Cimitile, and G. A. D. Lucca, "Design pattern detection using a DSL-driven graph matching approach," *Journal of Software: Evolution and Process*, vol. 26, no. 12, pp. 1233–1266, 2014.

[15] P-MARt, "P-MARt," http://www.iro.umontreal.ca/ labgelo/p-mart/.

[16] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, and C. Pirolli, "ePadEvo: A tool for the detection of behavioral design patterns," in *Intl. Conf. on Softw. Maint. and Evol.*, 2015, pp. 327–329.

[17] Understand, "Understand," https://scitools.com.

[18] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley India Pvt. Limited, 2006.

[19] V. Kampenes, T. Dyba, J. Hannay, and I. Sjoberg, "A systematic review of effect size in software engineering experiments," *Information and Softw. Technology*, vol. 4, no. 11-12, pp. 1073–1086, 2007.

[20] L. C. Briand, J. Wüst, and H. Lounis, "Replicated case studies for investigating quality factors in object-oriented designs," *Emp. Softw. Eng.*, vol. 6, no. 1, pp. 11–58, 2001.