Evaluation of Design Pattern Utilization and Software Metrics in C# Programs

Anna Derezińska [0000-0001-8792-203X] and Mateusz Byczkowski

Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
A.Derezinska@ii.pw.edu.pl

Abstract. Utilization of design patterns is supposed to have a considerable impact on software quality and to correlate with different software metrics. Much experimental research has considered these issues but almost none related to C# programs. This paper examines utilization of Gang-of-Four design patterns combined with results of software metrics calculated on a set of C# programs. The design patterns have been automatically detected in source code. Analyzed applications with design patterns evaluated to be more complex but in the same time better maintainable than applications without design patterns. Usage of design patterns contributed to a growth in class encapsulation. Classes that implemented design patterns were more complex that other classes used in both types of applications with and without design patterns. The outcomes could be of importance in software maintenance, reverse engineering and program refactoring.

Keywords: Design Patterns, Software Metrics, Design Pattern Detection, Gang of Four, C#.

1 Introduction

Design Patterns (DP in short) presented by Gang of Four [1] are programming artefacts commonly used in object-oriented software development that could influence software quality, especially its ability to be tested, refactored, maintained, etc. However, the detailed impact of design pattern usage on software quality is still an open question, as ambiguous conclusions have been drawn from many experiments [2,3,4].

Utilization of DP has been analyzed in numerous programming environments [4,5]. Different variants of DP implementation in C# have been presented by Metsker [6] and Sarcar [7], and applied in practice. However, to the best of our knowledge, no research, except of the works published by Gatrell et al. [8,9], has been performed on usage of DP in C# programs. Furthermore, despite a lot of studies considering utilization of DP in relation to various software metrics have been performed [2,3,5], there is a gap in such analysis in the area of C# programs.

This is a pre-print of a contribution published in Engineering in Dependability of Computer Systems and Networks. DepCoS-RELCOMEX 2019, Zamojski W., Mazurkiewicz J., Sugier J., Walkowiak T., Kacprzyk J. (eds) by Springer, Cham. The definite authenticated version is available online via https://doi.org/10.1007/978-3-030-19501-4_13

Cite this paper as:
Derezinska A., Byczkowski M. (2020) Evaluation of Design Pattern Utilization and Software Metrics in C# Programs. In: Zamojski W. et al. (eds) Engineering in Dependability of Computer Systems and Networks. DepCoS-RELCOMEX 2019. Advances in Intelligent Systems and Computing, vol 987. Springer, Cham, pp. 132-142.

We have tried to fill in this gap. Therefore, we have refined detection rules of DP in C# programs [10]. While using an enhanced DP detector integrated with software metric tools we have conducted experiments on 20 C# programs. In this paper we have presented results of software analysis at the class level, comparing classes participating in DP instances (DP classes) with the remaining classes. We have also related this distinction of classes to the outcomes of selected software quality metrics [11].

Our experiments have showed that applications with design patterns seemed to be more complex but in the same time better maintainable than applications without design patterns. Membership of DP was associated with a high encapsulation of classes. Classes that implemented design patterns were more complex that other classes used in both types of applications with and without design patterns.

The reminder of this paper is organized as follows. Next Section describes some related work. Selected software metrics are surveyed in Sec. 3. A brief overview of C# Analyzer is given in Sec. 4. Results of experiments are presented and discussed in Sec. 5. Finally, Sec. 6 concludes the paper.

2 Related Work

Utilization of GoF design patterns and their impact on the software quality have been examined in various studies [2,5]. Evaluation of DP has been performed through two main activities: (1) computing code metrics, (2) using expert opinion (e.g. surveys). In this paper we refer to the first approach.

Gatrell et al. reported on investigation of development of a commercial C# system [8,9]. Different software features in DP-based and other classes were recognized. The authors focused on DP introduced intentionally, mentioned in documentation, and manually detected after code inspection. Classes participating in design patterns were found to be changed more frequently than other classes [8]. Familiarity with those classes to developers was supposed to make them easy to adopt and change. As the highest change prone patterns were observed Adapter, Method, Proxy, Singleton and State. Pattern-based classes were also more fault-prone than non-pattern classes, both in terms of the number of changes made to a class, and the size of the changes required to fix a fault [9]. Adaptor, Method and Singleton patterns were said to be the most fault prone patterns. Coupling of classes had a significant relationship with the fault proneness of classes in the system. Those results are not directly comparable to our work as they did not use any automatic detection of DP and focused on a system evolution while comparing many versions of a program under development.

The majority of other experimental studies on DP usage, combined often with software metrics measurement, considered C++ or Java programs; or referred to a class model level and thus could be considered language independent.

Hussain et al. observed significant correlation between structural complexity of Java programs and usage of Template, Adapter, Command, Singleton and Factory Method design patterns [12].

Gravino and Risi investigated 10 Java tools and founded that DP classes have a greater number of LOC and a greater number of comments, better LCOM (Lack of

Cohesion in Methods) in comparison to other classes, and are more complex in terms of WMC (Weighted Method per class) [13].

The effect of DP on stability has been studied on a sizeable set of Java classes [14]. Classes that play exactly one role in a GoF design pattern were found to be more stable than classes that play zero or more than one role in GoF. However, the statistical results varied across projects from different application domains and different types of patterns.

Coupling and cohesion of classes have also been investigated in a set of Java applications in [15]. Assessment of CBO (Coupling Between Objects) and LCOM metrics showed that DP-related classes are more coupled and less cohesive than the DP non-participant classes.

3 Measurement of Software Metrics for C# Programs

Different software metrics [11] have been considered in the context of programs with DP, while the most often subsets of the CK (Chidamber and Kemerer) and QMOOD (Quality Model of Object Oriented Design) sets of software metrics [12,13, 15,16,17]. In the study discussed in this paper we have used metrics supported by four static analysis tools of C# code [18,19,20,21]. As many of these metrics are similar, we have limited our discussion to the following subset of metrics:

1. LOC - Lines Of Code. Code lines are counted as program statements possible to be executed. Instructions of type “for”, “if”, “while” that create a simple block enclosed by “{“, “}” are counted as one line, “if-else” statements as two lines.
2. DIT - Depth of Inheritance. It returns a depth level in the inheritance tree. All classes inherit from *System.Object* and have at least DIT=1. Interfaces have DIT=0.
3. CBO – Coupling Between Objects (Class Coupling) [11]. It counts how many elements of other classes (methods, attributes) are used by a considered class.
4. CC - Cyclomatic Complexity (McCabe metric) [11]. It gives a structural complexity of program control dependencies (for a program, class or method).
5. MI – Maintainability Index [22]. It estimates a relative effort to maintain a program and takes values from 0 to 100. In Visual Studio it is calculated as:

$$MI = max(0, (171 - 5.2*ln(HalsteadVOL) - 0.23*CC - 16.2*ln(LOC)) * 100/171) \quad (1)$$

Where Halstead Volume [11] is defined as:

$$HalsteadVOL = (OP+OD)*log2(UOP+UOD) \quad (2)$$

OP – the total number of operators, OD - the total number of operands,
UOP – the number of distinct operators, UOD – the number of distinct operands.

6. lLOC – Logical statements Lines of Code – The number of statements ended with semicolon “;”.
7. IC - Interface Complexity. The sum of the number of input parameters and the number of return points.

8. NM - Number of Methods (public, protected, private, total).
9. NA - Number of Attributes (public, protected, private, total).

4 C# Analyzer

Automatic evaluation of DP utilization in C# programs and calculation of software metrics have been the main goals of a prepared tool. The C# Analyzer tool has been implemented as an extension of Visual Studio. It integrates some VS facilities with a program developed for DP detection and external tools for software metric assessment. Analyzer consists of three main modules: a general manager, a software metric manager, and a DP detector (Fig. 1). The general manager collaborates directly with VS and supervises system configuration, user interface, as well as other components. It collects results from other modules and supports their evaluation.

The second module detects DP according to a set of structural rules. It is an enhanced version of the tool implemented by Nagy and Kovari [24]. The main enhancements have referred to recognizing of five additional design patterns and refinement of criteria of pattern detection [10]. The criteria have been adjusted to specific features of the C# programing language. In the current version, the following patterns can be recognized: Singleton, Factory Method, Proxy, Decorator, Composite, Adapter, Mediator, Chain of Responsibility, Observer, and Visitor.

In general, the improvements have resulted in higher number of recognized pattern instances than using the previous tool [10]. It was mainly due to the additional types of pattern detected. However, the more refined criteria have eliminated some of pattern instances accepted formerly, as they have now been counted as false positive.

The third module is devoted to software metric calculation. It calls the tools assigned to the system. The current version cooperates with Visual Studio Code Metrics Powertool [18], which is a built-in VS static analysis facility, and three external tools: SourceMonitor [19], Resource Standard Metrics [20], and LocMetrics [21]. Result data of metrics are further processed by an ETL (Extract, Transform, and Load) tool. The ETL submodule is realized with ETL-Pentaho Data Integration [23]. Given workflows control transformations corresponding to specific software metric tools.

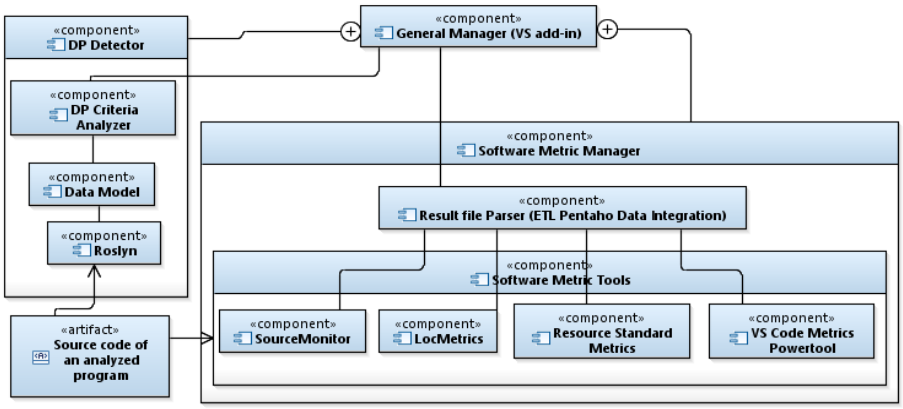

Fig. 1. Architecture of the C# Analyzer.

5 Experiments on Design Patterns and Software Metrics in C#

In this section, we describe the experimental analysis performed on a set of C# programs with support of C# Analyzer, and discuss the results.

5.1 Experiment Subjects

The experiments have been conducted on 20 real programs (**Table 1**). The programs were prepared by various developers in different purposes, not associated with the project under concern. Within this set, 13 programs (P.1, P.2, and P10-P.20) were libraries or other open source applications originated from GitHub resources. Remaining 7 programs (P.3-P.9) were developed by students from our University (WUT) in the context of courses they had attended before or due to other their activities.

Table 1. Programs analyzed in experiments.

Id	Program	Id	Program	Id	Program	Id	Program
P.1	TDDEvaluation	P.6	LuaLinter	P.11	Figures	P.16	Mario Objects
P.2	Log4net	P.7	WPFCalculator	P.12	PacManDuel	P.17	RedDog
P.3	CentralOffice	P.8	SPGenerator	P.13	Cat.Net	P.18	DogeSharp
P.4	AngularCalculator	P.9	BinaryStructure	P.14	Cars	P.19	Play
P.5	HuffmanCoder	P.10	FactoryPattern	P.15	PlainElastic.Net	P.20	Catnap

5.2 Detection of Design Patterns

The subject programs have been analyzed using the DP detector module. The numbers of design pattern instances found in the programs are shown in **Table 2**. The first column lists the patterns that could be detected by the tool. Next columns correspond

to programs in which patterns were detected. The last column gives a sum of the pattern instances in all programs. In the last row, all occurrences of patterns in a given program are summarized.

Entirely, 61 occurrences of design patterns were observed. Only in 8 programs some DP have been detected. In the remaining 12 programs no patterns from the considered DP list were found. The most pattern instances referred to *Factory method* (40). In several cases *Singleton* and *Proxy* were recognized. Only few code extracts were identified as *Decorato*r, *Visitor*, or *Composite*. The remaining four patterns were not detected in any of programs from this set.

Among 8 programs with DP detected, three programs have been prepared by WUT students. Therefore, we could verify the results of automatic detection with intentions of the developers. The authors of programs P.6, P.8, and P.9 endorsed that intentionally developed such design patterns in their programs in the locations indicated by the tool. In this way, we have additional confirmation of correct detection of *Singleton*, *Factory method*, *Decorator* and *Visitor*, and approved DP occurrence in 23 cases.

Table 2. Numbers of design pattern instances detected in programs.

Design Pattern	Programs								Sum
	P.2	P.6	P.8	P.9	P.10	P.13	P.16	P.17	
Singleton	5		1						6
Factory method	13	13	6		2	1		5	40
Proxy	9								9
Decorator	1			2					3
Composite								1	1
Adapter									0
Mediator									0
Chain of respon.									0
Observer									0
Visitor				1			1		2
Sum	28	13	7	3	2	1	1	6	61

5.3 Utilization of Classes with Design Patterns

In comparison to the former tool [24], the current DP detector returns not only information about an occurrence of a given pattern with its main class but also identifies all classes that are members of the detected pattern [10].

In all programs, 2392 classes have been analyzed. It was recognized that 112 classes are members of DP, which corresponded to 4.7% of all classes. While taking into account only projects with DP, classes that are DP members cover 13.5% of all classes. Though, this feature differs noticeably among different projects. Characteristics of programs with DP are shown in **Table 3**. Some programs consist of several projects, and in those cases the results are given separately for member projects denoted

by A), B), C). The numbers of all classes in a project (column *#classes*) are compared with the numbers of classes covered by DP. The next column gives a percentage of those classes. Except of the program P.10, which is small and specially devoted do design pattern usage, classes used in DP covered from a few to twenty-several percent of all project classes. For three projects it was about ¼ of the code, which indicates that it is an important factor in further code evaluation and maintenance.

For comparison, DP-based classes in the C# project examined in [8,9] represented 8.85% of all classes. However, in that case study the most popular patterns turned out to be Singleton and Strategy, while Factory Method was rarely used. DP classes covered from 6 to 34% of all classes in commonly used Java tools according to [13].

The last three columns of **Table 3** give information about depth of inheritance tree, coupling between objects, and maintainability index.

In projects with DP, we compared classes participated in DP with the remaining classes. The data are given in two rows for each project, denoted as DP *yes* or *no* (**Table 4**). The results of the following metrics are shown: Logical Statement Lines of Code - ILOC, Interface Complexity - IC, Cyclomatic Complexity - CC, as well as the number of *public*, *protected*, and *private* methods and attributes, accordingly.

Table 3. Selected metrics of projects with design patterns

Pro-gram	Project	LOC	#classes	# DP classes	% of DP cl.	DIT	CBO	MI
P.2	A) Log4net	7834	294	32	10.9%	5	423	85
	B) Log4net.Tests	2715	67	1	1.5%	5	243	78
P.6	LuaLinter	1347	125	28	22.4%	4	187	86
P.8	A) SPGenerator.Generator	179	38	1	2.6%	3	73	90
	B) SPGenerator.Model	55	11	2	18.2%	3	6	95
	C) SPGenerator.SharePoint	739	29	7	24.1%	3	155	84
P.9	BinaryStructureLib	653	55	7	12.7%	3	77	85
P.10	FactoryPattern	10	7	5	71.4%	1	5	98
P.13	Cat.Net	1034	39	10	25.6%	2	100	85
P.16	MarioObjects	3381	72	3	4.2%	7	186	91
P.17	A) RedDog.ServiceBus	349	32	4	12.5%	3	88	85
	B) RedDog.Messanger	602	64	12	18.8%	3	136	89
Sum		18898	833	112	13.5%			

Table 4. Mean metrics of classes with and without DP in projects using DP (DP classes>10%)

Project	DP	ILOC	IC	CC	Number of methods			Number of attributes		
					public	prot.	priv.	public	prot.	priv.
P.2.A)	yes	28.3	11.0	13.7	3.6	0.8	1.0	0.3	0.1	3.5
	no	20.1	10.2	9.5	2.8	0.7	0.9	0.3	0.1	2.0
P.6	yes	6.3	4.6	1.7	1.4	0.0	0.0	0.0	0.0	1.2
	no	4.5	4.2	2.7	1.3	0.0	0.3	0.0	0.1	1.1

P8.B)	yes	6.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
	no	4.1	1.1	0.4	0.4	0.0	0.0	0.4	0.0	0.1
P8.C)	yes	25.3	16.0	13.3	2.4	2.1	0.7	0.7	0.7	1.7
	no	18.3	10.0	6.4	1.6	0.6	1.1	0.3	0.0	3.4
P.9	yes	8.57	5.4	3.7	2.0	0.0	0.9	0.4	0.0	0.4
	no	9.48	4.1	3.9	2.0	0.0	0.6	0.2	0.0	2.8
P.10	yes	1.0	0.8	0.8	0.8	0.0	0.2	0.0	0.0	0.0
	no	2.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.5
P.13	yes	35.2	9.7	8.6	2.2	0.7	0.7	0.0	0.0	1.6
	no	22.1	7.9	7.8	2.9	0.0	1.3	0.0	0.0	1.6
P.17.A)	yes	9.5	5.8	2.5	2.5	0.0	0.5	0.0	0.0	1.8
	no	8.1	9.6	4.1	1.6	0.1	0.5	0.3	0.0	1.0
P.17.B)	yes	6.2	6.8	3.8	3.2	0.0	0.7	0.0	0.0	0.0
	no	6.9	7.0	4.5	1.5	0.2	0.9	0.1	0.0	0.3

Summary of mean metrics calculated over three sets of classes are given in **Table 5**. The selected metrics are the same as in **Table 4**. Two upper rows refer to projects with DP. They include mean values for classes that cover DP (*Yes* in the DP column) and other classes from the same projects (*No*). The last row contains values for classes not covering DP from all kinds of projects considered in the experiments (*No-all*).

Table 5. Comparison of mean metrics of sets of classes

DP	#Class	ILOC	IC	CC	Methods			Attributes		
					public	prot.	priv.	public	prot.	priv.
Yes	112	**16.7**	**7.8**	**7.0**	2.5	0.44	0.6	0.3	0.06	1.7
No	721	**18.7**	**9.1**	7.1	2.8	0.30	1.2	0.7	0.03	1.6
No-all	2280	17.6	7.3	**5.4**	2.9	0.29	0.7	0.5	0.04	2.1

5.4 Discussion of Results of Metrics

Comparison of DP classes with remaining classes within the same projects (Table 4) does not indicate on unambiguous impact of DP on classes covered by them.

If we compare all DP classes (the first row in Table 5), with no DP classes of all considered programs (the last row), it could be observed that the cyclomatic complexity (CC) is higher in the first case. This could imply higher complexity of DP classes. However, comparison of classes within DP projects does not confirm the fact, as CC metrics are almost the same. We can only learn that in general projects with DP were more complex than projects without DP. Furthermore, in some projects, CC is higher for DP classes while in others vice versa (Table 4). This fact could be correlated with the number and types of DP applied in the programs. Though, this would have required more DP occurrences than those encountering in the programs of concern.

A situation is different when we compare the mean numbers of instructions (ILOC) and of interface complexity (IC). In these cases, DP and no DP classes within the DP projects have higher discrepancy, than DP classes compared to all no DP classes (Table 5). The DP classes have less instructions (2 on average) and lower interface complexity (about 1.3) than the remaining classes. However those discrepancies are not

high, and furthermore if we look at separate programs (Table 4), we can find different outcomes both for higher and lower such values.

Private and protected attributes constitute 85.6% of all attributes in classes with DP, while 70.3% in the remaining classes of the same set of projects. These results show that DP classes are more encapsulated than the remaining classes within the projects, which is consistent with the paradigm of object-oriented programming.

In general, usage of DP is associated with implementation a slightly more complex structures, in which a comparable amount of instructions is used preserving a similar interface complexity and encapsulation. This founding of DP class complexity has agreed with those of Java projects [13] although assessed with different metrics.

Maintainability Index, which takes 100 as maximum, is considerably high for all DP projects (**Table 3**). The average MI value of all DP projects equals to 87 and is higher than of projects without DP (80), testifying better ability of maintenance.

5.5 Threats to Validity

An external threat to validity is lack of guarantee that the results of the programs could be generalize. To alleviate this problem, we chose 20 subject projects from real software, written by different authors and from different domains. However, experiments on bigger sets could be valuable. Construct validity has been associated with measurements of software metrics and detection of DP. Metrics have been measured by commonly used tools; many of metrics have similar outcomes from different tools, hence their values could have been verified. Design patterns have been recognized according to the structural rules implemented in the DP detector [10]. The study was mainly bound by the set of 9 DP supported by the tool. Within the set of programs, no detected patterns were found to be classified by mistake. Analysis of some programs was confirmed by their authors in terms of all DP occurrences and DP types.

6 Conclusions

Development of the C# Analyzer tool, especially its DP detection module, has allowed us to conduct experiments on DP utilization in C# programs. As expected, usage of DP seems to have a positive impact on the code quality, although it could also have been influenced by other factors, including developer skills. According to project-level analysis, DP increase maintainability of software. Class-level analysis of programs with DP approved high encapsulation of DP code. However, it should be noted, that the detailed results might vary among different projects.

In general, differences between code with and without DP were more visible at the project level than at the class level. One of possible reasons could be the fact that the projects were developed by various programmers. Those developers who knew DP and applied them might have been able to implement high quality code that would be easy to comprehend and maintain.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston (1995).
2. Ali, M., Elish, M. O.: A comparative literature survey of design patterns impact on software quality, In: Proc. of International Conference on Information Science and Applications (ICISA), pp. 1-7. (2013). doi:10.1109/ICISA.2013.6579460
3. Ampatzoglou, A., Charalampidou, S., Stamelos, I,: Research state of the art on gof design patterns: a mapping study. Journal of Systems and Software 86(7), 1945–1964 (2013). doi: 10.1016/j.jss.2013.03.063
4. Khomh, F., Gueheneuc,Y-G.: Do design patterns impact software quality positively? In: 12th European Conference on Software Maintenance and Reengineering, pp. 274-278. IEEE Comp. Soc. (2008).doi: 10.1109/CSMR.2008.4493325
5. Mayvan, B. B., Rasoolzadegan, A., Yazdi, Z. G.: The state of the art on design patterns: a systematic mapping of the literature, J. of Systems and Software 125, 93-118 (2017). doi: 10.1016/j.jss.2016.11.030
6. Metsker, S.J.: Design patterns in C#, Addison-Wesley, Boston (2004).
7. Sarcar, V.: Design patterns in C#. Apress, Berkeley, CA (2018). doi: 10.1007/978-1-4842-3640-6
8. Gatrell, M., Counsell, S., Hall, T.: Design patterns and change proneness: a replication using proprietary C# software. In: 16th Working Conference on Reverse Engineering (WCRE), pp. 160-164, IEEE Comp. Soc. (2009). doi: 10.1109/WCRE.2009.31
9. Gatrell, M., Counsell, S.: Faults and their relationship to implemented patterns coupling and cohesion in commercial C# software. International Journal of Information System Modeling and Design, 3(2), pp. 69-88, (2012). doi: 10.4018/jismd.2012040103
10. Derezińska, A., Byczkowski, M: Enhancements of detecting Gang-of-Four design patterns in C# programs. In: L. Borzemski et al. (Eds.): ISAT2018, AISC 852, pp. 277-286. Springer,Cham (2019). doi: 10.1007/978-3-319-99981-4_26
11. Kan, S. H.: Metrics and models in software quality engineering. 2nd edn. Addison-Wesley Professional (2002).
12. Hussain, S., Keung, J., Khan, A.A., Bennin, K.E.: Correlation between the frequent use of Gang-of-Four design patterns and structural complexity. In: 24th Asia-Pacific Software Engineering Conference, pp. 189-198. (2017). doi: 10.1109/APSEC.2017.25
13. Gravino, C., Risi, M.: How the use of design patterns affects the quality of software systems: a preliminary investigation. In: 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 274-277. IEEE Comp. Soc. (2017). doi: 10.1109/SEAA.2017.32
14. Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., Averiou, P.: The effect of GoF design patterns on stability: a case study. IEEE Transaction on Software Engineering. 41(8), 781-802 (2015). doi: 10.1109/TSE.2015.2414917
15. Mohammed, M., Elish, M., Qusef, A.: Empirical insight into the context of design patterns: modularity analysis. In: 7th International Conference on Computer Science and Information Technology (CSIT), pp. 1-6. IEEE (2016). doi: 10.1109/CSIT.2016.7549474
16. Hsueh, N-L., Chu, P-H., Chu, W.: A quantitative approach for evaluating the quality of design patterns. J. of Systems and Software, 81(8), 1430–1439 (2008). doi:10.1016/j.jss.2007.11.724.
17. Derezińska, A.: Metrics in software development and evolution with design patterns. In: Silhavy R. (eds) CSOC2018, AISC, vol. 763, pp. 356-366. Springer, Cham (2019).

18. VS Docs: Code metrics values, https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2017, last accessed 2019/01/14.
19. SourceMonitor, http://www.campwoodsw.com/sourcemonitor.html, last accessed 2019/01/14.
20. Resource Standard Metrics (RSM), http://www.msquaredtechnologies.com/base/index.html, last accessed 2019/01/14.
21. LocMetrics, http://www.locmetrics.com, last accessed 2019/01/14.
22. Maintainability Index Range and Meaning, https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/, last accessed 2019/03/06.
23. Etl-pentaho-data-integration, https://www.etltool.com/etl-vendors/pentaho-data-integration-kettle/, last accessed 2019/01/14.
24. Nagy, A., Kovari, B.: Programming language neutral design pattern detection, In: Proc. of 16th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), pp. 215-219, Nov. (2015). doi:10.1109/CINTI.2015.7382925