ICIC Express Letters
Part B: Applications
Volume **12**, Number **12**, December **2021**

ICIC International ©2021 ISSN 2185-2766
pp. 1111–1119

THE IMPACT OF DESIGN PATTERNS ON SOFTWARE MAINTAINABILITY AND UNDERSTANDABILITY: A METRICS-BASED APPROACH

Mohammed Ghazi Al-Obeidallah

Faculty of Computer Sciences and Informatics
Amman Arab University
P.O. Box 2234, Amman 11953, Jordan
m.obeidallah@aau.edu.jo

Received May 2021; accepted July 2021

Abstract. *The impact of design patterns on software quality attributes provides a support for decision-making during software design and refactoring. This article presents a metrics-based approach to address the impact of design pattern instances on software maintainability and understandability. This approach classifies system classes into two groups: classes that are playing roles in design patterns (pattern classes) and classes that are not playing roles in design patterns (non-pattern classes). Size, coupling and inheritance metrics were calculated for both groups using JHawk, a Java metrics tool. Furthermore, the participation percentage (for both groups) in the whole metric value was calculated. The correlation of metrics to software maintainability and understandability presented by previous research studies has been used. The experiment results illustrate that design pattern classes have fewer roles in size and inheritance metrics than do non-pattern classes, a sign that design pattern classes enhance software understandability and maintainability.*

Keywords: Design patterns, Software quality, Maintainability, Understandability, Software metrics, Gang of four, Design templates

1. **Introduction.** Design patterns are the focus of many works studying their relevance, visualization and identification, with the hypothesis that their use improves quality. Gamma et al. – henceforth GoF – claim in the preface of their book: "You will have insight that makes your own designs more flexible, modular, reusable and understandable" [1].

GoF describes through discussions how design patterns support adaptability and are expected to promote software evolution; they ease maintenance tasks by explicitly identifying class roles and by localizing where extensions and change should occur [1]. However, the authors did not validate empirically the benefits to software development projects. One benefit, for example, is that design patterns promote adaptability by supporting modifications through specialization. Developers can adapt a system built using these patterns by creating new concrete classes with desired functionality rather than by direct modifications to existing classes. However, design patterns usually lead to an increased number of software artifacts, such as classes, associations and delegations, which increase the static complexity of a software system. This article aims to address whether the classes playing roles in design patterns have better software metrics than do other classes in the system. Higher values of certain software metrics indicate good quality (e.g., cohesion). In contrast, higher values of other software metrics indicate bad quality (e.g., coupling between objects). This will provide an indication of how the implementation of design pattern instances affects the quality of a subject system. The focus of previous research studies was on the detection of design patterns [2]. Some studies claim that the implementation of certain design patterns has a positive impact on the quality of

DOI: 10.24507/icicelb.12.12.1111

software systems. In contrast, other studies claim that the implementation of the same design patterns has a negative impact on software quality.

The systematic literature review presented by Wedyan and Abufakher shows that documentation of patterns, size of pattern classes, and the scattering degree of patterns have clear impact on quality [3]. The study presented by Kermansaravi et al. addressed the bidirectional mutations between design patterns and design anti-patterns and the impacts of these mutations on software change- and fault-proneness. The results show that some mutations of design anti-patterns and design patterns are faultier in specific contexts [4]. Laosen et al. [5] presented an automatic approach for ranking and recommending GoF patterns where design-pattern vectors, representing the GoF patterns in terms of the problem types they address, are constructed based on the design pattern intent ontology (DPIO) developed by Kampffmeyer. The study presented by Khomh and Guéhéneuc in [6] has identified seven main themes of design patterns in software development: forward engineering, reverse engineering, documentation, knowledge sharing, development tools, formalization and quality. The authors argued that the research community should follow examples set in other research fields and systematically define, relate, categorize and classify design patterns.

Gravino and Risi presented a study which followed a similar approach to that we are introducing here in this article [7]. They analyzed at the class level the quality of software portions of ten software systems including classes participating in design patterns instances with respect to the remaining software portions. The study claimed that the use of design patterns impacts the quality of the software. However, the study of Gravino and Risi [7] was limited to the CK (Chidamber and Kermerer) measure values and there were no correlations between these metrics and the quality of the subject systems [8].

Scanniello et al. studied the importance of documenting design pattern instances and how it can affect the comprehensibility of source code [9]. The results showed that documenting design patterns yields an improvement in correctness of understanding source code for participants who have an adequate level of experience.

Ampatzoglou et al. presented a case study to investigate the stability of classes that participate in instances of GoF design patterns [10]. They examined whether the stability of these classes is affected by the pattern type, the role that the class plays in the pattern, the number of pattern occurrences in which the class participates, and the application domain. The results suggested that classes that participate in coupled pattern occurrences appear to be the least stable.

The remaining of this article is organized as follows. Section Two outlines a methodology to address the impact of design patterns, using software metrics and design pattern occurrences, on software maintainability and understandability. The experiments and results are presented in Section Three. Finally, conclusions are summarized in Section Four.

2. **Methodological Approach.**

There is no formal theory that links design patterns to software quality concepts. However, it has been claimed that the use of design patterns provides several advantages, such as increased reusability and improved maintainability and comprehensibility of existing systems.

The methodology presented in this article relies on software metrics, which are useful measurements for characterizing software systems, in an attempt to assess the impact of design patterns on software maintainability and understandability. Specifically, software design metrics for all system classes, at the class level, will be calculated to investigate whether a safe conclusion can be drawn regarding the impact of design patterns on software maintainability and understandability. To calculate the required metrics, a Java metrics tool named JHawk has been used. We used the latest version of JHawk, v6.1.3, under the academic license granted from virtual machinery [11].

2.1. **A metrics-based approach.** Figure 1 presents a metrics-based approach to assess the impact of design pattern instances on software quality attributes. As mentioned above, all metrics will be calculated using JHawk at the class level. The steps of the metrics-based approach could be summarized as follows.

- Recover design pattern instances from each subject system.
- For each design instance, determine its participant classes.
- Classify system classes into two groups: classes that play roles in design patterns (henceforth pattern classes) and classes that do not (henceforth non-pattern classes).
- Calculate size, coupling and inheritance metrics for all system's classes.
- For each class in the system, calculate the percentage of participation in the whole system metric value.
- Calculate the percentage of participation in the whole metric value for both groups: pattern classes and non-pattern classes.
- Correlate software metrics to quality attributes using previous research studies.

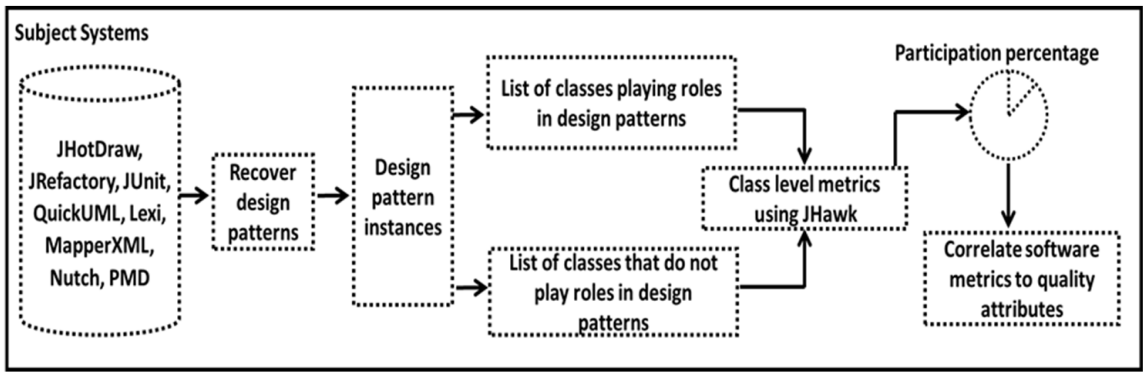

FIGURE 1. A metrics-based approach to assess the impact of design patterns

The input of the metrics-based approach is the source code of the subject system, whereas the output is the percentage of participation for both groups in the whole metric value. The participation percentage of each individual class is calculated based on the whole system metric value. For example, if the whole system coupling, calculated by JHawk, was 1396 and a class has a coupling value of 65, then its participation percentage in the whole system coupling (whole metric value) could be calculated as $(65/1396)\times100\% = 4.66\%$. Hence, the participation percentage for all classes in the system could be calculated in the same manner.

Based on a validated set of design pattern instances recovered using our research prototype MLDA (a **M**ultiple **L**evel **D**etection **A**pproach for design patterns recovery) [12-14], all classes participating in design patterns were determined in eight subject systems. More specifically, we referred to the all publicly published results in the available literature to generate a list of design pattern instances implemented in subject systems. We used the repository of Perceron [15], the design pattern detection tools benchmark platform [16] and P-MARt [17] as the main benchmarks for validating the recovered design instances using MLDA. Hence, the presented list of design pattern instances for all subject systems was validated based on the common agreement in the available literature.

A design pattern instance should reflect the required pattern structure and behaviour presented by the GoF [1]. Pattern's participant classes form a design pattern instance or occurrence. For example, classes Adapter, Adaptee and Target (grouped together) form one instance of the Adapter design pattern. The investigated systems are JHotDraw, JRefactory, JUnit, QuickUML, Lexi, MapperXML, Nutch, and PMD. Furthermore, the following software metrics will be calculated [8,18]: Number of Methods (NOM), Lack of Cohesion of Methods (LCOM), Total Response for Class (RFC), Coupling between

Objects (CBO), Total Lines of Code (LOC), Fan-IN (F-IN), Depth of Inheritance Tree (DIT), Number of Children (NOC), Cohesion (COH), and Fan-OUT (F-OUT). The selection of these metrics was made since they have a key role in characterizing the quality of software systems. More specifically, these metrics reflect the key aspects of any object-oriented program, i.e., size, coupling and inheritance [18,19].

2.2. **Correlation of software metrics to quality attributes.** The external behaviour of software systems can be recognized based on its internal metrics. Several studies were presented in the literature to correlate the impact of calculated software metrics with quality attributes. This correlation was made based on certain statistical analysis and experiments. The impact can be positive or negative, or there can be no impact at all. We are trying to address the impact of design patterns on software understandability and maintainability since they are the most commonly investigated quality attributes. Table 1 illustrates the correlated impact of NOM, LOC, RFC, CBO, LCOM, COH, F-IN, F-OUT, DIT and NOC on software understandability and maintainability.

TABLE 1. The correlation between software metrics and software understandability and maintainability

Software quality/metrics		**Understandability**	**Maintainability**
Size	NOM	▼	▼
	LOC	▼	▼
Coupling	RFC	▼	▼
	CBO	▼	▼
	LCOM	▼	▼
	COH	▲	▲
	F-IN	▼	▼
	F-OUT	◄ ►	◄ ►
Inheritance	DIT	▼	▼
	NOC	▼	▼
▲ Positive impact	▼ Negative impact	◄ ► No impact	

This correlation was made based on the correlation presented by [20]. This study was selected since it has significant correlation levels. However, to the best of our knowledge, there are no studies that contradict the reported impact presented by the selected study. The positive impact (▲) of a metric indicates that high values of that metric are desirable. On the other hand, a metric's negative impact (▼) indicates that high values of that metric are not desirable. Table 1 demonstrates that most of the selected metrics have a negative impact on software maintainability and understandability (i.e., high values of these metrics are a sign of bad quality). The cohesion metric has been reported to have a positive impact on understandability and maintainability efforts, whereas F-OUT has no impact at all.

3. **Results and Discussion.** All the experiments have been run on with Intel Core i5-2400 CPU. JHawk calculated the required software metrics for all subject systems. The calculation process was quite fast where JHawk spent only a few seconds to generate the results. Furthermore, based on a validated set of design pattern instances recovered using our research prototype, MLDA, two sets of classes were created: pattern classes and non-pattern classes.

3.1. **Recovered design pattern instances.** Table 2 presents the number of design pattern instances implemented in all subject systems, as they are recovered by MLDA and as they are validated based on all public published results in the available literature (i.e., the number of implemented design instances in a subject system is the number of true positive instances plus the number of false negative instances). Hence, all classes that are playing roles in design patterns were identified. As Table 2 demonstrates, 1051 design pattern instances were implemented in all subject systems.

Table 2. Total number of design pattern instances implemented in subject systems

Type	**Design pattern**	**JHotDraw**	**JRefactory**	**JUnit**	**QuickUML**	**Lexi**	**Nutch**	**PMD**	**MapperXML**
Creational patterns	Singleton	2	12	0	1	2	2	4	3
	Prototype	2	0	0	0	1	0	1	0
	Abstract factory	0	0	0	1	0	0	2	2
	Factory method	2	87	2	18	0	5	32	22
	Builder	0	2	0	1	0	0	16	1
Structural patterns	Adapter	11	16	11	29	30	44	87	41
	Bridge	4	0	0	1	0	0	1	0
	Composite	1	0	1	1	3	0	0	1
	Decorator	3	1	1	3	0	4	0	0
	Façade	1	2	0	1	1	1	1	1
	Flyweight	1	0	0	1	1	1	0	1
	Proxy	0	0	0	2	0	4	4	0
Behavioral patterns	COR	0	0	0	0	0	0	1	0
	Command	9	25	0	18	6	25	3	26
	Interpreter	0	0	0	0	0	0	0	0
	Iterator	0	0	1	0	0	1	0	0
	Mediator	0	0	0	0	0	0	0	0
	Memento	0	0	0	0	10	1	1	12
	Observer	2	0	1	1	0	0	1	3
	State/Strategy	6	11	3	10	1	113	21	11
	Visitor	2	2	0	0	0	0	1	0
	Template method	4	4	1	3	0	7	108	56
Total		50	162	21	91	55	208	284	180

3.2. **Participation percentage in the whole metric value.** After the identification of all design pattern instances in all subject systems, the number of classes playing roles in design patterns, the number of classes playing roles in structural, creational and behavioural instances and the number of classes playing more than one role have been identified.

Table 3 presents the total number of classes playing roles in design patterns for all subject systems. A class may play more than one role and participant in two or more different design patterns. The percentage of design patterns in a subject system is the percentage of classes playing roles in design patterns to the total system classes. Around 40% of the subject systems' classes participate and play roles in design patterns.

Table 4 illustrates the average percentage of participation in the whole metric value for both pattern classes and non-pattern classes in all subject systems. Consistent behaviour

TABLE 3. The number of classes playing roles in design patterns in all subject systems

Number of classes/subject systems	**JHotDraw**	**JRefactory**	**JUnit**	**QuickUML**	**Lexi**	**Nutch**	**PMD**	**MapperXML**
Total number of classes	201	612	112	145	170	398	570	374
NCPR in creational instances	8	122	3	37	24	12	83	52
NCPR in structural instances	35	35	30	66	43	100	112	79
NCPR in behavioral instances	30	80	10	39	20	82	155	114
Number of classes playing more than one role	17	48	5	37	25	54	133	76
NCPR in all design pattern instances	56	189	38	105	62	140	217	169
Percentage of design pattern classes to the total system classes	28%	31%	34%	72%	36%	35%	38%	45%
Note. NCPR: Number of classes playing roles								

can be noticed for size and inheritance metrics. These metrics negatively affect the maintainability and understandability of a subject system. More specifically, the calculated averages for size and inheritance metrics indicate that the participation of pattern classes, in the whole metric value, is less than that of the other classes in the system. Hence, total system size and inheritance were shaped based on the non-pattern classes. On the other hand, the average participation of the pattern classes in the whole coupling metrics, except RFC, is higher than that of the other classes in the system. Both groups participate almost equally in the average RFC metric. The functionality of the system might be relying on the design pattern classes (i.e., design pattern classes provide key functionality to the system) which require interacting and collaboration between pattern classes and other classes in the system. This could explain why the participation of design pattern classes, in the whole system coupling, is higher than that of other classes in the system. Furthermore, whole system cohesion was formulated based on the non-pattern classes, which contradicts the common belief that the implementation of design pattern enhances system cohesion. Consequently, the participation of pattern classes in five out of nine metrics is less than that of other classes in the system. These metrics have a negative impact on software maintainability and understandability. The pattern participant classes provide the key functionality to the system, which may explain why these classes tend to couple and interact with other classes.

In addition, the whole system cohesion was formulated based on the non-pattern classes. High value of cohesion is desirable. Non-pattern classes implement more methods than do pattern classes in all subject systems, except in PMD and MapperXML. Pattern classes of PMD and MapperXML have RFC, CBO and F-IN higher than non-pattern classes do. This could be explained by investigating the implemented design instances in these two systems where most of the implemented instances are behavioural instances.

Furthermore, in most subject systems, pattern classes have fewer cohesion values than do non-pattern classes. Consequently, whole system cohesion relies on the non-pattern classes. This is a sign of the improper use of pattern classes where they perform more than a single purpose function. The inheritance metrics, depth of inheritance tree and number of children of pattern classes for all subject systems are all fewer than those of non-pattern classes. Hence, pattern classes require less maintainability and understandability effort. Pattern classes of F-OUT metric, which has been reported to have no impact on software

TABLE 4. The average participation percentage in the whole metric value for both classes groups in all subject systems

Number of classes/subject systems		**NOM**	**LCOM**	**RFC**	**CBO**	**NLOC**	**F-IN**	**DIT**	**COH**	**F-OUT**	**NOC**
JHotDraw	CR-Pattern classes %	31	11	22	26	30	33	15	25	15	14
	SR-Pattern classes %	9	3	15	8	4	11	7	2	12	4
	BE-Pattern classes %	12	11	18	16	12	12	10	11	17	11
	Non-pattern classes %	48	75	45	50	54	44	68	62	56	71
JRefactory	CR-Pattern classes %	15	23	18	18	17	15	15	18	21	15
	SR-Pattern classes %	8	3	5	4	5	14	1	6	2	4
	BE-Pattern classes %	11	8	12	14	13	11	7	9	9	5
	Non-pattern classes %	66	66	65	64	65	60	77	67	68	76
Junit	CR-Pattern classes %	14	15	14	20	10	35	8	15	15	10
	SR-Pattern classes %	6	3	4	11	3	6	3	6	8	2
	BE-Pattern classes %	6	6	8	11	10	19	3	11	5	1
	Non-pattern classes %	74	76	74	58	77	40	86	68	72	87
QuickUML	CR-Pattern classes %	21	11	23	31	22	38	18	20	30	44
	SR-Pattern classes %	6	8	7	5	14	16	6	2	13	10
	BE-Pattern classes %	22	7	20	24	12	12	7	10	8	15
	Non-pattern classes %	51	74	50	40	52	34	69	68	49	31
Lexi	CR-Pattern classes %	33	15	21	22	21	41	8	3	16	30
	SR-Pattern classes %	11	1	11	14	12	14	1	2	5	8
	BE-Pattern classes %	11	5	13	10	8	11	3	3	4	12
	Non-pattern classes %	45	79	55	54	59	34	88	92	75	50
Nutch	CR-Pattern classes %	21	9	22	22	30	41	9	12	22	NA
	SR-Pattern classes %	10	1	11	10	1	16	3	2	11	NA
	BE-Pattern classes %	10	6	7	23	7	9	9	7	10	NA
	Non-pattern classes %	59	84	60	45	62	34	79	79	57	NA
PMD	CR-Pattern classes %	55	24	44	26	38	50	31	13	45	15
	SR-Pattern classes %	2	2	1	21	8	8	1	12	3	3
	BE-Pattern classes %	13	9	21	20	21	22	12	10	12	9
	Non-pattern classes %	30	65	34	33	33	20	56	65	40	73
MapperXML	CR-Pattern classes %	30	35	34	40	32	30	33	33	35	11
	SR-Pattern classes %	15	9	4	13	4	21	9	3	2	5
	BE-Pattern classes %	21	22	30	22	30	25	14	20	39	3
	Non-pattern classes %	34	34	32	25	34	24	44	44	24	81

CR-Pattern classes: classes that play roles in creational design pattern instances;
SR-Pattern classes: classes that play roles in structural design pattern instances;
BE-Pattern classes: classes that play roles in behavioral design pattern instances;
Non-patterns classes: classes that do not play roles in design patterns.

maintainability and understandability, have fewer metric values than non-pattern classes. Consistent with other coupling metrics, pattern classes of PMD and MapperXML have fewer F-OUT metric values than other classes in the system since most of their design instances are behavioural instances. We noticed that whole system coupling and cohesion was directly affected by the implementation of behavioural design instances. Classes playing roles and participating in behavioural patterns have higher coupling and lower cohesion than do other classes in the system.

4. **Conclusion.** A metrics-based approach has been presented in an attempt to address the impact of design pattern instances on software maintainability and understandability. The metrics-based approach could not reach a safe conclusion regarding the impact of design patterns on software understandability and maintainability. However, the metrics-based approach shows that classes that play roles in design patterns have better inheritance and size metrics than do non-pattern classes. This gives a sign that design patterns enhance software understandability and maintainability. The whole system inheritance and size metrics rely on the pattern classes. In contrast, non-pattern classes have better coupling metrics than do pattern classes where the whole system coupling relies on the pattern classes.

REFERENCES

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

[2] M. G. Al-Obeidallah, M. Petridis and S. Kapetanakis, A survey on design pattern detection approaches, *International Journal of Software Engineering (IJSE)*, vol.7, no.3, pp.41-59, 2016.

[3] F. Wedyan and S. Abufakher, Impact of design patterns on software quality: A systematic literature review, *IET Software*, vol.14, no.1, DOI: 10.1049/iet-sen.2018.5446, 2020.

[4] Z. A. Kermansaravi, M. S. Rahman, F. Khomh, F. Jaafar and Y.-G. Guéhéneuc, Investigating design anti-pattern and design pattern mutations and their change- and fault-proneness, *Empirical Software Engineering*, vol.26, no.9, DOI: 10.1007/s10664-020-09900-0, 2021.

[5] N. Laosen, C. Bou and E. Nantajeewarawat, Automatic recommendation of design patterns based on patterns' intent, *International Journal of Innovative Computing, Information and Control*, vol.16, no.4, pp.1147-1163, 2020.

[6] F. Khomh and Y.-G. Guéhéneuc, Design patterns impact on software quality: Where are the theories?, *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[7] C. Gravino and M. Risi, How the use of design patterns affects the quality of software systems: A preliminary investigation, *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2017.

[8] S. R. Chidamber and C. F. Kemerer, A metrics suite for object oriented design, *IEEE Trans. Software Engineering*, vol.20, no.6, pp.476-493, 1994.

[9] G. Scanniello et al., Documenting design-pattern instances: A family of experiments on source-code comprehensibility, *Trans. Software Engineering and Methodology (TOSEM)*, vol.24, no.3, pp.1-35, 2015.

[10] A. Ampatzoglou et al., The effect of GoF design patterns on stability: A case study, *IEEE Trans. Software Engineering*, vol.41, no.8, pp.781-802, 2015.

[11] V. Virtual_Machinery, *JHawk – Java Code Quality Management – By Fact!*, http://www.virtualmachinery.com/jhawkprod.htm, Accessed in January 2021.

[12] M. G. Al-Obeidallah, M. Petridis and S. Kapetanakis, A multiple phases approach for design patterns recovery based on structural and method signature features, *International Journal of Software Innovation (IJSI)*, vol.6, no.3, pp.36-52, 2018.

[13] M. Al-Obeidallah, *A Multiple Level Detection Approach for Design Patterns Recovery from Object-Oriented Programs*, Ph.D. Thesis, University of Brighton, 2018.

[14] M. G. Al-Obeidallah, M. Petridis and S. Kapetanakis, A structural rule-based approach for design patterns recovery, in *Software Engineering Research, Management and Applications. SERA 2017. Studies in Computational Intelligence*, R. Lee (ed.), Cham, Springer, 2017.

[15] A. Ampatzoglou, O. Michou and I. Stamelos, Building and mining a repository of design pattern instances: Practical and research benefits, *Entertainment Computing*, vol.4, no.2, pp.131-142, 2013.

[16] F. A. Fontana, A. Caracciolo and M. Zanoni, DPB: A benchmark for design pattern detection tools, *2012 16th European Conference on Software Maintenance and Reengineering*, pp.235-244, DOI: 10.1109/CSMR.2012.32, 2012.

[17] Y.-G. Guéhéneuc, D. L. Huynh, G. Straw et al., P-MARt: Pattern-like micro architecture repository, *Proc. of the 1st EuroPLoP Focus Group on Pattern Repositories*, 2007.

[18] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*, Prentice-Hall, Inc., 1994.

[19] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, Springer Science & Business Media, 2007.

[20] F. Dandashi, A method for assessing the reusability of object-oriented code using a validated set of automated measurements, *Proc. of the 2002 ACM Symposium on Applied Computing*, 2002.