Applied Computer Science, vol. 15, no. 1, pp. 5–17
doi:10.23743/acs-2019-01

Submitted: 2018-12-07
Revised: 2019-02-04
Accepted: 2018-03-18

design patterns, software quality, quality assessment

Rafał WOJSZCZYK [0000-0003-4305-7253]*

VERIFICATION OF ACCURACY AND COST OF USE METHODS OF QUALITY ASSESSMENT OF IMPLEMENTATION OF DESIGN PATTERNS

Abstract

Professional programmers use many additional tools over the Integrated Development Environment during their work. Very often they are looking for new solutions, while expecting that the new tool will provide accurate results, and the cost of use will fit within the planned budget. The aim of the article is to present the results of two comparative analyzes carried out in terms of accuracy and the cost of using the quality assessment method of implementation of design patterns.

1. INTRODUCTION

A programmer is a unique craftsman because the products he produces are made using tools created by other programmers or sometimes by himself. This allows programmers to create new, unique solutions, often non-standardized. Ultimately, this leads to the creation of new tools that support the work of programmers. Examples of such solutions are design patterns. Patterns from (Gamma, Helm, Johnson & Vlissides, 1994) have been known for many years, although these are still the same patterns that their implementation is constantly changing.

A programmer implementing design patterns does so on the sample templates from (Gamma, Helm, Johnson & Vlissides, 1994; Metsker, S. J., 2004) and his own knowledge, during which he usually focuses on achieving the purpose

* Koszalin University of Technology, Department of Computer Science and Management, Śniadeckich 2, 75-453 Koszalin, Poland, rafal.wojszczyk@tu.koszalin.pl

of the pattern (solving the programming problem). The implementation of the pattern goal in accordance with the template from (Gamma, Helm, Johnson & Vlissides, 1994) does not mean a beneficial implementation, because each computer program is different. The preferred implementation of the template is a fragment of the source code that meets additional expectations, otherwise it provides benefits in selected criteria. Assuming the low development and integration cost criterion, this means that the template code will not require additional modi-fications when expanding and integrating with this code. Therefore the cost of the development will consist of the cost of adding new parts of the code that use existing pattern implementation. In this context, a programmer working in an agile team after doing his job (writing the source code, usually without complete documentation) is looking for the answer to the question: *will the implement-tation of a given design pattern provide the benefits expected from this pattern?* The method that supports the answer to this question should be accurate and at the same time cheap to use. Well-known software quality models are too imprecise for this purpose, or generally do not take into account design patterns. However, the methods analyzing the implementation of design patterns are often too expensive to use (especially in Agile teams, where the amount of documentation is limited). The aim of the article is to present the results of the verification of the method, which allows the answer to the above question, and at the same time meets the imposed restrictions on accuracy and cost.

The second chapter explains what the quality of pattern implementation is and presents selected related works. The third chapter contains comparative analyzes and results. Fourth chapter shows the results of the use of the method in the production environment, it means practical use. The last chapter is a summary of the article.

2. QUALITY OF IMPLEMENTATION OF DESIGN PATTERNS AND ALTERNATIVE METHODS

2.1. Quality of pattern implementation

The criteria of the assessment of quality in terms of the cost of development and software integration are one of the most important for the vendor. The vendor, who constantly keeps and develops his product, even for many years, should take care of the fact that the cost of running and development are as low as possible. For that purpose design patterns are used. It has been widely accepted, that programmers are implementing patterns on the second level of quality, i.e. so that the implementation meets only the presented aim of the pattern, e.g. one instance of the object in the Singleton pattern. First level of implementation quality is undesirable, such an implementation contains errors, e.g. the public constructor of the class of Singleton pattern. Both 1st and 2nd level of implementation quality

does not provide the benefits that were explained in the introduction, this is only ensured by implementation on the third level of quality. Level 0^{th} is a special case when there is no fragment in the code that matches the pattern. A comparison of all quality levels is shown in fig.1. Leaving the implementation on the 1^{st} and the 2^{nd} level in the production software will cause additional costs in the future.

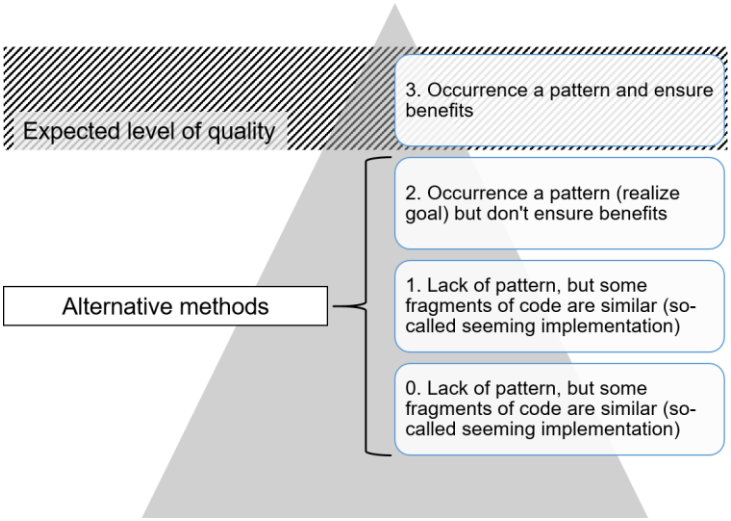

Fig. 1. A comparison of the levels of implementation of design patterns quality

2.2. Alternative methods

The quality of the source code is commonly associated with object-oriented software metrics. Unfortunately, popular metrics do not apply to the implementtation of design patterns, despite the cost of use acceptable in agile vendor teams. Amongst the scientific research related to this issue, the dominating problem is the search for occurrence design patterns (Singh Rao & Gupta, 2013; Tsantalis, Chatzigeorgiou, Stephanides & Halkidis, 2006). The result of the method of finding the occurrence is the number of occurrences of patterns in examined part of the program code or the equivalent of the code. One occurrence of the pattern in most methods is only an information about a compatibility of a part of the code with the template describing reference pattern, on the basis of this part of the code is classified as the occurrence of the pattern. Most methods searching for an occurrence of patterns works in binary, i.e. indicates an occurrence of the pattern or no pattern, which corresponds to an estimation of the assessment of 2^{nd} or 0^{th} level of the quality of implementation, despite all this is insufficient accuracy. Chosen methods additionally enable to show an incomplete occurrence of the pattern (e.g. it contains errors or deficiency in implementation), which corresponds to 1^{st} level of implementation quality. The cost of using methods

searching for occurrence of the patterns is in most cases accepted agile vendor teams. Other research concern methods of verification of pattern implementations, which once again rely on showing the compliance of the tested part of the code with design pattern template (Mehlitz & Penix, 2003; Nicholson et al., 2014). The result of the implementation verification method is the indication of a part of the code, that is compatible with the pattern template. Full compliance with the template corresponds to the 2nd level of quality of implementation, while the exceptions from this correspond to 1st and 0th level. Cost of using methods verifying the implementation of patterns is bigger than possibilities of the agile team, since detailed documentation is required. To sum up, alternative methods do not allow to distinguish implementation compatible with the 3rd level from the 2nd level of quality, i.e. it is not possible to assess whether the implementation of a given pattern provides the expected benefits, including lower costs of development and integration.

3. COMPARATIVE ANALYZES

3.1. Accuracy

Most of the alternative methods are designed to detect instances of design patterns, in addition, these methods are limited to the most popular implementations of patterns that only provide the goal, i.e. the 2nd level of implementation quality. Direct comparison of the proposed method with methods of searching occurrences is unreliable because the result of the search methods (number of occurrences of a given pattern) does not contain information on the quality of implementation of these instances.

In addition to the destination, alternative methods differ in application to selected programming languages. Most alternative methods use Java, and in the case of Danyko the basic language is C#. Despite the many similarities of these languages, this is another reason for direct comparison.

Having considered the above-mentioned difficulties in conducting a direct comparison, methods of similar purpose were selected: (Blewitt, 2006; Nicholson et al., 2014; Mehlitz & Penix, 2003). Then, on the basis of a common representation, a comparative analysis of these methods was carried out, the aim of which is to demonstrate greater accuracy in the analyzed properties of design patterns (which is necessary to distinguish between level 2nd and level 3rd of the implementation quality).

The comparative analysis was performed by decomposing the properties of design patterns, which are analyzed by methods. The Singleton (Wojszczyk & Khadzhynov, 2017) and Strategy (Wojszczyk, 2018) patterns have been limited to an exhaustive example. Each property broken down by the methods compared is assigned the appropriate point value:

- 0 – the method prevents the measurement of a given property of the pattern,
- 0.5 – the method measures ownership inaccurately or does not include all elements in a given property,
- 0.7 – the specification of the method allows to measure a given property, but the author of a given method did not include it in the application to a given pattern,
- 1 – the method measures a given property without reservation.

The result of the comparative analysis is presented in Tables 1–2. Values were introduced after the analysis of each method, using the specification of standards in (Blewitt, 2006), instruction manual up to (Nicholson et al., 2014). The result of the comparative analysis is presented in Tables 1–2. Values were introduced after the analysis of each method, using the specification of standards in (Blewitt, 2006), instruction manual up to (Nicholson et al., 2014). The Strategy template is not described in the specification (Blewitt, 2006), which does not mean that it is not possible to verify the implementation of this pattern. The values in Table 2 are entered on the basis of other standards described in (Blewitt, 2006).

After analyzing the results presented in Tables 1–2, it can be noticed that accuracy in alternative methods is underestimated by fine grained properties, i.e. occurring at the level of individual lines of code. This type of property can be measured with typical numerical metrics (eg, the AHF metric from the MOOD set measures the encapsulation of fields). Next factor reducing the accuracy of alternative methods is the lack of other modifiers and access modifiers. In a case where exactly one of the modifier is expected, it is obvious. However, in other cases, when other modifiers are allowed, this limits accuracy. In the case of (Blewitt, 2006), the lower accuracy is caused by the lack of alternative properties, i.e. only those defined in (Nicholson et al., 2014) are allowed and the others are unacceptable, although they do not constitute inferior solutions.

Tab. 1. Result of the benchmarking for the Singleton pattern (instance sharing by the field), method A – (Blewitt, 2006), method B – (Nicholson et al., 2014), method C – (Mehlitz & Penix, 2003)

Category	Element	Occurrence	Danyko	Method A	Method B	Method C
Field	Modifier	static	1	1	1	0
		default	1	0.7	0	1
		others	1	0.7	0	0
	Access Modifier	public	1	1	1	0
		default	1	0.7	0	1
		others	1	0.7	0	0
	Name	contain „Singleton”	1	0	0.5	0.5
Type	Kind of Type	class	1	1	1	1
		others	1	0.7	0	0
	Modifier	abstract	1	1	1	0
		default	1	0.7	0	0.5
	Access Modifier	public	1	1	1	0
		default	1	0.7	0	1
Constructor	Modifier	default	1	1	1	1
		others	1	0.7	0	0
	Access Modifier	private	1	1	1	0
		others	1	0.7	0	0
Initialization	Checking the existence of an object		1	1	0	1
	Initialization on first use		1	1	0	1
Multi-threading	Synchronization of access to instances		0.5	1	0	1
Use by other types	Kind of relation	association	1	0	1	1
		Inheritance	1	0.7	1	1
	Number of uses		1	0	0	0
Content of Singleton class	The number of methods / fields / properties		1	0.5	0.5	0
	Encapsulation of fields		1	0.5	0.5	0
	detection of additional static elements		1	1	1	0
		Total	**25.5**	**19**	**11.5**	**11**

Tab. 2. The result of the comparative analysis for the Strategy pattern, method A – (Blewitt, 2006), method B – (Nicholson et al., 2014), method C – (Mehlitz & Penix, 2003)

Category	Element	Occurrence	Danyko	Method A	Method B	Method C
Interface declaration	Modifier	default	1	1	1	1
		others	1	0.7	0	0
	Access Modifier	public	1	1	1	0
		default	1	0.7	0	1
		others	1	0.7	0	0
	Name	contain „Strategy”	1	0	0.5	0
	Kind of type	Interface	1	1	1	0
		class	1	0.7	0	1
		others	1	0.7	0	0
Operation declaration	Modifier	abstract	1	1	1	0
		default	1	0.7	0	1
		others	1	0.7	0	0
	Access Modifier	default	1	1	1	1
		others	1	0.7	0	0
	Number of operation		1	0.5	0	0
Implementation of the interface	Modifier	abstract	1	1	1	0
		default	1	0.7	0	1
		others	1	0.7	0	0
	Access Modifier	default	1	1	1	1
		others	1	0.7	0	0
	Kind of type	class	1	1	1	1
		others	1	0.7	0	0
	Implementation of the interface		1	0.5	1	0
	Number of operation		1	0	0	0
Choice of strategy	Number of called strategies		1	0	0	0
Total			**25**	**17.4**	**9.5**	**8**

3.2. The cost of use

When choosing the methods for comparison in terms of the cost of use, it was limited to the methods selected in the previous section, excluding the method (Mehlitz & Penix, 2003) due to the lack of sufficient information about the costs of using this method.

The cost of using the method can be divided into two types: one-off costs initially incurred, before the first use of the method and recurring costs each time the method is used. One-off costs are the construction of templates for design patterns, which should be preceded by assimilation of the appropriate formal representation. The comparison made is limited to individual costs, i.e. one pattern template, one use of the method. The recurring costs include:

- obtaining software or converting source code to a formal form,
- performance of the quality assessment process (or verification of implementation in the case of alternative methods),
- extension of the pattern template with a new variant,
- adding a new assessment criterion,
- obtaining information about changes to improve implementation.

The proposed method and (Blewitt, 2006) include both of these types of costs. However, in (Nicholson et al., 2014) it is necessary to create the appropriate documentation every time, which means that it cannot be considered a one-time cost.

Man-hours are the most authoritative unit that can be used to express the cost of using the method. Unfortunately, the comparison of methods based on such a unit of measure may be biased, because it significantly affects this experience with a given method. An alternative unit of measure may be the number of data entered into the methods, e.g. number of words, operations performed, etc. The number of data entered may be influenced by many factors that are not directly related to the method, e.g. interfaces for communication with the operator, developed tools. Defects resulting from imperfections of interfaces and tools should not affect the cost of using the method. After taking into account these shortcomings, a proposed unit cost per use 1us was proposed – one imaginate Singleton, which corresponds to the workload needed to define a template for a Singleton design pattern in a given formal representation. There is a finite number of elements describing this pattern with each template of the pattern, so with such a defined unit 1us corresponds to 16 elements in the Danyko method, 12 in (Blewitt, 2006) and 8 in (Nicholson et al., 2014). In simplified terms: let Singleton (static field) consist of 3 elements (class, constructor, field) then the work needed to build a template of this pattern equals 1us. Then Singleton enriched with a property (meaning one more element), will be equal to 1⅓ us. In the case when the method prevents the execution of a process related to a given cost component (eg. it does not provide information on possible changes in the implementation), 1us is assigned. Table 3 presents the result of the comparative analysis carried out in terms of the cost of using the methods.

The cost analysis presented in Table 3 does not reflect the production cost of use, i.e. the addition of a new variant is performed once per several iterations, as opposed to the evaluation that is performed cyclically in each iteration. The production cost of using the methods was calculated by simulation, which is based on information received from the external team of the programming company.

The employees estimated that during one year of work they would have incurred the following costs of the method (for one pattern): 1x learning the formalization method, 1x building the pattern, 30x obtaining the code and also the cost of performing the assessment, 15x getting a suggestion for improvement, 3x adding a new variant, 1x adding a new assessment criterion. The sum of individual costs and the sum of simulations are presented in Table 4.

Tab. 3. Comparative analysis of the cost of using particular methods

The type of cost	Danyko	(Blewitt, 2006)	(Nicholson et al., 2014)	Comments
Learning how to formalize	1	1	3	Cost estimated by a team of an external software company
Construction of the reference Singleton	1	1	1	The reference cost from which the unit 1us results
Acquiring the source code	0.1	0.1	0.5	In Danyko and (Blewitt, 2006) it is automated
Performing the assessment or verification	0.1	0.1	0.1	Each method is able to automate this process
Addition of a new variant	0.3	1	1	In the case of (Blewitt, 2006) and (Nicholson et al., 2014) this is not possible, it is necessary to replicate the whole pattern
Addition of a new assessment criterion	0.3	1	1	
Getting suggestions for improvement	0.1	1	0.1	In case of (Blewitt, 2006) this is not possible, in the others it requires reading from the template pattern

Tab. 4. The result of the cost comparison of methods

	Danyko	(Blewitt, 2006)	(Nicholson et al., 2014)
The sum of individual costs from table 3	2.9	5.3	6.7
Sum of costs from simulation	10.7	27	27.5

The high costs of using alternative methods that resulted from the simulation occur mainly in repeatedly performed single costs, such as acquiring source code or obtaining suggestions for improvement. This underlines the important role of adequate formal representation, which confirms the thesis about the choice of data structures based on the object-oriented programming paradigm.

4. PRACTICAL VERIFICATION

Verification of the method carried out in cooperation with the company Poland, which provided the source code. Experiment was carried out using the Command and Factory patterns, which belong to one of the most popular patterns.

The aim of the Command pattern is (Gamma, Helm, Johnson & Vlissides, 1994): *encapsulation of requests in the form of an object*. This allows the client to be parameterized using different requests, and putting requests in queues and logs, as well as provide and undo operation support. Implementation of the pattern is useful when many different operations can be performed on one object (e.g. a bank account). Figure 2 shows a class diagram with an example pattern implementtation, on the basis of (Gamma, Helm, Johnson & Vlissides, 1994). The diagram from the figure 2 shows a structural variant, the modification of this variant is a variant with dynamic mapping (connections in Client class are created dynamically, e.g. by reflection mechanism or injection of dependencies). Presented implementation meets 3rd level of quality.

Elements, of which the Command pattern is made of (Gamma, Helm, Johnson & Vlissides, 1994):

- *AbstractCommand* class – declares a common point to perform operations, other names: parent class, parent type, general command,
- ConcreteCommand – includes the implementation of the *Execute* operation in the form of calling appropriate operation of the Receiver object, other names: Concrete command, subclass,
- *Receiver* – executes a specific command (algorithm), other names: recipient,
- *Client* – creates objects of specific commands and determines connections (maps) with recipients, other names: map, connection mapping,
- *Invoker* – request servicing of the command, other names: sender.

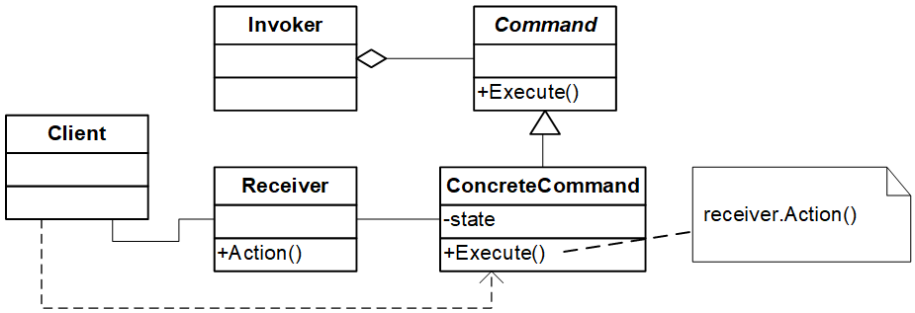

Fig. 2. Class diagram of command pattern, UML notation

The purpose of the Factory pattern is (Gamma, Helm, Johnson & Vlissides, 1994) *to define the interface for creating objects, while the act allows subclasses to determine the class of a given object the creation process is passed to the subclasses*. The implementation of the pattern is useful when different objects carrying information can be created from one operation. Figure 3 shows the class diagram of the sample implementation pattern, on the basis of (Gamma, Helm, Johnson & Vlissides, 1994) and (Metsker, 2004). In (Gamma, Helm, Johnson & Vlissides, 1994) Factory patterns, i.e. Abstract Factory and Factory Method

are described separately, although they are included in one group. In practice, however, programmers unify these patterns and define them as two variants of the Factory pattern. Presented implementation meets 3rd level of quality.

Elements that Factory patterns is made of (Gamma, Helm, Johnson & Vlissides, 1994):

- *Product* – declare the interface of objects generated by the factory, other names: product,
- *ConcreteProduct* – includes the implementation of the Product class, other names: a specific product,
- *Creator* – contains a declaration of the vendor methods that returns *Product* objects, other names: vendor,
- *ConcreteCreator* – override the method from *Creator* to return a copy of the *ConcreteProduct* class, other names: concrete vendor.

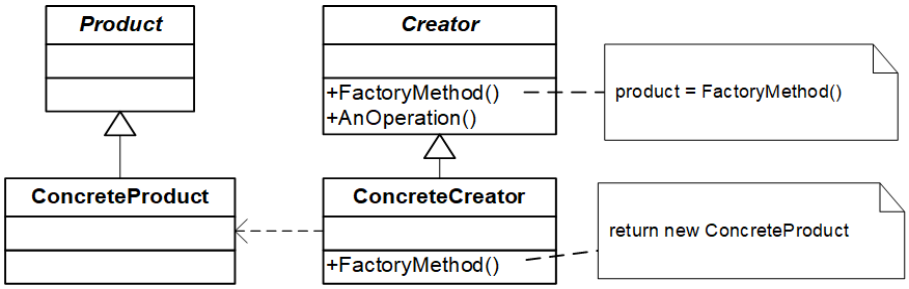

Fig. 3. Class diagram of Factory pattern, UML notation

The results of the assessment of the implementation of design patterns obtained during the experiment are presented in below, these are only parts of the code lower than the 3rd level of implementation:

- In command pattern – type is a class, the class should be replaced with an interface (limited ability to inherit in specific commands) – 2nd quality level,
- In command pattern – occurrence the method with a similar signature, the method name should be changed (possible a risk that the programmer may use a different Execute method than expected) – 2nd quality level,
- In command pattern – not all specific commands are included in the connection map, add the missing commands to the map (unused specific commands can be deleted or re-implanted) – 2nd quality level,
- In command pattern – there is a single call to the so-called in-line (the override type declaration was omitted), the call should be preceded by ICommand declarations (it disrupts the use of the pattern Command, limits the flexibility of the code, the execution of the selected commands is beyond the control of the Command pattern) – 1st quality level,

– In factory pattern – internal modifier (limited availability of the method), changes the access modifier to public (limited availability of the factory, will not be available outside the package, risk of reimplementation) – 2nd quality level.

In the case of the Command pattern, several errors occurred. Probably many of them would not be improved as part of further work, which in the case of expanding the software with new features in the future means more work. By using the method, the errors can be corrected even during the iteration of the experiment. In total, about 10% of the pattern code is below the 3rd level of implementation quality.

In the case of the Factory pattern there was only one error related to access modifiers. These types of errors are often the result of oversight of the implementers and, presumably, they would be successively repaired as part of other code work, however, that they are significantly extended in time. By assessing the quality of implementation of this pattern, the detected errors can be repaired earlier. In total, only 2% of the pattern code is below the 3rd level of implementation quality.

The cost of changes to be made resulting from the detected defects is small in the case of the Factory pattern. The cost of work without the use method Danyko was estimated at 20 man-hour, in case of the Command pattern. There contains time devoted to finding faults, testing, implementation work. After hearing the results of the quality assessment, the company team estimated the cost of work for 6 man-hour (implementation work), therefore the estimated savings in software development is 14 man-hour. The described time consumption does not take into account the time needed to develop reference implementations, however it is a one-time cost. Once developed models will be used many times in the production application of the method.

Obtained result, about 20% of time work of one worker in one iteration, is similar to previous experiment conducted with students (Wojszczyk, 2018), where obtained 28% time profit.

5. SUMMARY

The work presents verification of research results related to the method of assessing the quality of implementation of design patterns. Against the background of the comparative analysis, it was shown that the proposed method is from 45 to 57% less costly in the overall cost of use and from 60 to 61% less costly in the simulation of use for one year. As a result of another comparative analysis, it was also shown that the proposed method is more accurate than 25 to 68% compared to alternative methods.

The obtained results in practical experiment confirm the usefulness of the method in small, agile teams of programmers, where the costs of using such methods should be as low as possible, while maintaining the required accuracy. Further work anticipates automation of selected elements of the method, which will further reduce the cost of use. It is also planned to add additional elements to improve the accuracy of the method.

REFERENCES

Blewitt, A. (2006). *HEDGEHOG: Automatic Verification of Design Patterns in Java* (doctoral dissertation). University of Edinburgh, Edinburgh.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley Professional.

Mehlitz, P. C., & Penix, J. (2003). Design for Verification Using Design Patterns to Build Reliable Systems. *Proc. Work. on Component-Based Soft. Eng*.

Metsker, S. J. (2004). *Design Patterns in C# 1st Edition*. Boston: Addison-Wesley Professional.

Nicholson, J., et al. (2014). Automated verification of design patterns: A case study. *Science of Computer Programming*, *80*, 211-222. doi:10.1016/j.scico.2013.05.007

Singh Rao, R., & Gupta, M. (2013). Design Pattern Detection by Greedy Algorithm Using Inexact Graph Matching. *International Journal Of Engineering And Computer Science*, *2*(10), 3658–3664.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., & Halkidis, S. T. (2006). Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, *32*(11), 896-908. doi:10.1109/TSE.2006.112

Wojszczyk, R., & Khadzhynov, W. (2017). The Process of Verifying the Implementation of Design Patterns—Used Data Models. In L. Borzemski, A. Grzech, J. Świątek, & Z. Wilimowska (Eds), *Information Systems Architecture and Technology: Proceedings of 37th International Conference on Information Systems Architecture and Technology – ISAT 2016 – Part I. Advances in Intelligent Systems and Computing* (*521*, pp. 103–116). Cham: Springer.

Wojszczyk R. (2018). The Experiment with Quality Assessment Method Based on Strategy Design Pattern Example. In: J. Świątek, L. Borzemski, & Z. Wilimowska (Eds.), *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology – ISAT 2017. ISAT 2017. Advances in Intelligent Systems and Computing* (656, 103–112). Cham: Springer.