design patterns, internal software quality, quality assessment

Rafał WOJSZCZYK[*]

THE MODEL AND FUNCTION OF QUALITY ASSESSMENT OF IMPLEMENTATION OF DESIGN PATTERNS

Abstract

One of the ways of providing high internal software quality (that is a source code) is using design patterns. The article aims at presenting a suggested model which enables one to assess the quality of implementation of design patterns. The model assumes verification of different aspects of the patterns and a numeric expression of the obtained results. The analysis of the obtained results may show the occurrence of certain problems which are difficult to be identified during code review or testing.

1. INTRODUCTION

Software is an intangible product, therefore, as a product, it shares some features with tangible products. The distinguishing feature in case of the two mentioned types of products is, among others, the fact that software is not subject to failure, wear or replacement. It seems that once software is produced it could be used forever. Unfortunately, as the reality has shown many times, software must undergo changes and be adjusted to this reality. An example of forced changes was implementation of new VAT rates in Poland in 2011. Not every software was prepared for this implementation without interfering into the source code or using the help of an IT specialist. The mentioned situation shows the importance of providing user-friendly enhancement or modification of software.

A similar example is adapting software to the clients' needs. Large ERP systems are very flexible, therefore, individual changes can be implemented by skilled specialists without interfering into the source code. In case of software produced by small companies, individual adjustment at the implementation stage is often impossible and this leads to changes in the source code. In order to avoid increased costs of these changes, it is necessary to implement suitable

[*] Department of Electronics and Computer Science, Koszalin University of Technology, 75-453 Koszalin, Poland, rafal.wojszczyk@tu.koszalin.pl

mechanisms which will allow for simple development, simultaneously, without disturbing the rest of software. One of the trusty solutions to the mentioned problems are design patterns.

Using design patterns is also very important at other stages of software development cycle. Another example encountered in reality is discontinuation of software development or a change of a development team responsible for certain software. Design patterns are recognized in the community of programming practitioners as a certain language of communication, therefore, the code of software containing design patterns will be more recognizable and under-standable than the one without the patterns.

2. SOFTWARE QUALITY AND DESIGN PATTERNS

Software quality is a very general notion, and a definition of the measure that would directly determine this value is impossible to be provided. When analysing different models of software quality and, especially, maintainability property included in the third part of ISO/IEC 9126 standard [6], one may conclude that the internal software quality (that is software code quality) is determined by the following features: easy development, easy modification, easy-to-understand code. Using design patterns in programming promotes positive values of the mentioned features (positive in a way that thanks to using design patterns, software is easier to be developed, etc.) [3]. Therefore, one can assume that the quality of design patterns implementation is one of the factors determining software quality.

The quality of design patterns implementation, similar to the general software quality, does not have any direct measure. Verification of correctness of the implemented design pattern' structure (in relation to the general template) is only one of the aspects of the process of assessment of design patterns. What is more, there are many other features which influence the implementation quality, e.g. distortion and bad smells, overlapping design patterns, typical errors, anti-patterns, and improper intention of using the patterns. Additionally, in the very structure of design patterns one can distinguish problems with verification, namely, considerable diversity of implementations and the occurrence of many variants of each pattern. There are yet many more problems connected with analysing implementation of design patterns - these were presented, among others, in [14] and [9]. These problems may lead to a misleading situation which maintains structural correctness of the pattern implementation but the use does not correspond with the intention or it does not serve its purpose.

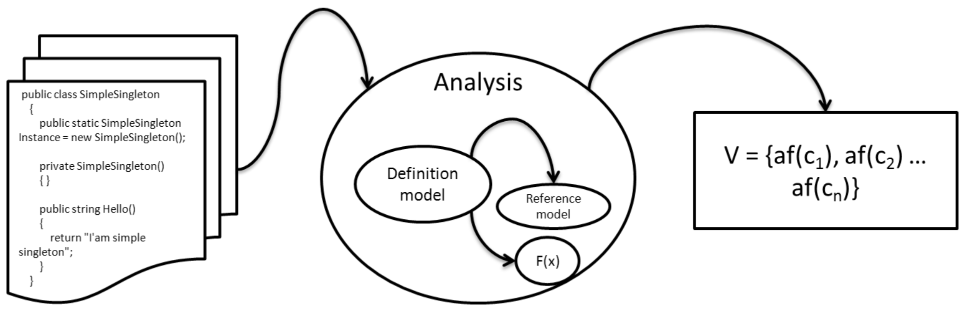
The design patterns described in [3] are templates of ready-made mechanisms which can be used to solve typical problems occurring cyclically in object-oriented designing and programming. However, these are not ready-made solutions since using each pattern requires its proper implementation according to the software context.

The issues discussed in the research studies connected with design patterns very often concern the problem of searching for instances of design patterns in software [11, 10, 1]. The measure of the mentioned studies is presenting a number of instances of design patterns in software, which is insufficient to consider it as quality characteristics. What is more, there are studies conducted which mainly aim at showing structural correctness of implement-tation of the patterns [2], and those also provide too little information. Other attempts of numeric expression of design patterns were introduced in [7] and [8], and those consist in using existent object-oriented software metrics or author's metrics for software containing design patterns. Many years of research on metrics have provided the recommended values of metrics for the software generic case, which obtainment shows high quality. On the other hand, in [4] it was shown that the occurrence of design patterns does not have a positive influence on the results obtained from metrics, therefore, one should perform proper over-interpretation of metrics applied with the patterns, which was presented in [12].

3. ASSESSMENT MODEL

3.1. General assumptions

Figure 1 shows the general concept of the offered model. The first stage is acquiring software which will be subject to assessment of the quality of implementation of design patterns. The acquiring consists in converting software source code to a formal representation and, more specifically, to a proper data structure based on the assumptions of the object-oriented programming paradigm [13]. The analysis stage uses the acquired data solely in a form of the formal representation, and independent variables which are definition model, reference model and mechanisms used to verify the analysed software. The basic result is the vector of assessment obtained from assessment function for the analysed software. In the extended case, the result may include additional information concerning artefacts in the analysed software which show possible errors and change suggestions.


```
public class SimpleSingleton
  {
     public static SimpleSingleton
Instance = new SimpleSingleton();

     private SimpleSingleton()
     { }

     public string Hello()
     {
        return "I'am simple
singleton";
     }
  }
```

Fig. 1. General concept of the model [source: own study]

3.2. Definition model

The definition model is a hierarchical data structure describing the set of features required for verification in the analysed software. Figure 2 shows a symbolic hierarchy of the definition model. The root in the hierarchy is a specific design pattern since, for each design pattern, one should have a proper set of features. The first level of the hierarchy are general categories which group the features. The level of features concerns specific aspects of design patterns and the semantics of this level allows for defining relations "or", "and" as well as "include" between the features. Each feature may be broken down into elements describing the occurrence of detailed artefacts (that is, data types, components, instructions and others).

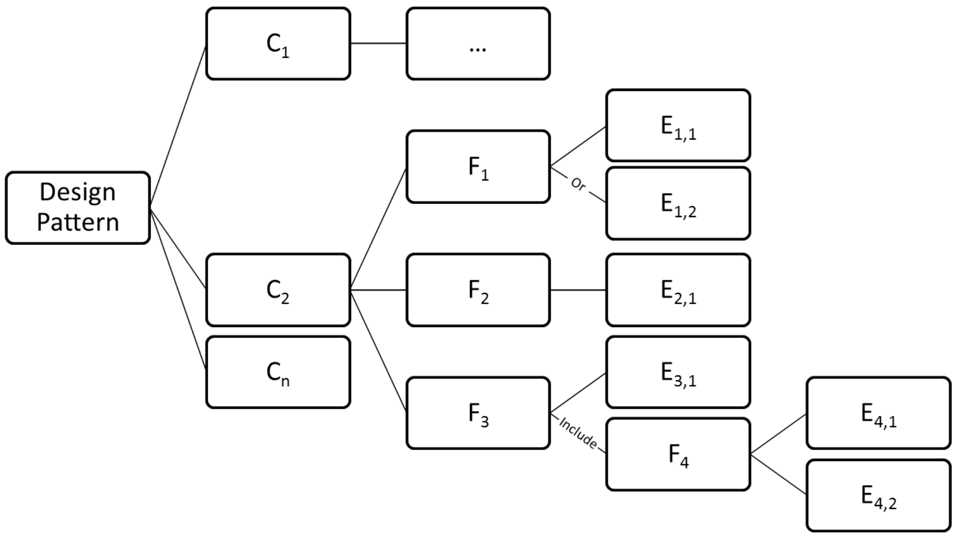


Fig. 2. Symbolic representation of the definition model [source: own study]

Below, one can find an exemplary definition of Singleton design pattern:

- structure - type: non-abstract class, no inheritance; instance: private, static field restored by public static property of type identical to the one of holding class, a suggested name is Instance; constructor: private,
- behavior - initialization: checking for object existence and creation at the first utilization,
- utilization - at least one shareholder and no more than 1/2 of the sum of classes and interfaces in the analysed software,
- connections to other patterns: abstract factory pattern,
- typical errors: other constructors than private ones are forbidden, there should be only one element functioning as an instance,
- other: a proper number of components (fields, methods) included in the pattern class, no inheritance from pattern class.

A principal role of the definition model is providing the description of the features to be verified; still, this model cannot conduct the mentioned verification. The definition model can only show one of the methods of the verification. Thanks to separating verification methods (in other words: operational definitions) from the definition model, increasing effectiveness of this verification is possible through selecting the most appropriate method. Additionally, this approach give great development possibilities in case previous verification methods turn out to be insufficient.

Features occurring in the definition model are diverse in relation to the analysed design pattern, similarly to the occurred categories. An exception is the structural correctness category which occur in every design pattern offered in [3], therefore, one of the verification methods was developed especially for this purpose. It is the verification of the analysed software in relation to the reference model. The reference model includes a general, abstract description of the structure of design patterns. It has been developed as an enhanced data structure based on the assumptions of object-oriented programming paradigm. It enables one to define different variants of the particular elements of design patterns with determining the level of their adjustment to the assumed ideal. Verification in relation to the reference model and other verification methods (e.g. making queries for the obtained software) may be used to verify any features; it is important for every method of verification to return proper result which then will be interpreted by the assessment function. The result is returned in the standardized form; it is the maximum in the interval scale; more specifically, it falls within the range from 0 to 1, where 1 stands for total occurrence of the analysed element. Table 1 presents a dataset of a single feature in the definition model. Each feature belongs to a category. Data included in the category is a name and assessment (identical to feature assessment).

Tab. 1. Dataset included in the feature [source: own study]

Data name	Description
Name	Verbal determination of the feature, understandable to a human.
Feature dependence	Indicating other feature and determining type of dependence related to it: and, or, include.
Assessment	Complex data structure used directly by the assessment function. It includes: weighting factor, activation threshold and recommended value, and, after verification is done, also the obtained result.
Multiplicity	Complex data structure containing information on the expected number of occurrences of a particular feature in the analysed software. It determines expected value in a form of the following ranges: from to, greater than or equal, less than, exactly.
Set of elements	Complex data structure which indicates the detailed artefacts and describes their possible variants. It includes: name identifying an element, assessment, that is, the structure identical to feature assessment, set of references to the reference model or other verification method, and negating the element, which means that all possibilities different from the indicated one are allowed.

3.3. Working principle

The above-described definition model has one more important role – it indicates subsequent stages of analysis. In the generic case, the process of analysis assumes the following: for each category one should verify desired features, while for each verified feature one should verify elements. Each stage of verification gives a partial result even if it does not satisfy the activation threshold. The final selection of results will be performed by the function initializing the assessment. Each verification stage is encumbered with checking for occurring restrictions, e.g. those connected with multiplicity (number) of occurring elements and performance of required verification operations.

The partial results are added to assessments of the corresponding elements (see Table 1 - Set of elements, Assessment). After conducting the required verifications, the function initializing assessment and assessment function are performed. Both functions are performed up the hierarchy, that is, initializing assessment is performed on the basis of results obtained from verification of elements; then, on the basis of a modified dataset, the assessment function is performed, which result is a result for the feature. Subsequently, this process is repeated for the features and, finally, for categories. The initializing functions checks the activation threshold and selection on the basis of relations between the features. If activation threshold is not satisfied then a particular element or feature does not occur; in special case, if activation threshold is 0, then occurrence of a particular element or feature is obligatory.

3.4. Execution

The offered model was executed as a prototype tool in Microsoft .NET technology. The used technology allowed for improvement of the process of acquisition of the analysed software, which consists in acquisition of object-oriented structure from CIL managed code using one of the methods of the reverse engineering, e.g. Mono.Cecil [13]. Using the managed code allowed for avoiding typical syntactic errors occurring in the source code and removed useless code fragments, e.g. unit test code, processor directives and other from the analysed software. An additional advantage resulting from the used technology is a possibility of applying the model to different programming languages compiled to the managed code. Formal data representation containing the acquired software was executed as a database in Microsoft SQL Server environment.

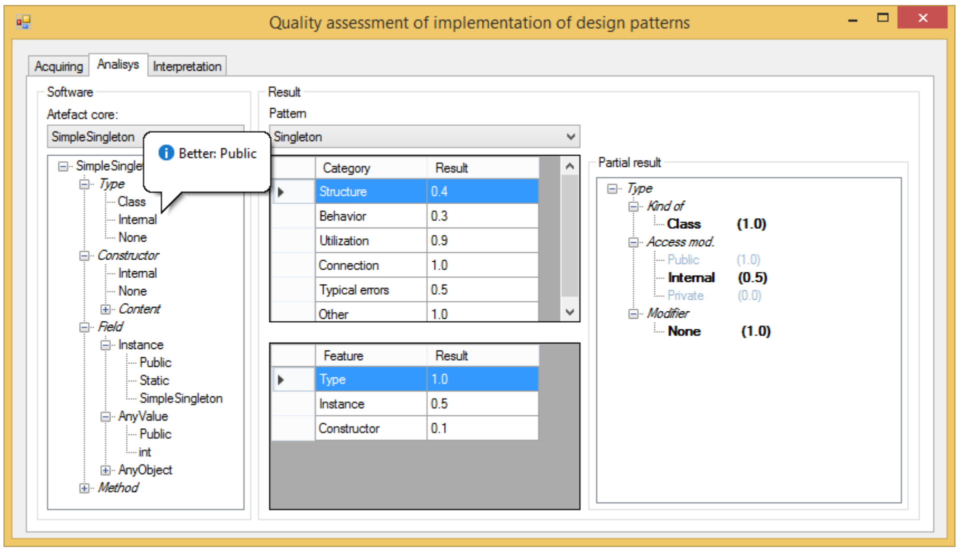
The analysis of the features of design patterns takes place according to the hierarchy described in the definition model which was also executed as a database in Microsoft SQL Server environment, similarly to the above-mentioned reference model. The worked out structure of the definition model allowed for accessible implementation of the major loop in which the subsequent features are iterated, and the analysis is conducted for each of those.

In case of "include" dependence, recurrence function is performed into the hierarchy; this is presented by listing 1. Possibility of verifying features by means of different mechanisms is a polymorphism property joining the definition model with verification mechanisms. After conducting the analysis, the results are recorded in an object-oriented data model creating the vector of assessment. An exemplary result of calculation is shown in Figure 3. After selecting a required category and then a feature (see Fig. 3), the software displays additional information resulting from the particular feature analysis: the partial results show attributes which occur for the analysed software (with level of adjustment in brackets); way of presentation of partial results is dependent on the used verification mechanism; in the symbolic representation of the analysed software, one can find hints concerning better solutions and error messages with improvement suggestions.

Listing. 1. Major loop code

```
        public void AnalyzeAll(Definition definition)
        {
            foreach (Feature feature in definition.Features)
                AnalyzeRecursively(feature);
        }

        private void AnalyzeRecursively(Feature feature)
        {
            Verify(feature.Elements);
            if (feature.KindOfDependence == KindOfDependence.Contain)
                AnalyzeRecursively(feature.DependenceFeature);
        }
```



Software

Artefact core: SimpleSingleton

Better: Public

```
SimpleSingle
  Type
    Class
    Internal
    None
  Constructor
    Internal
    None
    Content
  Field
    Instance
      Public
      Static
      SimpleSingleton
    AnyValue
      Public
      int
    AnyObject
  Method
```

Result

Pattern: Singleton

Category	Result
Structure	0.4
Behavior	0.3
Utilization	0.9
Connection	1.0
Typical errors	0.5
Other	1.0

Feature	Result
Type	1.0
Instance	0.5
Constructor	0.1

Partial result

```
Type
  Kind of
    Class       (1.0)
  Access mod.
    Public      (1.0)
    Internal    (0.5)
    Private     (0.0)
  Modifier
    None        (1.0)
```

Fig. 3. Prototype execution [source: own study]

The main area of using the model is the whole process of software implementation where it can be used simultaneously with other tools taking care of generally understood quality, good practices and code cleanliness. It should be especially useful in companies or development teams which take care of providing high quality in the aspects of software maintenance and development. A very practical application would be integrating the model with programming environments to follow and assess changes connected with implemented design patterns on an ongoing basis. Finally, using the model in the opposite direction, that is, with at least semi-automatic implementation of the patterns in software on the basis of provided definitions is worth considering.

4. ASSESSMENT FUNCTION

The basic output of the offered model is vector of assessment of the following form

$$v = \{af(c_1), af(c_2), ..., af(c_n), \qquad (1)$$

where: $af(c_1)$, $af(c_2)$, ..., $af(c_n)$ is a result of assessment function for each subsequent category c_1, c_2, ..., c_n. The quality of each of those categories is expressed by the following function

$$af(c_1) = af(f_1, f_2, ..., f_n), \qquad (2)$$

where: f_1, f_2, ..., f_n is a set of features belonging to c_1. Quality of the particular features in the generic case is determined by the following formula

$$af(f_1) = af(e_1, e_2, ..., e_n), \qquad (3)$$

where: e_1, e_2, ..., e_n is a set of elements belonging to a particular feature. In a special case, where a particular feature is in "include" dependence in relation to other feature, the quality may be determined by the following function

$$af(f_1) = af(e_1, e_2, ..., e_n, f_{1,1}, f_{1,2}, ..., f_{1,n}), \qquad (4)$$

where: e_1, e_2, ..., e_n is once again a set of elements belonging to a particular feature, while $f_{1,1}$, $f_{1,2}$, ..., $f_{1,n}$ are features of "include" dependence.
The basic formula describing the assessment function is

$$af(x) = \frac{\sum_{n=1}^{n} x_n * w_n}{\sum_{n=1}^{n} w_n}, \qquad (5)$$

where: n – sum of arguments,
x_n – particular argument (that is, element, feature, category),
w_n – argument weighting factor.

One can notice that it is the weighted mean. Means (arithmetic or weighted) are often used in methods assessing software quality, e.g. in [5] or in recognized CK object-oriented software metrics [12]. The offered model allows a possibility of presenting other assessment function in case when the primary function turns out to be insufficient. Weighting factors may initially be determined on the basis of expert knowledge or by method for valuation. What is more, the possibility of selecting proper weighting factors by means of artificial neural networks is taken into consideration.

5. INTERPRETATION OF THE RESULTS

An example of the obtained assessments rounded to decimal for three different occurrences of Singleton pattern was presented in Table 2.

Tab. 2. Exemplary results [source: own study]

	Case 1	**Case 2**	**Case 3**
Structure	1	0,4	1
Behavior	0,7	0,3	0,7
Utilization	0,6	0,9	0,1
Connection with other patterns	1	1	0
Typical errors	1	0,5	1
Other	1	1	0,3

Implementation of Singleton pattern in case 1 may be considered as of good quality. High structural correctness and connections with other patterns were maintained. The maximum assessment in typical errors category shows that these errors did not occur. Lower assessment in the behavior category results from a lack of reference synchronization to Singleton instance in multithread environments. Lower assessment in the utilization category results from too many references to instance which may indicate too high responsibility of the class with Singleton pattern.

Case no 2 showed low quality of implementation in the structural correctness, typical errors and behavior categories. Software containing such implementation may work improperly, e.g. there may be errors connected with data incompa-tibility or the so-called runtime errors. These errors should be detected already at the stage of software execution (testing stage) and repair cost should be on the budget (for error recovery) in the general production process. However, one should take into account that Singleton design pattern is considered to be one of the simplest patterns, therefore, the repair costs may be higher in case of other patterns.

Case no 3 showed low quality of implementation in categories of utilization, connections with other patterns and other features connected with too many components. The analysis of that case showed that the pattern was misused in connection to the intention of its use or it was improperly understood by the implementing developer. The problems resulting from such implementation are difficult to be detected, both during code inspection and at testing stage. The most noticeable problems will occur during software maintenance and development. Repair costs are nearly unpredictable.

6. SUMMARY

The article briefly explains the influence of quality of implementation of design patterns on internal quality of software and presents selected research studies connected with verification of implementation of design patterns.
It discusses the proposal of the model which allows for numeric expression of design patterns. The model consists of the hierarchical definition model and mechanisms responsible for verifying features of patterns. The result of the offe-red model is a vector of assessment which is obtained through the assessment function. The assessment function uses partial results acquired in the verification process and weighting factors from the definition model. The analysis of the vector of assessment and detailed results enables one to predict possible problems connected with software development and its improper working.

Further works related to the model include development of dependences of definitions occurring in the model (e.g. set of artefacts which satisfies requirements of a certain feature may be an input set for other feature), development of verification mechanisms and possibilities of analysing the results.

REFERENCES

[1] BINUN A.: *High Accuracy Design Pattern Detection*. PhD Thesis, Rheinischen Friedrich Wilhelms Universitat Bonn, 2012.

[2] BLEWITT A.*: HEDGEHOG: Automatic Verification of Design Patterns in Java*. PhD Thesis, University of Edinburgh, 2006.

[3] GAMMA E. et al.: *WZORCE PROJEKTOWE. Elementy oprogramowania wielokrot-nego użytku*. Helion, Gliwice, 2010.

[4] HERNANDEZ J. et al.: *Selection of Metrics for Predicting the Appropriate Application of design patterns*. 2nd Asian Conference on Pattern Languages of Programs, 2011.

[5] HOŁODNIK-JANCZURA G.: *Badanie jakości produktu informatycznego metodą wartościowania*. Badania Operacyjne i Decyzje, Oficyna Wydawnicza Politechniki Wrocławskiej, ISSN 1230-1868, Wrocław 2007, pp. 55-69.

[6] ISO/IEC TR 9126-3, Software Engineering – Part 3: Internal metrics, ISO/IEC 2003.

[7] KHAER Md. A. et al.: *An Empirical Analysis of Software Systems for Measurement of Design Quality Level Based on Design Patterns*. Computer and information technology, IEEE, 2007.

[8] MASUDA G., SAKAMOTO N., USHIJIMA K.: *Evaluation and Analysis of Applying Design Patterns*. IWPSE - International Workshop on Principles of Software Evolution, 1999.

[9] RASOOL G.: *Customizable Feature based Design Pattern Recognition Integrating Multiple Techniques*. PhD Thesis, Technische Universitat Ilmenau, Ilmenau 2010.

[10] SINGH RAO R., GUPTA M.: *Design Pattern Detection by Greedy Algorithm Using Inexact Graph Matching*. International Journal Of Engineering And Computer Science, Vol. 2, No. 10, 2013, pp. 3658-3664.

[11] TSANTALIS N. et al.: *Design Pattern Detection Using Similarity Scoring*. IEEE Transactions on Software Engineering, Vol. 32, No. 11, 2006, pp. 896-908.

[12] WOJSZCZYK R.: *Zestawienie metryk oprogramowania obiektowego opartych na statycznej analizie kodu źródłowego*. Zarządzanie projektami i modelowanie procesów, Zeszyty Rady Naukowej Polskiego Towarzystwa Informatycznego, ISBN 978-83-7518-599-7, Warszawa 2013, pp. 95-107.

[13] WOJSZCZYK R.: *Pozyskiwanie struktury obiektowej z kodu zarządzanego przy wykorzystaniu metod inżynierii odwrotnej*. Inżynieria oprogramowania: badania i praktyka, Zeszyty Rady Naukowej Polskiego Towarzystwa Informatycznego, ISBN 978-83-63919-15-3, Warszawa 2014, pp. 199-213.

[14] WOJSZCZYK R.: *Koncepcja hybrydowej metody do oceny jakości zaimplementowanych wzorców projektowych*. Zeszyty Naukowe Wydziału Elektroniki i Informatyki nr 7, Wydawnictwo Uczelniane Politechniki Koszalińskiej, ISSN 1897-7421, Koszalin 2015, pp. 17-26.