An Approach to Class Diagrams Verification According to SOLID Design Principles

Elena Chebanyuk[1] and Krassimir Markov[2]
[1]Department of Software Engineering, National Aviation University, Kyiv, Ukraine
[2]Department of Information Modelling, Institute of Mathematics and Informatics at Bulgarian Academy of Sciences, Sofia, Bulgaria

Keywords: SOLID, Class Diagram Verification, Software Architecture, Class Diagram Designing, Model-Driven Engineering, Model-Driven Development.

Abstract: An approach, verifying class diagram correspondence to SOLID Design Principles, is proposed in this paper. SOLID is an acronym, encapsulating the five class diagram design principles namely: Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation and Dependency Inversion. To check whether a class diagram meets to SOLID, its analytical representation is analyzed by means of predicate expressions. For every SOLID design principle corresponded predicate expressions are proposed. Analytical representation describes interaction of class diagram constituents, namely classes and interfaces, in set-theory terms. Also criteria for estimation of obtained results are formulated. Example of class diagram verification according to the suggested verification approach is also represented in this paper. The advantages of the proposed verification approach implementing to improve the quality of different software development lifecycle processes are outlined in the conclusions.

1 INTRODUCTION

Class diagrams are central artefacts for performing many operations such as analysis of software architecture, software designing, reengineering and different other activities.

To design effective class diagram it is necessary to meet all tiers of patterns for designing software. The highest designing tier corresponds to SOLID design principles. The next tier is architectural styles describing interconnection of main components in software system. The next tier touches of architectural patterns. And the most concrete designing tier is application of design patterns used to set interactions between class diagram constituents.

When scalable project is designed, amount of class diagrams to be processed is great. Considering this fact, the task to verify class diagram structure is actual. Using an analytical representation of class daigrams and formal description of verification rules, provides a background for designing automated tools for class diagram refinement.

2 RELATED PAPERS

Consider papers, describing algorithms and methods when analysis or processing of class diagram structure is required.

Today, methods for class diagrams processing are developing in several directions:

- class diagram refinement (López-Fernández et al., 2014);
- estimation of architectural solutions (Tombe R. et al., 2014);
- development of Model-Driven Architecture (MDA) operations (Wang et al., 2014); (Sandhu, 2015).

Let's consider research results, represented in the mentioned papers.

Criteria of metamodel quality estimation are proposed in the paper (López-Fernández et al., 2014). A library of metamodel properties is created and outlined. Because of the number of such properties is great, the procedure of analysing metamodel is difficult to be formalized. That is why, only a verbal description of metamodel quality criteria is proposed.

Chebanyuk, E. and Markov, K.
An Approach to Class Diagrams Verification According to SOLID Design Principles.
DOI: 10.5220/0005830104350441
In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016)*, pages 435-441
ISBN: 978-989-758-168-7

A method for software maintenance risk assessment is represented in the paper (Tombe et al., 2014). This method expects class diagram designing. The obtained class diagram is matched with a set of design patterns. A prototype of software for analysing source code is also described. The functionality of this prototype is to analyse source code to find known patterns. Due to the variety of styles for code development, the task of designing universal method for code analysis is very complicated.

A method for verification of Graph-Based model transformation is proposed in the paper (Wang et al., 2014). In order to implement the method, a Model-Transformation system had been designed. Transformation rules, as a part of this system, are proposed. Using predicate logic formal description of transformational conditions in these transformation rules is described. Due to complexity and variety of conditions more detail formal description is very difficult to be obtained.

Model-Driven Development (MDD) challenges are analysed in the paper (Sandhu, 2015). One of the main important MDD tasks is to facilitate the code reuse process. Before code reusing, the procedure of code analysis should be performed. Effective code analysis allows improving its structure and quality. This process can be automated when patterns matching code and model elements are used.

Class diagrams are central artefacts for domain analysis (Sandhu, 2015). Processing of class diagram analytical representation increases the quality of domain analysis artefacts, for example, ontologies.

Applying quality class diagrams for performing of any software development activity, such as model execution, ontology designing, models comparison or refactoring, and others, improves the quality of all software development processes. It grounds the actuality of task to design techniques, approaches, and methods for class diagrams verification.

3 TASK

Task: to propose an approach to check whether class diagram corresponds to SOLID principles.

For this purpose we have to do the following:

- prepare an analytical representation of class diagram according to algebra describing software static models;
- using predicate logic design expressions for checking whether considering class diagram meets to every of SOLID design principle. The aim of using expressions to process class diagram is to obtain quantity characteristics for measurement the level of satisfying class diagram to SOLID principles.
- obtain quantity initial information for further analysis.

4 DENOTATIONS FOR CLASS DIAGRAM ANALYTICAL REPRESENTATION

All denotations are taken from notation of algebra, describing software static models, represented in the paper (Chebanyuk, 2013).

Algebra operates with such instances as class – $c \in C$ (where C is a set of classes in class diagram), abstract class – $c^a \in C^a$ (where C^a is a set of abstract classes in class diagram), interface – $i \in I$ (where I is a set of interfaces in class diagram), software component and software module.

Class diagram is represented as a tuple of classes and interfaces.

$$CD =< C, I > \quad (1)$$

Then, algebra contains detailed description of operations that are made to interconnect classes and interfaces. Functionality of class is spread when it interacts with other classes (interfaces) by means of operations. Set *OPER* of operations is the following:

$$OPER = \{inh, ass, aggr, comp\} \quad (2)$$

where: *inh* - inheritance, *ass* - association, *aggr* – aggregation, *comp* - composition.

Consider $c, c^| \in C$. General idea of spreading class functionality $F(c)$, when c and $c^|$ are connected by means of $oper \in OPER$, is denoted as follows:

$$F(c)^{oper} = F(c) \cup F(c^|) \quad (3)$$

Sign $\cup$ depicts that functionality of class c ($F(c)$) is extended with functionality of class $c^|$ $(F(c^|))$.

The same is true when functionality of $c \in C$ is spreading by inheritance or including reference to $i \in I$ It is denoted as follows:

$$F(c)^{oper} = F(c) \cup i \quad (4)$$

Number of public methods of class $c \in C$ is denoted as follows: $n(B_c^{public})$, where - B_c^{public} is a set of public methods of class.

More detail description of class constituents and class diagram operations are represented in the paper (Chebanyuk, 2013).

5 RULES FOR CLASS DIAGRAM VERIFYING ACCORDING TO SOLID DESIGN PRINCIPLES

5.1 Single Responsibility Design Principle

The essence of Single responsibility Design Principle is that a class should have just one function, namely: "*a class should have only one reason to change"* (Martin et al., 2006).

In other words, in well-designed class every public method is aimed to realise concrete task or part of it. Then, in order to follow this considered principle, the number of such methods should be limited. Otherwise BLOB class, mixing several responsibilities, is obtained.

When class diagram is designed, cognitive principles of information processing should be taken into account (Chebanyuk and Markov, 2015). According to Miller recommendation (Miller, 1956), it is proposed to set the limit of class public methods as 9.

It is necessary to note that when public methods are absent, class can implement no operation.

Then, the condition that $c \in C$ satisfies to single responsibility design principle is formulated as follows:

$$P(c, n(B_c^{public})) = \begin{array}{l} c \in C, n(B_c^{public}) \in \{1,2,...,9\}, \\ = (\text{c has } n(B_c^{public}) \text{ public methods}) \end{array} \quad (5)$$

$\forall i \in I$ also should be checked to this design principle using (5).

5.2 Open-Closed Design Principle

The essence of Open-Closed Design Principle is the next: architectural solution should be open for extension and closed for modification simultaneously.

In order to prove that class diagram corresponds to Open-Closed Design Principle, it is matched to key structural elements of design patterns. This thesis is explained by the following:

a) flexibility of design patterns is provided by means of presence in their structure of some abstract entities such as abstract classes or interfaces (Gamma et al., 1994).

b) abstract entities on class diagram allow increasing its functionality when existence class diagram constituents do not touched.

c) design patterns fully correspond to SOLID Design Principles. It can be easily noticed by analysing of their class diagrams (Gamma et al., 1994).

In order to prove that group of classes corresponds to Open-Closed design principle, it is matched to design pattern structure. To achieve this goal the following steps are performed:

1. Key structural elements of design pattern are defined. Doing this, all information sources about design pattern structure are analysed, namely: design pattern purpose, software requirements that match to specific design pattern, its class diagrams and code templates,

2. Preparing an analytical description of design pattern structural components in terms of algebra, describing software static models.

Consider this process for analytical description of design pattern Strategy.

1. Define key structural characteristics of Strategy design pattern by means of analysing class diagram of this pattern, and its textual description (Gamma et al., 1994).

Analysis of class diagram and Strategy functional requirements allows defining structural characteristics of Strategy design pattern.

a) $\exists c \in C$, which has at least one reference to interface. Denote reference to interface i that is included to this class c as $i(c) \in I$.

b) $i(c) \in I$ should have classes inheritors.

c) Optional condition: considering $\exists c \in C$ should have classes' inheritors.

2. Preparing an analytical description of design pattern Strategy structural components in terms of algebra, describing software static models

a) $\exists c \in C$, which has at least one reference to interface. Denote a set of such interfaces as $I(c) \in I$. Then:

$$\begin{array}{l} \exists c \in C, \exists i \in I, \\ F(c)^{aggr} = F(c) \cup i \\ P(F(c)^{aggr}) = (F(c)^{aggr} \; exists) \\ I(c) = \{i \mid P(F(c)^{aggr}) = true;\; c \in C, i \in I\} \\ |I(c)| > 1 \end{array} \quad (6)$$

b) $\forall i(c) \in I(c)$ should have classes inheritors. Denote a set of such classes as $C(i)$. Then:

$$\begin{aligned} &\exists c \in C, \forall i(c) \in I(c), \\ &F(c)^{inh} = F(c) \cup i \\ &P(F(c)^{inh}) = (F(c)^{inh}\ exists) \\ &C(i) = \{i \mid P(F(c)^{inh}) = true; \\ &i(c) \in I(c), c \in C\} \\ &|\ C(i)\ | > 1 \end{aligned} \tag{7}$$

c) $\exists c \in C$ has at least one inheritor class. According to algebra (Chebanyuk, 2013), class that inherits $c \in C$ is denoted as c_1. Denote a set of such inheritors as $Cl(c)$. Then:

$$\begin{aligned} &\exists c \in C, \exists c_1 \in C, \\ &F(c_1)^{inh} = F(c_1) \cup F(c) \\ &P(F(c_1)^{inh}) = (F(c_1)^{inh}\ exists) \\ &Cl(c) = \{c \mid P(F(c_1)^{inh}) = true; c, c_1 \in C\} \\ &|\ Cl\ (c)\ | \geq 1 \end{aligned} \tag{8}$$

5.3 Interface Segregation Design Principle

The essence of Interface Segregation Design Principle is the following: *"the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients. Thus, some clients use one group of methods, and other clients use the other groups."* (Martin et al, 2006).

In other words, this Design Principle is formulated by following: every class that inherits interface (interfaces) should not contain empty methods.

$$\begin{aligned} &\exists c \in C, \exists i \in I, \\ &F(c)^{inh} = F(c) \cup i \\ &B_c^{public} = \{\beta_c^{public} \mid \beta_c^{public} \neq \varnothing\} \end{aligned} \tag{9}$$

5.4 Liskov Substitution Design Principle

The essence of Liskov Substitution Design Principle is the following: *"if for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."* (Martin et al., 2006).

Define the three structural characteristics for verifying Liskov Substitution Design Principle:

a) $\exists c \in C$ has a reference to another class diagram class $c^{|} \in C$. These classes are not connected by inheritance relationship. Then:

$$\begin{aligned} &\exists c, c^{|} \in C \\ &F(c)^{acc} = F(c) \cup F(c^{|}) \\ &P(F(c)^{acc}) = (F(c)^{acc}\ exists\ ) \\ &P\ (F\ (c)^{acc}\ ) = true\ ;\ c^{|}, c \in C \end{aligned} \tag{10}$$

b) $c^{|} \in C$ has at least two inherited classes. Denote a set of such inheritors as $Cl(c^{|})$. Then:

$$\begin{aligned} &\exists c^{|} \in C, \exists c^{|}{}_1 \in C, \\ &F(c^{|}{}_1)^{inh} = F(c^{|}{}_1) \cup F(c^{|}) \\ &P(F(c^{|}{}_1)^{inh}) = (F(c^{|}{}_1)^{inh}\ exists) \\ &Cl(c^{|}{}_1) = \{c^{|}{}_1 \mid P(F(c^{|}{}_1)^{inh}) = true, \\ &c_1, c^{|}{}_1 \in C\} \\ &|\ Cl(c^{|})\ | > 2 \end{aligned} \tag{11}$$

Expression (11) should be applied for other classes inheritors in $c^{|} \in C$ hierarchy, namely $c_2{}^{|}, c_3{}^{|}, ..., c_n{}^{|} \in C$.

c) $c_1{}^{|} \in Cl(c^{|})$ has non empty overridden methods. Denote a set of these methods as

$$B_{c_1^{|}}{}^{override} \in B_{c_1^{|}}{}^{public}$$

Then:

$$B_{c_1^{|}}{}^{override} = \{\beta_{c_1^{|}}{}^{override} \mid \beta_{c_1^{|}}{}^{override} \neq \varnothing\} \tag{12}$$

Expression (12) also should be applied for other classes inheritors in $c^{|} \in C$ hierarchy.

5.5 Dependency Inversion Design Principle

The essence of Dependency Inversion Design Principle is the following:

*"**a)** High-level modules should not depend on low-level modules. Both should depend on abstractions.*

***b)** Abstractions should not depend upon details. Details should depend upon abstractions."* (Martin et al, 2006).

Dependency Inversion Design Principle requires only one structural characteristic: functionality of $c \in C$ should be extended by means of one of the three variants, namely: $c^a \in C$, $c^| \in C$ (classes that are related on a top of hierarchy) or $i \in I$.

$$F(c)^{acc} = \begin{cases} F(c) \cup F(c^a) \\ F(c) \cup i \\ F(c) \cup F(c^|) \end{cases} \quad (13)$$

6 CASE STUDY

Consider a process of class diagram verification (Figure 1) according to the suggested approach.

Form a set of class diagram constituents.

Named class is denoted *c(name)* and Named interface: *i(name)*. Then:

$$CD = < C, I >$$
$$C = \{c(Car), c(Taxi), c(\text{Re}ntalCar),$$
$$c(RaceCar), c(ToyCar), c(Jeep), c(CarryPeople), \quad (14)$$
$$c(NonCarrer), c(CarryLoad), c(RacenoWay)\}$$
$$I = \{i(CarryBehaviour), i(RaceBehaviour)\}$$

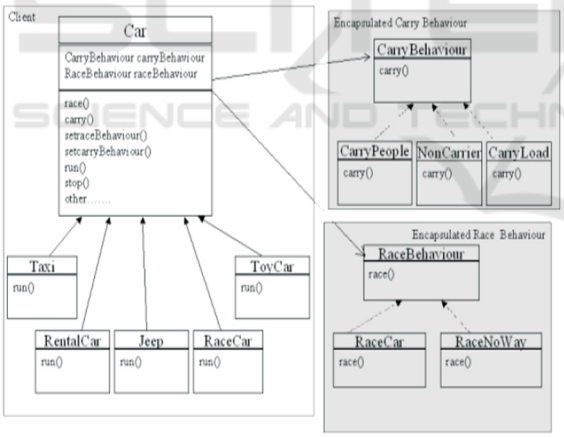

Figure 1: Class diagram representing Strategy design pattern (Ikram, 2005).

1. Single Responsibility Design Principle

To check class diagram to single responsibility design principle every $c \in C$ is considered.

$$c \in C, 1 \le n(B_c^{public}) \le 9$$

2. Open-Closed Design Principle

The class diagram, representing simple schema of Strategy design pattern is given on the Figure 1. This class diagram is taken from (Ikram, 2005).

In this figure class "Car" has a references to interfaces "CarryBehaviour" and "RaceBehaviour". These interfaces have classes' inheritors.

The purpose of class "Car" is to realize unique algorithm when some steps of this algorithm can be different. Namely for the realization of different steps of an algorithm these interfaces and their classes' inheritors are responsible.

In other words, classes, that inherit the interface "CarryBehaviour", namely "CarryPeople", "NonCarry", and "CarryLoad", encapsulate some step of general algorithm. Respectively classes that inherit the interface "RaceBehaviour", namely "RaceCar" and "RaceNoWay", encapsulate other step of the same algorithm.

Let's prove that this diagram corresponds to Strategy Design Pattern.

In order to do this, the key characteristics of Strategy design pattern according to (6)-(8) are defined.

Define number of $I(c(Car)) \in I$

$$\exists c(Car) \in C,$$
$$\exists i_1, i_2 \in I,$$
$$i_1 = i(CarryBehaviour),$$
$$i_2 = i(RaceBehaviour)$$
$$F_1(c(Car))^{aggr} = F(c(Car)) \cup i_1 \quad (15)$$
$$P_1 = P(F_1(c(Car))^{aggr}) = true$$
$$F_2(c(Car))^{aggr} = F(c(Car)) \cup i_2$$
$$P_2 = P(F_2(c(Car))^{aggr}) = true$$
$$| I(c(Car)) | = | \{i_1, i_2\} | = 2 > 1$$

3. Define number of classes, inheriting $i_1, i_2 \in I$

$$\exists c \in C, i_1 \in I(c(Car)),$$
$$C(i_1) = \{c(CarryPeople),$$
$$c(NonCarry), c(CarryLoad)\} \quad (16)$$
$$| C(i_1) | = 3 > 1$$

$$\exists c \in C, i_2 \in I(c(Car)),$$
$$C(i_2) = \{c(RaceCar), c(RaceNoWay)\} \quad (17)$$
$$| C(i_2) | = 2 > 1$$

4. Define number of c(Car) inheritances

$$\exists c \in C, \exists c_1 \in C,$$
$$F(c_1)^{inh} = F(c_1) \cup F(c) \quad (18)$$
$$P(F(c_1)^{inh}) = true$$

$$Cl(c(Car)) = \{c(Taxi), c(\mathrm{Re}ntalCar),$$
$$c(Jeep), c(RaceCar), c(ToyCar)\}$$
$$|Cl(c(Car))| = 5 > 1$$

The concussion: all class diagram quantity characteristics matching with structural key features of Design Pattern Strategy.

5. Interface Segregation Design Principle

As it was shown in the previous sub point, considering class diagram matches to Strategy Design pattern.

When class diagrams, implementing Strategy Design Pattern are created the condition:

$$\forall i \in C(i), P(c) = (\beta_c^{\ public} = \varnothing)$$
$$P(c) = true \quad (19)$$

Condition (19) provides flexibility of strategy Designe Pattern (Gamma et al, 1994).

But the same condition contradicts to (9).

6. Liskov Substitution Design Principle

In order to check whether this diagram satisfies the Liskov Substitution Design Principle, check it by (10)-(12).

Consider C. As there are no classes, containing references to another ones, the conclusion to be made: that class diagram does not satisfy to Liskov Substitution Design Principle. In other words, if any of the conditions (10)-(12) is not proved, class diagram does not satisfy this principle.

7. Dependency Inversion Design Principle

Review class diagram. Define association links in it.

$$F(c(Car))^{aggr} = F(c(Car)) \cup i_1 \cup i_2$$

As the condition formulated in (13) is proved then the concussion: that this class diagram is designed according to Dependency Inversion Design Principle.

7 CONCLUSIONS

The approach of class diagrams verification according to SOLID Design Principles is proposed in this paper.

Formalization of checking correspondence of class diagram to SOLID principles (5)-(13), proposed in this paper, allows designing methods and techniques for automated checking whether analytical representation of class diagrams meets to SOLID design principles. Applying of these methods and techniques allows estimating class diagram features before performing different operations with it.

The application of the suggested approach will allow:

- increase the quality results of risk assessment method, proposed in the paper (Tombe et al., 2014). Before risk assessment, class diagram can be verified for meeting SOLID. Results can be estimated in two ways, namely, increasing the range of risk factors or defining which diagrams need further estimation;
- improve the structure of metamodel for further transformation (Wang et al., 2014). Metamodels contain initial information for designing ontologies, profiles and other activities in Model-Driven Development. That is why class diagram refinement, when its verification is one of the refinement techniques operations, allows improving the class diagram quality.

REFERENCES

Chebanyuk E. 2013. Algebra Describing Software Static Models. *International Journal "Information Technologies and Knowledge", Vol.7, Number 1, 2013. ISSN 1313-0455 (printed) ISSN 1313-048X (online), pg. 83-93.*

Chebanyuk E., Markov K., 2015. Software Model Cognitive Value. *International Journal "Information Theories and Applications", Vol.22, Number 4, 2015. ISSN 1310-0513 (printed), ISSN 1313-0463 (online), pg. 338-355.*

Gamma E., Helm R., Johnson R., and Vlissides J., 1994. (the GangOfFour) Design Patterns: Elements of Reusable Object-Oriented Software. *AddisonWesley Professional, • ISBN 978-0201633610 , ISBN 0-201-63361-2 . 431 pg.*

Ikram S. 2005. Design Patterns (Strategy Pattern) Part – II. *C# Corner. http://www.c-sharpcorner.com/UploadFile/saif_ikram/DesignPatternsPart208312005062925AM/DesignPatternsPart2.aspx.*

López-Fernández J.J., Guerra E., de Lara J., 2014. Assessing the Quality of Meta-models. In *Boulanger F., Famelis M. and Ratiu D., editors, Proceedings of 11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014, co-located with Models 2014, Valencia, Spain, September 30th 2014. pg. 3-12.*

Martin R., 2000. Design Principles and Design Patterns. *http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.*

Martin R., Martin M. 2006. Agile Principles, Patterns, and Practices in C#. *Prentice Hall, 2006. ISBN-10: 0-13-185725-8, ISBN-13: 978-0-13-185725-4. Pg: 768.*

Miller G.A., 1956. "The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for

Processing Information," Psychological Review, vol. 63, no. 2, Mar. 1956, pp. 81–97. http://www.psych.utoronto.ca/users/peterson/psy430s2001/Miller%20GA%20Magical%20Seven%20Psych%20Review%201955.pdf (accesed 01.09.2015).

Sandhu R., 2015. Model-Based Software Engineering (MBSE) and Its Various Approaches and Challenges. In *COMPUSOFT, An international journal of advanced computer technology, 4 (6), June-2015 (Volume-IV, Issue-VI), ISSN:2320-0790. pg. 1841-1844.*

Tombe R., Okeyo G., Kimani S., 2014. Cyclomatic Complexity Metrics for Software Architecture Maintenance Risk Assessment. In *International Journal of Computer Science and Mobile Computing, Vol.3 Issue.11, November- 2014, ISSN 2320–088X, pg. 89-101. Available Online at www.ijcsmc.com.*

Wang X., Büttner F., Lamo Y., 2014. Verification of Graph-based Model Transformations Using Alloy. In *Hermann F., Sauer S., editors, Proceedings of the 13th InternationalWorkshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014). Electronic Communications of the EASST, Volume 67 (2014), ISSN 1863-2122. http://www.easst.org/eceasst/*