Optimizing UML Class Diagrams

Maxim Sergievskiy[1] and *Ksenia* Kirpichnikova[1]

[1] Department of System Analysis, National Research Nuclear University MEPhI, Moscow, Russia

Abstract. Most of object-oriented development technologies rely on the use of the universal modeling language UML; class diagrams play a very important role in the design process play, used to build a software system model. Modern CASE tools, which are the basic tools for object-oriented development, can't be used to optimize UML diagrams. In this manuscript we will explain how, based on the use of design patterns and anti-patterns, class diagrams could be verified and optimized. Certain transformations can be carried out automatically; in other cases, potential inefficiencies will be indicated and recommendations given. This study also discusses additional CASE tools for validating and optimizing of UML class diagrams. For this purpose, a plugin has been developed that analyzes an XMI file containing a description of class diagrams.

1 Introduction

Most recently, UML has become the most commonly used tool for modeling object-oriented software systems [1]. At the design stage, the most widely used UML tool is class diagram (CD). It is known, that any mistakes at the design stage, then affect the implementation phase and either require an additional fine-tuning, or, in more serious cases, could eve require an additional prototype. Complexity and wide applications of different UML patterns could potentially restrict the ability of automatic verification: screening for structural errors, incompatible or redundant components. It is also important to be able to assess the efficiency of CD from the point of view of subsequent implementation.

Since the knowledge of developers in the design of software system are not always sufficient, it not unusual to see CDs which require adjustments. This process can be organized in such a way, that changes are made to the diagram manually in a graphical format. However, certain changes can be carried out automatically with the help of a specialized CASE tool, which contains additional validation functions. To do this, the developer need to be able to analyze the description of the CD and identify structural errors in the diagrams. In addition, it is desirable to have a set of CD transformation rules in accordance, for example, with design patterns that allow you to optimize the model.

2 Representation of class diagrams

Let us focus on the approaches that exist for the representation of the CD. The two main ones are a standard description in the form of a graph with different types of vertices and edges, and a representation in the form of a formal system. There are several ways to formally describe a CD; the most commonly used of them are those that are based on the use of OCL, Z + language and descriptive logics. A real verification is feasible for the two latter options [3, 8].

Consistency check of a CD. It is known that semantics of UML does not allow the use of certain combinations of structures. For example, classes do not allow any direct or indirect inheritance from itself, also class associations should not have the same attributes as classes associated with them, etc. A formal description of all available limitations (of which are many) could turn into a complex and time-consuming process. Any attempts to do this lead to excessive complexity of formal models [10]. Instead, additional analysis can be used that will allow to detect errors of this kind.

Optimization of a CD is the development of rules for replacing certain constructions with more optimal ones. Which CD should be considered more optimal is often not a straightforward question [7]. For example, the use of many standard design patterns [2] leads to an increase in the number of CD elements, but improves its quality in terms of subsequent code generation and modifiability. For example, the inclusion of interfaces to the CD often increases the versatility and reusability of the future code, but this could reduce clarity. Optimization if often associated with identification of inefficient parts of a CD, from the point of view of the subsequent realization. For example, certain fragments of a diagram could have a more efficient form of representation. An approach based on the concept of a design anti-pattern – unpromising to implement a CD fragment [9] – can also be applied. The essence of this approach is that if an anti-pattern is identified in a CD, the designer is given information on the nature of the

---

[*] Corresponding author: sermax@yandex.ru

problem and proposes to make changes to the CD. Here is a list of typical fragments of the CD [2, 5], which either contain errors, or can be optimized.

In some cases, these fragments can be described in the form of design patterns or anti-patterns (see Table 1).

Table 1. Typical fragments of a class diagram

№	Name of a fragment	Importance	Reaction
1	Inheritance of classes	High	Error
2	Inheritance of interfaces	High	Error
3	Aggregation	High	Error
4	Class, not associated with other elements	Medium	Error
5	Interface, not associated with classes	High	Error
6	Multiple inheritance of classes	High	Warning
7	Multiple aggregation	Low	Warning
8	Circular association relations	Low	Warning
9	Circular association relations and associated classes	Low	Warning
10	Ternary association relations	Low	Warning
11	Pattern, lowering the arity of the association relation	Low	Recommendations for correction
12	Anti-pattern "Blot"	Medium	Warning
13	Pattern "Chain of duties"	Low	Recommendations for correction
14	The same attributes of the class and association class	High	Error
15	Pair dependency relations between classes: A -- B and B -- A	Low	Warning
16	Big number of methods in the class	Low	Recommendations for correction
17	Big number of parameters of the method	Low	Warning

3 Examples of optimization

Below are examples of several design patterns and anti-patterns the paragraph are straight.

3.1 The same attribute for class and class association associated with them

In general, the same attributes in different classes are permissible, although, in order to avoid ambiguity, one should strive to minimize their number. But duplicating attributes of a class and class association linked to it indicates an error, because directly or indirectly, class association attribute also applies to all classes that are members of the association. In this example (Figure 1), the NAME attribute in the TIMETABLE class is superfluous.
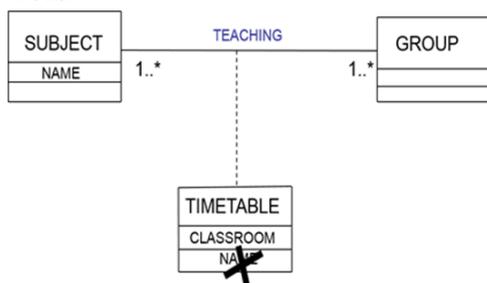
Fig. 1. The same attributes.

3.2 Anti-pattern "blot"

This situation arises when a given class monopolizes both the data itself and the methods of working with this data. That means that this anti-pattern is characterized by the presence of one complex class ("blot"), which includes many fields and methods, while the associated classes contain fields and possibly a small number of methods. The essence of the blot anti-pattern is that, most likely, the methods of the blot-class can and should be included in other classes containing information related to the work of these methods.

3.3 A cycle of association relations, which includes an association class

The meaning of this anti-template is the following: if semantically linked relations form a cycle that passes through a class association, then it is likely that one of them is redundant (Figure 2). A removal of the redundant association can be made only by the designer.

In this case, it is logical to select the TEACHING association relation for deletion. This choice is determined by the fact that you can find out which disciplines the lecturer teaches by navigating through the TIMETABLE association class to objects of the SUBJECT class.
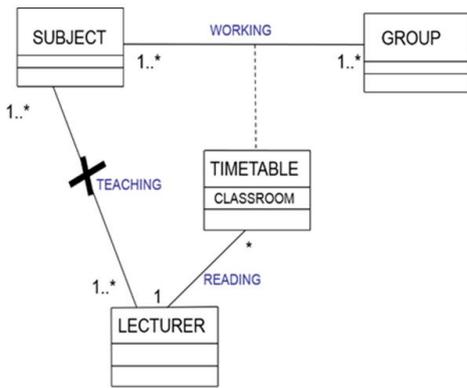
Fig. 2. Example of the anti-pattern "Cycle of association relations, including the class-association".

3.4 A pattern that allows you to switch from a ternary association to a binary association

If the ternary association has a class with multiplicity (1), then the ternary association can be replaced by a combination of a binary association and a class association [6]. Class diagram, illustrating this case, is shown in Figure 3.
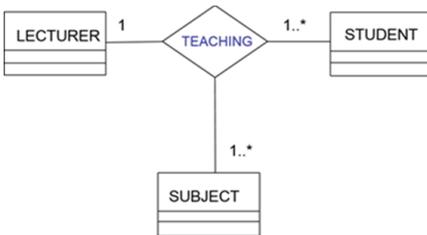

Fig. 3. Example of the ternary association.

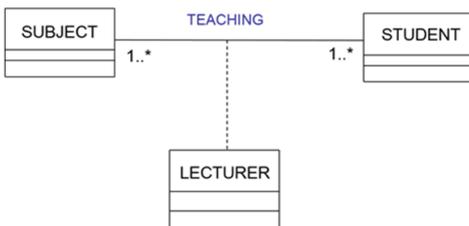The optimized diagram is shown in the Figure 4.

Fig. 4. Replacing ternary association on binary and class-association.

4 The tool for optimizing the class diagram

Modern CASE tools usually contain simplified means of validating CD, which identify only gross errors connected solely with the rules of charting, and do not contain optimization tools. Although the latter is fully understandable, since generally accepted rules for optimizing the CD do not exist, this does not prevent us from using existing methods of describing a CD on the one hand, and a set of optimization rules on the other. To expand CASE-tools with validation and optimization mechanisms for a CD is necessary to implement the following: (1) to develop a method of screening for non-

optimal fragments of the CD, using available patterns and anti-patterns. (2) to introduce a method allowing for automatic optimization of certain fragments of CD.

As a basic CASE tool, let us consider IBM's Rational Software Architect package based on the Eclipse platform. It supports all basic elements of the CD, including interfaces, class associations, scopes, etc. Rational Software Architect is written on Java and easily extendable with plugins. To search for erroneous and non-optimal fragments, it is necessary to know the internal representation of a CD for the CASE tool. To store the UML CD in Rational Software Architect the XML Metadata Interchange (XML) format is used, being standard for exchanging metadata using the XML language [11]. Files, created in XMI format, are most often used to exchange UML diagrams between different applications. They store information about diagrams in XML format.

Let us briefly consider the structure of the XMI file. After root tag followed by <packagedElement> tags, corresponding to various elements of the diagram. Each chart element must have the attributes xmi: type and xmi: id.

For the class (xmi: type = "uml: Class") and the interface (xmi: type = "uml: Interface") the name attribute is the class name. The class fields are marked with the child tag <ownedAttribute>, whose attributes are xmi: type = "uml: Property", xmi: id, name and visibility. The methods of the class are marked with the child tag <ownedOperation>, whose attributes are xmi: type = "uml: Operation", xmi: id, name, the method parameters are represented as the child elements <ownedParameter> with the attributes xmi: type = "uml: Parameter", xmi: id and name. A template for the fragment of the XMI code, describing the class, is shown below:

```
<packagedElement
    xmi:type="uml:Class"
        xmi:id="class ident"
    name="class name">
  <ownedAttribute
    xmi:type="uml:Property"
    xmi:id=" attribute ident"
        name="attribute name"
    visibility="access modificator"/>
  <ownedOperation
        xmi:type="uml:Operation"
        xmi:id="operation ident"
        name="operation name">
  <ownedParameter
        xmi:type="uml:Parameter"
        xmi:id=" parameter ident"
        name="parameter name"/>
  </ownedOperation>
 </packagedElement>
```

The generalization relation is represented by a child element in the class-successor with the <generalization> tag whose attributes are xmi: type = "uml: Generalization", xmi: id and general - identifier xmi: id of the ancestor class.

The implementation relationship is represented by a separate element with tag <packagedElement>, whose

attributes are xmi: type = "uml: Realization", xmi: id, supplier - xmi: id of the provider interface, client - xmi: id of the consumer class. Also the clientDependency attribute appears in the client class, in which the xmi: id of the implementation relationship is stored.

The dependency relationship is represented as a separate element with the <packagedElement> tag whose attributes are xmi: type = "uml: Dependency", xmi: id, name name, supplier - xmi: id of the provider class, client - xmi: id of the consumer class. Also the clientDependency attribute appears in the client class, in which the dependency relation identifier is stored.

The association relation is represented by a separate element with the <packagedElement> tag, whose attributes are xmi: type = "uml: Association", xmi: id, name of the dependency name, memberEnd - space-separated xmi: id of the child elements of the member classes of the association. These elements are represented by the <ownedAttribute> tag and have the attributes xmi: type = "uml: Property", xmi: id, name, visibility, xmi: id of another association member type, association identifier. The children of the children represent the association rates with the <upperValue> and <lowerValue> tags.

The aggregation and composition relationships are represented in the same way as the binary association relation, but for them additional attributes of the class-integer marked with the <ownedAttribute> tag are added to the aggregation attribute, which takes the value "shared" for aggregation and "composite" for the composition. The child element appears in the association element itself, which contains similar information for the part class.

The class association is represented as a separate element with the <packagedElement> tag and the attributes xmi: type = "uml: AssociationClass", xmi: id, name, memberEnd - space separated xmi: id of additional child elements of the association class. The child elements of the association class are similar to the elements of the class. Additional children are elements with the tag <ownedEnd>, they are similar to the same elements of the aggregation and composition relations. Below is the fragment of the XMI code, describing the association relation:

```
<packagedElement
    xmi:type="uml:Association"
    xmi:id="relation ident"
    name="association name"
    memberEnd="class1 ident" "class2 ident"/>
```

## 5 Description of the presentation of the class diagrams

To implement algorithms allowing to search for various fragments of a diagram (see Table 1), the excessive standardized XML form is ineffective. Hence,

it is more efficient to use a different form of storage, focused on a faster search for elements of a CD. Let us describe it. A CD can be described with a set of classes (each class contains a name, attributes and methods) and a set of directed unweighted graphs for all types of relations. Such a representation is possible, since the relations between classes and interfaces - generalization, association, aggregation, composition, implementation and dependence do not contain circular references. Thus, CD can be presented as a set of graphs, with peaks being classes (interfaces), and arcs - relations. In each graph the set of peaks will be the same; all the graphs will be oriented. As the data structure for the presentation of relationship graphs in CD, lists should be preferred to arrays, since in most cases graphs of relations arches. Base element and the list of adjacent to it should not be presented in the form of a mapping, with a base class identifier playing the key role and the value is a list of identifiers of adjacent classes.

What is stored in contiguity lists? For the relationship graph, the generalizations are the ancestor classes for the descendant, for the implementation relationship graph, the interfaces implemented by the class, for the dependence relationship graph, the source classes for the client class, for the aggregation and composition graph, the class-integer for the part class. Since the attitude of the association is not oriented, it will be represented by two relations, hence the association relation in the graph is duplicated. A class-association is represented as a standard class, associated with classes that generate it. In relation to identifying structural errors in UML class diagrams, duplications of association relationships are not critical and can be omitted. To avoid multiple copying, information about the fields and methods of the class is

placed in a separate list, objects of which will be objects-concepts (classes and interfaces) with mandatory fields and methods.

This presentation form is particularly useful to detect errors such as circular reference, multiple inheritance, or aggregation and incoherence of fragments. Note that transformation from the XMI format to the form of multi-level linked lists is usually a straightforward exercise, which could be performed by an existing algorithm, based on the standard capabilities of parsers. Since the total number of elements in the class diagrams is relatively manageable (usually within several thousand), the problem of finding erroneous and non-optimal fragments with this form of representation could be solved with a standard search algorithms.

The structural scheme, describing the result of adding the validation and optimization plugin to the CASE tool of Rational Software Architect, is shown in Figure 5. The scheme demonstrates additional functionality and data used.
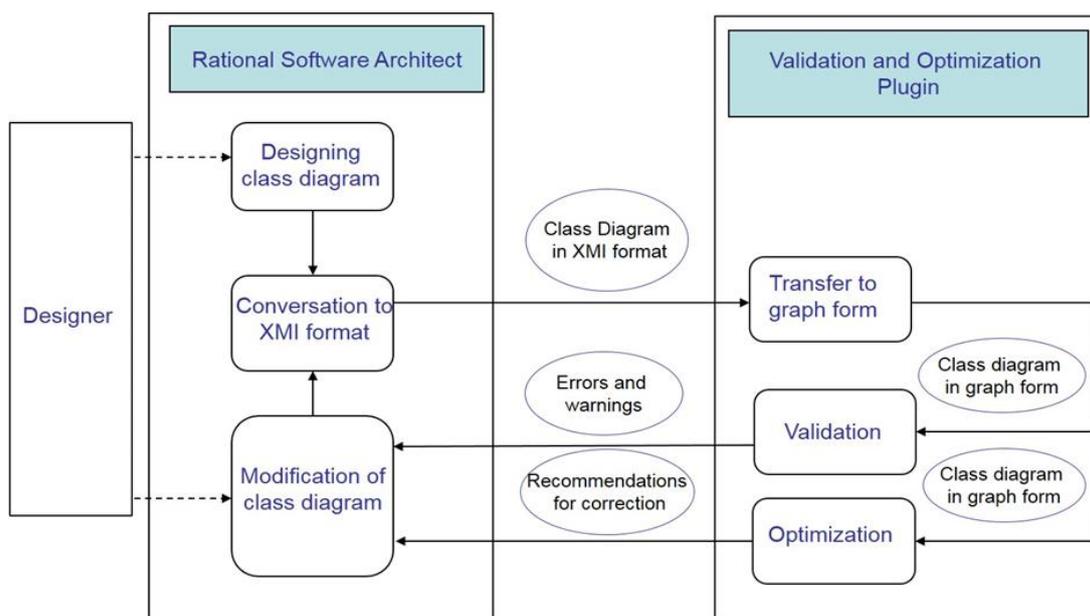
Fig. 5. Structural scheme, describing the operation of the improved CASE tool.

6 Conclusion

The manuscript describes an approach to optimizing models of software systems at their design stage. The described approach is based on an automated search for erroneous and inefficient fragments, using both design patterns and anti-patterns. The report with detected inefficient fragments is being made available to a designer, who makes final modifications to the model. In addition, the system can recommend certain transformations, and in some cases carry out the required transformation automatically. The relative advantage of the approach is that verification and optimization of the model are carried out at the design stage, which allows to minimize the time processes of debugging and refactoring code. This work describes the implementation of optimization of a CD for IBM's Rational Software Architect package. Since the description of CD in Rational Software Architect is presented in the XMI format, widely used for transferring UML diagrams, the developed optimization tool can be used for other CASE tools that support this format. To implement the algorithms of searching fragments of a CD, it appears to be more efficient to use another form of storage (not XMI) - a multilevel system of linked lists.

References

1. J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language. Reference Manual* (Addison-Wesley, Reading, MA 1998)

2. E. Gamma, R. Johnson, Helm R., J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software* (Addison-Wesley, 2001)

3. D. Berardi, D. Calvanese, G. D. Giacomo, Artificial Intelligence, **168**, 70 (2005)

4. M. Sergievskiy, A. Konkinб Cloud of Science, **4**, 465 (2017)

5. M. Sergievskiy, International Journal of Advanced Computer Science and Applications, **8**, 268 (2017)

6. M. Sergievskiy, International Journal of Advanced Computer Science and Applications, **7**, 265 (2016)

7. E. Niculchev, O. Deryugina, International Journal of Advanced Computer Science and Applications, **7**, 76 (2016)

8. A. Cali, D. Clavanese, G. D. Giacomo, M. Lcnzerini, A Formal framework for reasoning on UML class diagram. *In Proc. of the 13th Int. Sym. on Methodologies for Intelligent Systems* (2002)

9. W. Brown, R. Malveau, H. McCormick III, T. Mowbray. *AntiPatterns. Refactoring software, architectures, and projects in crisis* (John Wiley & Sons, Inc. 1998)

10. A. Grigoriev, A. Kropotin, E. Ovsyannikova. The problem of detecting consistencies on UML class diagrams. *In Proc. of International scientific-practical conference on modern problems and ways of their solution in science, transport, production and education '2012* (2012).

11. Materials of OMG (2005) http://www.omg.org/spec/XMI/2.1/