Checking Inconsistencies in UML Design

Iryna Zaretska[1], Oleksandra Kulankhina[1], Hlib Mykhailenko[1], and Roman Kovalenko[1]

[1] V.N. Karazin Kharkiv National University, Kharkiv, Ukraine
zar@univer.kharkov.ua, mary.cauliflower@gmail.com,
tas.nix@gmail.com, kovalenkoo.roman@gmail.com

Abstract. The paper presents a simple method and software implementation for checking inconsistencies in UML design and the general method of UML design verification using its own model and first order predicate logic to specify relations between components of the design. Unlike various existing methods the proposed ones are focused mostly on cross-diagram inconsistencies and strong adhering to object-oriented principles. The model proposed for the general method is based on the unified graph representation of UML diagrams.

Keywords. Software design, object-oriented approach, UML, design model, verification.

Key Terms. SoftwareSystem, SoftwareComponent, Object, Model, VerificationProcess.

1 Introduction

Software design has become an increasingly important part of the software lifecycle due to the increasing complexity of software under construction.

The Unified Modeling Language (UML) is the de facto standard for modeling software systems. The UML supports a wide range of diagrams for modeling software. UML diagrams are independent but connected; their meta-model describes them under a common roof.

Detection of errors at the software design level allows reducing a great number of problems in the late stages of software development. There are several approaches for testing design models but they are mainly dealing with intra-diagram inconsistencies or using scripts for design execution.

In this paper model faults concerned with cross-diagram inconsistencies are considered. First, we present a simple method for detecting the cross-diagram inconsistencies in a UML design and its Java implementation. Then the general model of UML design for verification and refining purposes is introduced and discussed.

2 Related Work

2.1 OCL Constraints

The most common approach to set the rules of consistency for a UML model is to use Object Constraint Language (OCL) which is supported by the majority of UML CASE tools. In fact a lot of OCL constraints are embedded into the wide spread UML CASE tools in order to provide inconsistency verification of the model. The fact is that one can freely use OCL on the intra-diagram level but not on the cross-diagram level.

2.2 Critic Approach

A number of UML visual design tools provide model verification support including syntax checking and structural and consistency analysis. One of the components of such integrated support tools are critics.

There are various definitions of a design critic or a critic system in the literature. A critic can be considered as an intelligent user interface that evaluates a design made by a user and provides feedback to assist the user to improve the design. Generally critic tools detect potential problems, give advice and alternative solutions, and possibly automated or semi-automated design improvements to the end user.

Design critic tools have been used in design tools for various domains, including software engineering, design sketches, education, etc. Several studies report the benefits of applying design critic tools in software developments activities [1 – 8].

One of the critic tools is ArgoUML, an open source UML CASE tool. This tool supports the editing of UML notation diagrams and detects common errors made by software designers. For example, after placing a class in a class diagram, several critiques are displayed reminding the user that the class requires a better attribute name, needs operations, constructor and associations with other classes, and its class name needs to be capitalized. Thus, the user is helped to improve the design through the critiques. Java API is used to implement these features.

Other examples of the critic-based tools are ArchStudio3, SoftArch, DAISY, IDEA, ABCDE-Criticand AIR. These tools provide knowledge to architects, designers, and requirement engineers who lack specific understanding of the problem or solution domains. These critic tools all produce critiques that are specific to their problem domain. They use various approaches such as Java API, Prolog rules and knowledge bases, first-order production systems etc. to design and define critiques constraints. These tools have several limitations such as particular code or design language orientation or difficulties in customization requiring comprehension the critic's domain.

2.3 UML Design Execution

A number of approaches have been proposed to execute UML models [9 – 12]. Most of them use UML models to generate high level language code and execute the generated code.

Mellor and Balcer [11] for example use model compilers to support UML model execution. A set of domain and platform-specific model compilers are available commercially for realtime system modeling. At present the compilers cannot be extended to incorporate specific checks as the compiler source is not freely available for modifications.

Another technique for executing UML designs is to execute code that is generated from the model. Assuming that the code and model both contain the same information, executing the code is the same as executing the model.

Trung T. Dinh-Trong at el. [12] offer the systematic approach to testing UML designs based on a Java-like action language (JAL) used to transform the UML design under test into the executable form and then exercise them with generated inputs.

3 Cross-Diagram Inconsistencies

It is quite important to verify that the information about the model of the system at one UML diagram does not contradict to the information at the other UML diagram. We call such contradictions cross-diagram inconsistencies.

A UML design may contain different cross-diagram inconsistencies. Some of them are listed below.

1. An instance of the class A sends the message to the instance of the class B at the Sequence diagram, but the class B isn't visible for the class A at the Class diagram (Fig. 1).

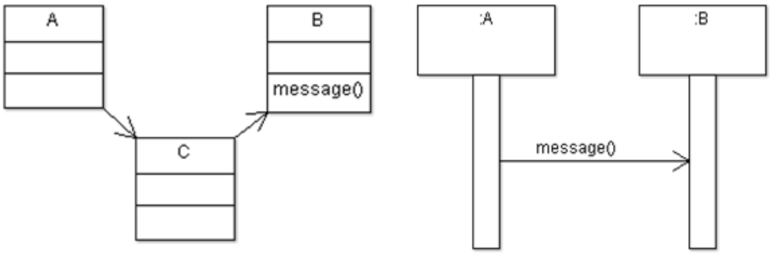


Fig. 1. Class B should be visible.

2. An instance of the class A sends the message to an instance of the class B at the Sequence diagram, but there is no corresponding method in the class B (Fig. 2).

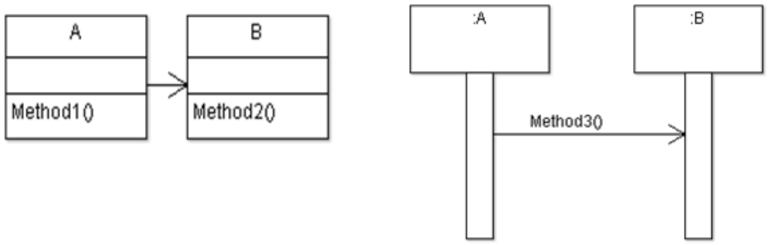


Fig. 2. Class B does not have method3().

3. Transition from one state of the class A to another at the State Chart diagram occurs by the class A method invocation, but there is no such method in the class A at the Class diagram (Fig. 3).

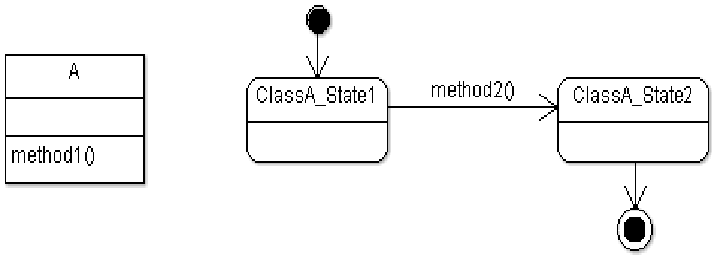


Fig. 3. Class A does not have method2().

4. An instance of the class A sends the message to the class B (but not to the instance of this class) at the Sequence diagram, but the corresponding method of the class B isn't specified as static (Fig.4).

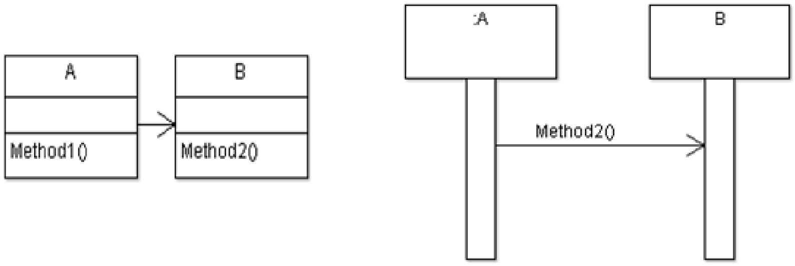


Fig. 4. In class B method2() is not specified as static.

As our analysis shows the most wide spread UML CASE tools cannot “see” such faults in the design and neither critic tools nor the design execution method are of any help in a cross-diagram verification process.

4 Simple Method for Detecting Cross-diagram Inconsistencies in UML Design

As most of the UML CASE tools allow exporting an object oriented design (OOD) of a target system into XMI format we offer a simple method of cross-diagram verification: we parse XMI file and find UML components dependences we are interested in. Depending on the type of an inconsistence under check we develop different check modules with their own models and consistency rules. The whole process looks like shown in Fig. 5. After parsing the XMI file into the Document Object Model (DOM) the Visitor takes care of getting over its elements and creating instances of the classes from the Checker’s model. Then the concrete Checker verifies this model according to its own rules and generates check results. To support this process the Java plug-in was developed. We call it Cross Diagram Inconsistency Check Plug-in (CDICP). It can be easily added to most of the UML CASE tools.

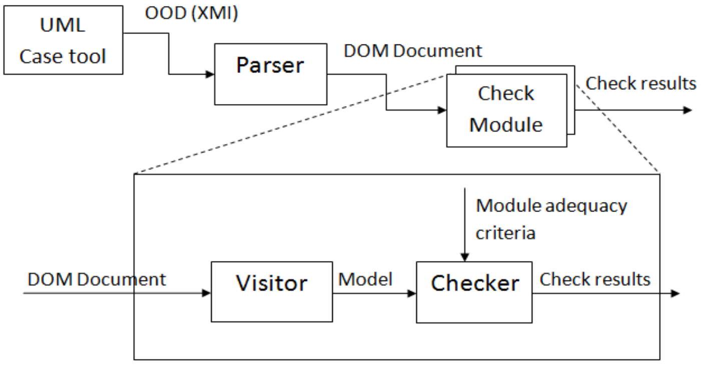


Fig. 5. The whole process of checking UML design.

As an example of a Check Module (Fig. 5) we developed Visibility Check Module (VCM) which finds the inconsistencies of the first type (Fig.1).

VCM uses three classes for UML components representing; we call them Role, AssRole, and SeqRole (Fig. 6). The class Role represents a class at the Class diagram, the class SeqRole represents this class at the Sequence diagram (in fact they are two different UML components), and the class AssRole represents an association or dependency between classes at the Class diagram or a message between classes at the Sequence diagram (Fig. 7).

The Visitor in VCM identifies these components in the DOM, creates their instances and places them to the lists of classes, associations, messages, etc. Then the Checker applies its rules, verifies them and generates the result messages. The

Checker of VCM for every instance of the class AssRole, which represents some message between classes, checks if there is an association or dependency between these classes (another instance of the class AssRole) and detects its direction. If the connection is not found the Checker forms error message. This message contains information about cross-diagram inconsistency specifying its type and names of classes.

The CDICP plug-in for the VCM was developed and incorporated into Eclipse (Fig. 8).

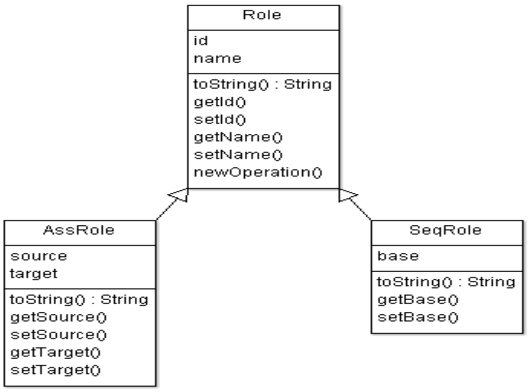


Fig. 6. Classes used by VCM.

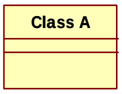
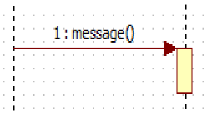
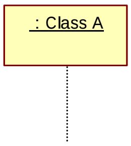
			
Role	AssRole	AssRole	SeqRole

Fig. 7. Elements of UML design and correspondent classes of VCM.

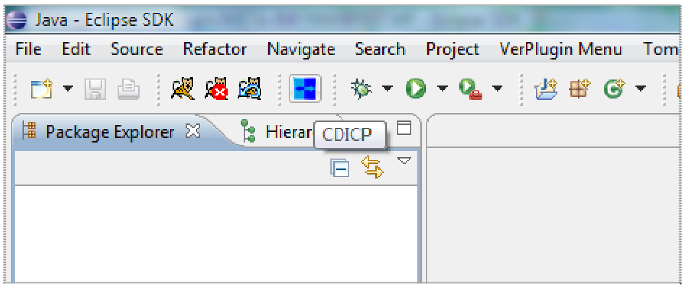

Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project VerPlugin Menu Tom
Package Explorer
Hierar
CDICP

Fig. 8. New plugin CDICP in Eclipse.

5 General Method for Detecting Cross-diagram Inconsistencies in UML Design

Analyzing only four most important at the design stage diagrams which are Class diagram, Sequence diagram, Object Diagram and State Machine diagram (in UML 2.0 specification) [13, 14] we defined more than 30 intra- and cross-diagram relations to be checked. None of well known CASE tools offers such checks. Using the simple method mentioned above is quite tedious as it supposes developing a special Checker (Fig. 8) for each relation, which in its turn requires multiple searches on XMI file. Even for a middle size project this file is quite big. The main idea here is to develop a special model of the system design for verification purposes (as usually done in verification methods), build it once by parsing XMI file, and then make all checks on this model. Moreover such model can be used for refining design on account of lessening couplings, strengthening cohesion and applying design patterns. All results of this model analysis regardless of the purpose take the form of recommendations so the corrective changes are up to the designer.

Thorough analysis of UML 2.0 specification led us to using graph representation of such model. It allows unified representation of all four diagrams by graphs with different types of vertices and edges. In this case checking relations between UML diagrams is just searching for the definite types of vertices or edges or their interconnections in the model. The first order predicate logic is used to formulate the relations leading to inconsistencies. In fact graph representation simplifies the description of diagrams comparing to their formal specification but is sufficient for verification purposes. For a class diagram the corresponding graph's vertices are classes and edges are connections between them which are association, dependency, generalization and interface realization. The information about generalization sets is stored separately to simplify search algorithms. For an object diagram the corresponding graph's vertices are objects and edges are connections between them. For a sequence diagram the vertices are objects and edges are messages between them. For a state machine (or state chart) diagram the vertices are states and edges are transitions between them. Each type of vertex and each type of edge stores information needed to check intra- and cross-diagram inconsistencies. Say an association of a class diagram as an edge of a graph keeps the name of the association, roles and multiplicities of its participants, etc. An example of the simple class diagram and its graph representation is given in Fig. 9. The edges of the graph represent different types of connections between classes and hence store different information.

Here is the formal representation of our model consisting of graphs of four types for Class, Object, Sequence and State Machine diagrams correspondently:

$$D = \{\{D_{cl}\} \cup \{D_{ob}\} \cup \{D_{seq}\} \cup \{D_{st}\}\} .$$

Each of these graphs consists of two sets: V stands for vertices and E stands for edges. Their description is given below.[1]

$$D_{cl} = \left\{ V_{cl}, E_{cl} \right\}$$

[1] Elements in [] are optional.

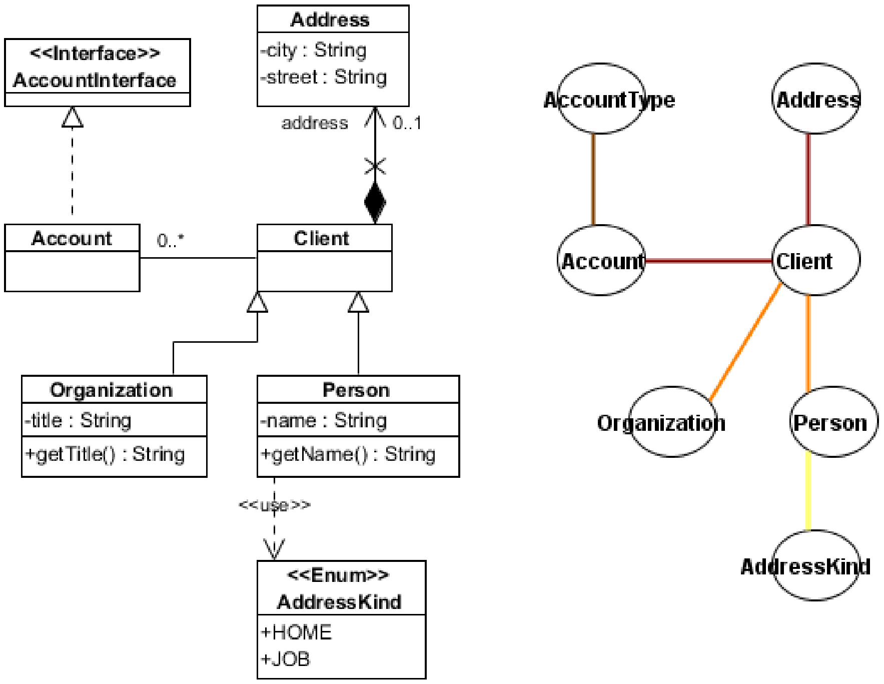


Fig. 9. Example of graph representation of a class diagram

$$V_{cl} = \{v : v = (name, isAbstract[, ATTR, MTHD, STRT, visibility])\}$$

$$ATTR = \{attr : attr = (name, domain, scope[, visibility, multiplicity])\}$$

$$MTHD = \{mthd : mthd = (mthdSgn, scope, visibility)\}$$

$$mthdSgn = (name[, PARAMS, returnDomain])$$

$$PARAMS = \{param : param = (num[, name], domain)\}$$

$$STRT = \{stereotype : stereotype = (name)\}$$

$$E_{cl} = \{e : e = (v_s, v_e, type[, info]); v_s, v_e \in V_{cl}, type = gen \mid ass \mid dep \mid impl\}$$

$$info = ([name, r_s, r_e, m_s, m_e, aggr_s, aggr_e, navig_s, navig_e])$$

$$D_{ob} = \{V_{ob}, E_{link}\}$$

$$V_{ob} = \{v : v = (name, clName[, ATTRVAL, STRT)\}$$

$$ATTRVAL = \{attrval : attrval = (name, value)\}$$

$$E_{link} = \{e : e = (v_s, v_e, name); v_s, v_e \in V_{ob}\}$$

$$D_{seq} = \{V_{cl} \cup V_{ob}, E_{msg}\}$$

$$E_{msg} = \{e : e = (v_s, v_e, msgCall); v_s, v_e \in V_{cl} \cup V_{ob}\}$$

$$msgCall = ([guard,]\, seqnum, mthdCall)$$

$$mthdCall = (name, ARGS[, returnValue])$$

$$ARGS = \{armnt : armnt = (num, value)\}$$

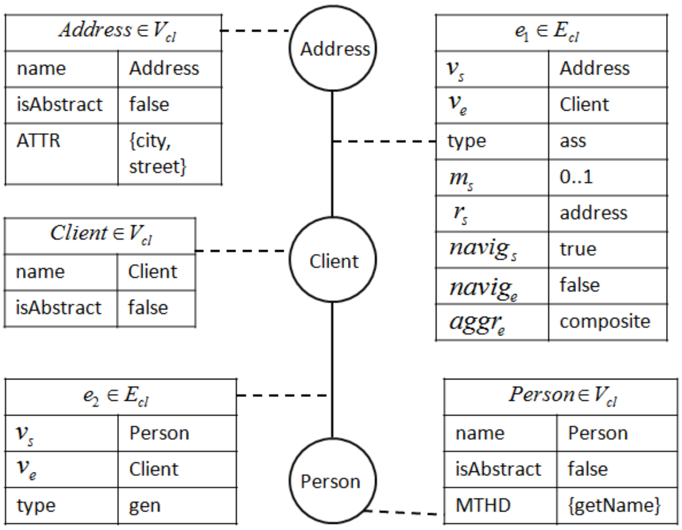
$$D_{st} = \{V_{st}, E_{tr}\}$$

$$V_{st} = \{v : v = (name, [, entry, do, exit]); entry, do, exit \in mthdCall\}$$

$$E_{tr} = \{e : e = (v_s, v_e, trCall); v_s, v_e \in V_{st}\}$$

$$trCall = ([guard,]\, mthdCall)\ .$$

An example in Fig. 10 illustrates information stored with some types of vertices and edges.



$Address \in V_{cl}$	
name	Address
isAbstract	false
ATTR	{city, street}

$e_1 \in E_{cl}$	
v_s	Address
v_e	Client
type	ass
m_s	0..1
r_s	address
$navig_s$	true
$navig_e$	false
$aggr_e$	composite

$Client \in V_{cl}$	
name	Client
isAbstract	false

$e_2 \in E_{cl}$	
v_s	Person
v_e	Client
type	gen

$Person \in V_{cl}$	
name	Person
isAbstract	false
MTHD	{getName}

Fig.10. Information stored in graph elements for the example above

The relation to be checked should be represented as the first order predicate logic formula. Propositional variables in this formula are the elements of the model above. Such unified approach to formulating the criteria of relations to be checked allows using the only Checker for any sort of relation.

Here is an example of such formula. It describes the fact that if the instance of one class sends the message to the instance of another class in the sequence diagram then the corresponding method should be among the methods of the latter class in the class diagram (Fig. 2 shows an example of this relation not satisfied).

$$(\forall e \in E_{msg} : v_e(e) \in V_{cl})$$

$$(\exists v \in genPath(v_e(e)))(\exists mthd \in MTHD(v(e))) : msgCall(e) \approx mthdSgn(mthd)$$

$$\wedge$$

$$(\forall e \in E_{msg} : v_e(e) \in V_{ob})$$

$$(\exists cl \in V_{cl})(\exists v \in genPath(v_e(e)))(\exists mthd \in MTHD(v(e))) : msgCall(e) \approx mthdSgn(mthd)$$

where

$$genPath(v) = v_1 ... v_n : v_1 = v \wedge (\forall i = 1, n-1)(\exists e \in E_{cl} : type(e) = gen \wedge v_s(e) = v_i \wedge v_e(e) = v_{i+1})$$

is introduced to take into account the fact that the method in question can be inherited along the path in the inheritance tree of the class.

At the moment the software tool for the proposed method is being developed and tested.

6 Conclusions

This paper offers a simple method for detecting inconsistencies between different UML diagrams. It was implemented as an Eclipse plug-in and tested on some types of cross-diagram inconsistencies.

Another more general method for checking inconsistencies in UML design is proposed. It uses the unified model with graph representation of the design components and formulae of the first order predicate logic to represent relations which should be satisfied to make the design consistent. This approach can also be used to evaluate the quality of a design and make recommendations on its improvement on account of better use of the main principles of the object-oriented design.

References

1. A. Andrews, R. B. France, S. Ghosh, and G. Craig.: Test Adequacy Criteria for UML Design Models. Journal of Software Testing, Verification and Reliability, 13(2), pp. 95--127 (2003)
2. Fischer. G. et al.: The Role of Critiquing in Cooperative Problem Solving, ACM Transactions of Information Systems, Vol.9, No.3, pp. 123--151 (1999)
3. Lionel Briand and Yvan Labiche: A UML-based approach to system testing. Software and System Modeling, 1(1), pp. 10--42 (2004)
4. Souza, C.R.B., et al.: Using Critiquing Systems for Inconsistency Detection in Software Engineering Models. In: Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2003), San Francisco Bay, pp. 196--203 (2003)
5. Souza, C.R.B., et al.: A Group Critic System for Object-Oriented Analysis and Design. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000), pp. 313--316 (2000)
6. S. Ghosh, R. B. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns: Test Adequacy Assessment for UML Design Model Testing. In: Proceedings of the International Symposium on Software Reliability Engineering, pp. 332--343, Denver, CO (2003)
7. Mara del Mar Gallardo, Pedro Merino, Ernesto Pimentelis: Debugging UML Designs with Model Checking. Journal of Object Technology, 1(2), pp. 101--117 (2002)
8. Martin Gogolla, Jrn Bohling, and Mark Richters: Validation of UML and OCL models by automatic snapshot generation. In: Proceedings of the 6th International Conference on Unified Modeling Language (UML'2003), pp. 265--279. Springer, Berlin, LNCS 2863 (2003)

9. Nilesh Kawane: Fault Detection Effectiveness of UML Design, Model Test Adequacy Criteria. In: Supplementary Proceedings of the International Symposium on Software Reliability Engineering, pp. 327--328, Denver, CO (2003)
10. Nilesh Kawane: EPTUD : An Eclipse plug-in for testing UML design models. Master's of science thesis, Colorado State University, Fort Collins, Colorado (2005)
11. Stephen Mellor and Marc Balcer: Executable UML: A Foundation for Model Driven Architecture. Addison Wesley Professional (2002)
12. T. Dinh-Trong, N. Kawane, S. Ghosh, R. B. France, and A. A. Andrews: A Tool-Supported Approach to Testing UML Design Models. In: 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2005), Shanghai, China, Proceedings (2005)
13. Object Management Group: UML 2.0 Superstructure Specification (2005), http://www.uml.org/
14. Pender. T.: UML Bible. Wiley Published Inc. (2003)