Verification of Good Design Style of UML Models

Bogumiła Hnatkowska[1]

[1]Institute of Applied Informatics, Wrocław University of Technology,
Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland
Bogumila.Hnatkowska@pwr.wroc.pl

Abstract. Software architecture, and its behavior can be expressed as UML models. Models of complex systems can be also complex and hard to read – they may consists of hundreds of artifacts. Analysis of such complicated models is very difficult. Applying design guidelines make this process easier. Design guidelines consist of some rules, and constraints that should be applied during models construction. They increase readability of models, and in consequence assure higher quality of the final product. The paper presents Model Verificator – the tool for checking UML models against a given design standard. The tool works as a plug-in to Rational Rose. The guidelines take the form of XML file compliant with some DTD. Verification is done fully automatically. The tool also enables correction of checked models.

Keywords: UML models, verification, design guidelines, good style, quality assurance

1 Introduction

Software development process results with many kind of artifacts. Beside a source code, system models are the most important artifacts built during system construction. UML is the most popular modeling language today, and UML models are used for presenting structure and behavior of the software product.

Models of a complex system can be also complex and hard to read – they may consists of hundreds of model elements. Model quality may be improved by applying different standards. Standardized documents characterize similar appearance and structure. Therefore they are more readable and understandable. The rules and constraints, associated with "document style", which should be applied by software engineers during model construction are usually gathered in design guidelines document also called design standard. The positive impact of using modeling conventions is obvious and was experimentally proven, e.g. [5].

Verification of a system model against a given design standard can be done in two ways: manually or automatically. Usually it is done manually using techniques like audits, walkthroughs or inspections. The paper presents an idea of automatic verification of UML model(s) against design style guidelines. The idea was proven by implementing Model Verificator tool [6]. Model Verificator works as a plug-in to

Rational Rose and checks models built within it. The usability of the tool was checked during Software System Design courses at Wrocław University of Technology.

The author found only one tool for automatic evaluation of modeling rules and design guidelines [2]. This tool uses the OCL for expressing modeling rules. The OCL is used in many tools, e.g. [1], [3], for checking inter and intra-consistency of UML models. As the solutions usual are opened, OCL could be also used in them for formulating constraints for modeling style. Rational Rose does not support the OCL directly. Existing Oclarity plug-in for Rose is commercial. So a person, who wants to use the OCL for checking the design style of UML models prepared in Rational Rose, needs to export the model to xml format (using e.g. Unisys plug-in) and then to and check such model in an OCL tool.

The role of design style guidelines is explained in Section 2. Section 3 presents assumptions about verification tool and the tool itself. Results of usability testing of Model Verificator are shown in Section 4. Some concluding remarks are gathered in Section 5.

2 Design Style Guidelines

Style guidelines are used in many areas of software engineering, e.g. human interface design (human interface guidelines) or coding (programming style). Programming style (coding standards, coding conventions) is defined as a set of ”conventions for writing source code in certain programming language” [10]. These conventions have a form of very practical hints that tell programmers how to code some elements of the program or what programming techniques to use. An example of such a popular coding style is the document ”Java Coding Conventions” [9], available free from the Internet. Human interface guidelines are used to provide a consistent look and feel. They embody good practices in interface design and increase the consistency between screens [8].

RUP methodology recommends to prepare project-specific guidelines [7]. Design guidelines are listed among the others (e.g. programming guidelines, or test guidelines). Project specific guidelines provide prescriptive guidance on how to perform a certain activity or a set of activities in the context of the project. Project specific guidelines are appropriate whenever there are project-specific standards that must be followed, or good practices that need to be communicated [7].

Within the paper the understanding of design guidelines is limited to the second meaning (standards or good practices that need to be followed and communicated). Design style guidelines play the same role as other kind of guidelines, but they focus other issues. They do not concern the source code, or user interface, but artifacts like specifications, models, etc.

Design style guidelines usually take a form of paper documents written in informal language. In such case the verification if an artifact fulfil the rules, included in design guidelines, must be done manually.

Design guidelines can be also used for templates. A template is defined as a predefined structure for an artifact [7]. Within Rational Rose there are many predefined system model templates, prepared for a given technology (e.g. J2EE, VB6) or

methodology (e.g. RUP). An exemplary system model created with RUP template is presented in Fig. 1. This model template defines the introductory model structure. The structure is divided into views and models, defined in RUP. There are 5 views and 6 models. There are some exemplary elements placed in different packages of model structure. The elements are commented with notes explaining their role.

The author's experiences show that templates are helpful, but not sufficient to preserve the uniform model structure. The template doesn't ensure the model as a whole is complete, their elements, e.g. models, are built according to the used methodology and called according to accepted naming convention. That was the rationale for elaborating the tool that was able to automatically check the models against some, user defined, modeling rules.

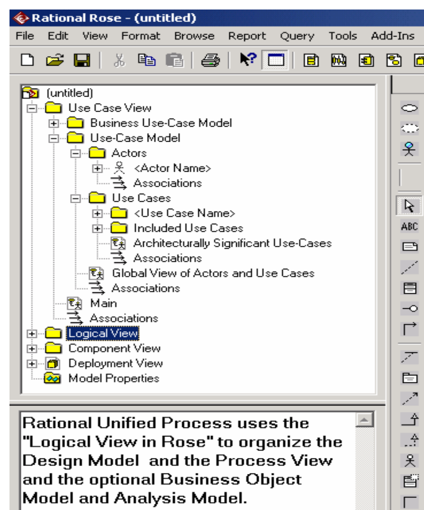

Fig. 1. A model created with RUP model template.

3 Model Verificator Tool

3.1 The Idea of the Tool

Model Verificator tool analyses Rational Rose models. The analysis is done in order to check the conformance of a model to some style guidelines. These guidelines are described as a set of rules written in a text file. Rules are written in a special language, presented in subsection 3.3.

The important feature of the solution is the ability of creating own style guidelines, suited for a given purpose. This concept follows the idea of Codespector tool [4]. Codespector aims to verify java source code against a certain coding standard, expressed with a specially designed CLR language.

The verification tool should be user friendly. The usage of the tool should be as easy as possible, and should not require knowledge about additional languages or technologies. Therefore the tool works as a plug-in to Rational Rose. Decision is more legitimate as the tool verificates models built within Rational Rose.

3.2 Verification Scope

The language, used for expressing modeling rules should be flexible enough:

- to set a naming convention for particular model elements,
- to set a proper structure of a model,
- to define right places for particular model elements,
- to define stereotypes that may be used in different places within model structure,
- to define allowed types of used diagrams,
- to define allowed multiplicity for particular model elements, e.g. at least one class diagram.

These requirements were formulated after analyzing of students' projects and catching on the most often appearing mistakes. They were defined from the point of view of a person who has to read, understand and check many models prepared by different design teams. In such situation the things like colors and shapes are of minor importance. The most important is to find elements of the same type in the same places in different models.

Model structure is strictly defined by RUP template. Different model elements should be carefully placed within the system model structure. This treats both to individual elements of UML language (e.g. system actors should be placed in Use-case model/Actors package), and UML diagrams (e.g. each use-case realization should be described by class and sequence diagrams).

Some modeling rules, written informally, which the author would like to express in the language, are presented below. All rules concern the contents of Analysis Model, defined by RUP.

Rule 1. *The analysis model should contain two class diagrams, first named "Architecture Overview – Packages", and second named "Key Abstractions".*

Rule 2. *The analysis model could be divided into packages (zero or more).*

Rule 4. *Each package could contain either use-case realization or analysis classes, and class diagrams.*

Rule 5. *Use-case realizations could contain only class diagrams (zero or more), sequence diagrams (zero or more), and at least one class diagram which name begins with* Traces.

Rule 6. *Analysis class should have one of stereotypes: <<control>>, <<boundary>>, <<entity>>.*

Rule 7. *The names' of class diagrams that are used to show traces from previous models, should begin with* Traces.

Rule 8. *The class diagrams showing the traces are allowed in: the root of Analysis Model, and use-case realizations.*

3.3 Rules Language Definition

The guidelines contain a set of rules stored in a text file. These guidelines have a form of Extensible Markup Language (XML) while rules definition language has the form of Document Type Definition (DTD). DTD constrains the proper structure of XML file. A part of DTD is shown on Listing 1.

Listing 1. A part of guidelines language definition, [5].

```
<!ELEMENT Guide (Model+)>
<!ELEMENT Model (Package| Any)>
<!ELEMENT Package ((Package| Class| UseCase| Diagram|
                   Component| Node)*| Any)>
<!ELEMENT Class ((Class| Diagram)*| Any)>
<!ELEMENT UseCase (Diagram)*>
<!ELEMENT Diagram EMPTY>
…
<!ATTLIST Model name CDATA #IMPLIED>
<!ATTLIST Package name CDATA "*" stereotype CDATA #IMPLIED count
CDATA "1">
<!ATTLIST Diagram
       type ( ActD| ClaD| ColD| ComD| DepD| SeqD| StaD| UseD)
       #REQUIRED name CDATA "*" count CDATA "1">
…
```

Diagram is the only obligatory model element. There are eight diagrams enumerated (all that may be constructed within Rational Rose). Other model elements are optional.

CDATA was chosen as a domain for most of language elements. The elements may have some attributes defined, i.e. name, number or stereotype. The name of a concrete element may contain wildcards, e.g. '?' or '*'. The star sign ('*") is a default name for each element. Stereotypes may be defined or may be omitted in guidelines definition. DTD defines the default multiplicity for any kind of language element, e.g. it is defined as 1 for a package.

Exemplary definition of style guidelines is presented on Listing 2.

Listing 2. Exemplary style guidelines.

```
<Guide>
<Model name="Use-Case Model">
  <Package name="Use Case View">
  <Package name="Use-Case Model">
    <Package name="Actors">
       <Class stereotype="Actor" count="1..n"> </Class>
    </Package>
    <Package name="Use Cases">
       <UseCase name="?*UC" count="1..n"> <Any></Any>
       </UseCase>
```

```
    </Package>
    </Package>
    <Diagram type="UseD" name="Main"></Diagram>
  </Package>
</Model>
</Guide>
```

Tag <Guide> is an obligatory start element. *Use-Case Model* is the only allowed model within system model. This model should contain two packages (*Actors* and *Use Cases*), and one use-case diagram named *Main*. *Use Case* package should contain at least one actor of any name. *Use Cases* package should contain at least one use-case, which name must be conformant to the "?*UC" pattern. A given use case may be described by any number of elements.

3.4 Tool Usage Example

As is mention above, Model Verificator was implemented as a plug-in to Rational Rose. Usage of the tool is very easy. In order to verify a model of a system against given design guidelines it is enough to chose *Execute Verification* option from *Tools* menu (see Fig. 2) and select a file with style guidelines. The tool automatically checks conformance of the model with style guidelines.

Exemplary results of verification of an model, created with RUP template (see Fig. 1), against the guidelines presented on Listing 1 are shown on Fig. 2.

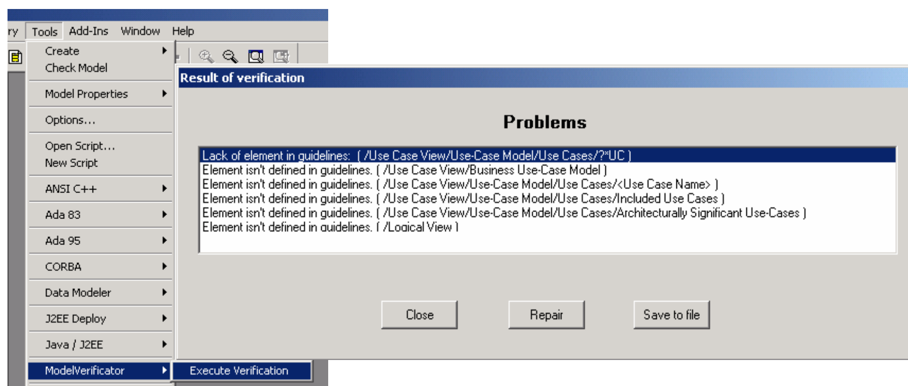

Fig. 2. Model Verificator usage example.

As it is easy to notice the model lacks with use-cases and has many elements not defined in the guidelines, e.g. Business Use-Case Model or Logical View.

As the tool implementation was limited to two Rational Rose views, i.e. Use-Case View and Logical View some violations of guidelines constraints were not recognized (Component and Deployment Views are not defined in guidelines).

There is a possibility to save verification report into a file or to perform some corrective actions. The correction is quite simple. It removes redundant or adds lacking elements to the model.

4. Usability Testing of Model Verificator

Before Model Verificator was implemented, recommendations about the proper model structure were formulated informally and communicated to students orally during the course. Students applied these recommendations in different degree.

In order to check Model Verificator usability, the author has prepared the file with design guidelines. The design guidelines consist of above 100 rules of different granularity. For example, all the rules, presented in section 3.2 are included in the guidelines.

The students were divided into two groups:

- the base group that applied the tool for verification of models,
- the control group that had prepared models, basing only on informal design recommendations.

The base group consisted of 8 teams (8 models to be checked), and the control group consisted of 4 teams. The author asked the students from base group for enumerating the most important mistakes, that were found by Model Verificator, and next corrected by the students. The students reported the following kinds of grievous errors:

- usage of improper diagram within a given context, e.g. the usage of use-case diagram for presenting the logical structure of the system;
- placement of model elements in wrong places (inconsistent with methodology), e.g. definition of business actors in Object Business Model or business workers in Business Use-Case Model;
- lack of required artifacts, e.g. lack of diagrams showing traces relationship to previous models.

The students from the base group managed with correction of reported mistakes, and in result they prepared models that satisfy all rules in design guidelines.

The models, prepared by students from the control group, were checked by the author. The errors, reported by the tool were the same kind like those, submitted by students. The number of reported errors oscillated from dozen to tens. For the worst model the tool reported above 70 violations (of 7 types) of design rules.

5. Conclusions

The paper presents an idea of automatic verification of UML models against design guidelines. The idea was proven by implementing Model Verificator tool. The tool works as a plug-in to Rational Rose. The rules of the standard are written in the special language. The guidelines take the form of XML file compliant with some DTD. Verification is done fully automatically. The tool also enables correction of models (deleting existing or introducing needed elements).

The idea isn't new – the same concept was the base of many tools checking source code conformance to a given coding standard. The fact that the Model Verificator works directly on Rational Rose models and need not their exporting to any other format is the novelty of the presented solution. The same guidelines may be reused for many projects. There may exists many guidelines in the same time.

The tool may be evaluated from two perspectives, i.e. design guidelines users, and design guidelines developers. The usability of the tool, from the users' point of view, was checked and confirmed by students of Wrocław University of Technology. The students reported the following advantages of the solution:

- better knowledge of used methodology, and produced models;
- increased readability, maintainability, and understandability of the models;
- improved communication between team members.

The main observed disadvantage is insufficient feedback information about wrong written elements, reported by the tool.

The author acted as a design guidelines developer. The language, used for writing modeling rules is easy and understandable (especially for computer science specialists). The only drawback is that the language does not offer the possibility for expression of recursive structure of the model, i.e. when the same rules should be applied recursive inside the model. In such situation all the rules need to be written once again. The guidelines documents are rather long, for example that one prepared for the purpose of the course has 160 lines (almost 7KB).

As the tool is helpful for its users in creation "well-formed" models its usage will be obligatory. Implementation of Model Verificator will be extended to include also Implementation and Deployment Models.

References

1. Chiorean, D., Pasca, M., Carcu, A., Botiza, Ch., Moldovan, S.: Ensuring UML models consistency using the OCL Environment. 6th International Conference on the Unified Modelling Language – the Language and its applications. San Francisco. Available at: http://i12www.ira.uka.de/~baar/oclworkshopUml03/papers/06_ensuring_uml_model_consistency.pdf (2003)
2. Farkas, T., Hein, Ch., Ritter, T.: Automatic Evaluation of Modelling Rules and Design Guidelines. The 2nd Worksop – From code centric to model centric software engineering: Practices, Implications and ROI. Spain. Available at: http://www.esi.es/modelware/c2m/docum/papers/PAPER2-ModellingRules-and-Guidelines.pdf (2006)
3. Hnatkowska, B., Walkowiak, A.: Consistency checking of USDP models, In: Understanding and usage of dependency relationships. International Workshop Consistency Problems in UML-based Software Development, Oficyna Wydawnicza PWroc (2004) 59–70
4. Jadach, M., Hnatkowska, B.: Codespector – A Tool for Increasing Source Code Quality. In: Zieliński, K., Szmuc T. (eds): Software engineering: evolution and emerging technologies, IOS Press, Vol. 130. (2005) 124–134
5. Lange, Ch.F.J., DuBois, B., Chaudron, M.R.V., Demeyer, S.: Experimentally Investigating the Effectiveness and Effort of Modeling Conventions for the UML. External research report. Technische Universiteit Eindhoven. Available at: http://library.tue.nl/csp/dare/LinkToRepository.csp?recordnumber=610229 (2006)
6. Muskała, P.: UML model verification against design style standards, Master Thesis; the work supervised by Bogumila Hnatkowska, Wroclaw University of Technology (2006)
7. Rational Unified Process, IMB Corp. 1987 (2005)
8. UsabilityNet: Tools and methods. Available at: http://www.usabilitynet.org/tools.htm
9. Sun Microsystems, Inc., Code Conventions for the Java Programming Language. Available at http://java.sun.com/docs/codeconv
10. The Free Dictionary. Available at: http://encylopedia.thefreedictionary.com