VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

A TOOL FOR CHECKING CORRECTNESS OF DESIGN DIAGRAMS IN UML

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE Bc. IVO DLOUHÝ
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO KONTROLU SPRÁVNOSTI NÁVRHOVÝCH DIAGRAMŮ V UML

A TOOL FOR CHECKING CORRECTNESS OF DESIGN DIAGRAMS IN UML

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE AUTHOR	Bc. IVO DLOUHÝ
VEDOUCÍ PRÁCE SUPERVISOR	RNDr. RYCHLÝ MAREK, Ph.D.

BRNO 2014

Nástroj pro kontrolu správnosti návrhových diagramů v UML
A Tool for Checking Correctness of Design Diagrams in UML
Supervisor: Rychlý Marek, RNDr., Ph.D., UIFS FIT VUT
Student: Dlouhý Ivo, Bc.
Assignment:

1. Seznamte se podrobně s možnostmi použití UML diagramů při návrhu software, možnostmi OCL, jejich podporou v dostupných nástrojích a přenositelnými formáty ukládání diagramů v UML.
2. Analyzujte a popište vyžadované či doporučené vlastnosti jednotlivých diagramů při správném návrhu software, možnosti detekce chybně navržených diagramů (vč. možností OCL) a způsobů jejich opravy.
3. Navrhněte nástroj, který umožní automatickou detekci chybně navržených diagramů v UML, uložených ve vhodném formátu, a navrhne uživateli-návrháři možnosti korekce nalezených chyb.
4. Po konzultaci s vedoucím navržený nástroj implementujte a otestujte na vhodných vstupech. Porovnejte automatické kontroly nástrojem s manuálním posouzením správnosti návrhových diagramů.
5. Výsledky zveřejněte jako open-source, zhodnoťte a navrhněte případná rozšíření.

The Term Project discussion items: Bez požadavků.
Category: Softwarové inženýrství
Programming language: dle výběru studenta, po konzultaci s vedoucím (preferována Java)
Operating system: nezávislé na OS
Commercial software: Visual Paradigm for UML (akademická licence)
Literature:

1. Russ Miles, Kim Hamilton. Learning UML 2.0. O'Reilly Media, 2006. ISBN 978-0-596-00982-3 http://my.safaribooksonline.com/0596009828.
2. Mira Balaban, Maraee Azzam, Sturm Arnon. Management of Correctness Problems in UML Class Diagrams Towards a Pattern-Based Approach. International Journal of Information System Modeling and Design, 1(4), 2010. http://www.cs.bgu.ac.il/~mira/incorrectness-patterns.pdf]

. . . .
Ivo Dlouhý
January 30, 2014

Abstrakt

Cílem semestrálního projektu je vytvořit nástroj pro kontrolu správnosti návrhových diagramů v UML zvláště diagramu tříd. Hlavními částmy práce je vytvoření databáze vzorů chyb, samostatné aplikace použitelné z příkazové řádky a zásuvného modulu pro rozšířený UML návrhový software Visual Paradigm. Důraz je kladen na otevřenost a rozšiřitelnost software. Po analýze problému chybovosti návrhových diagramů, podobných nástrojů a možných technologií je vytvořen návrh logiky. Na základě návrhu bude implementována aplikace využívající databázi vzorů nesprávnosti specifikovaných QVTr transformacemi ke kontrole správnosti UML diagramu.

Abstract

Aim of this term project is to create a tool for checking correctness of design diagrams in UML. Main task is to create a pattern database, independent command line application and a plugin for UML design software Visual Paradigm. Openness and extensibility of the software is important. At first UML Design Diagram correctness, similar tools and possible technologies are analyzed and design is created. Based on the design application will be implemented using incorrectness pattern database that consists of QVTr transformations to check the correctness of the UML Diagram. Application will be implemented and tested based on design.

Klíčová slova

správnost uml, návrhové diagramy, visual paradigm, xmi, uml, ocl, qvtr

Keywords

uml correctness, design diagrams, visual paradigm, xmi, uml, ocl, qvtr

Citace

Ivo Dlouhý: A Tool for Checking Correctness of Design Diagrams in UML, semestrální projekt, Brno, FIT VUT v Brně, 2014

A Tool for Checking Correctness of Design Diagrams in UML

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením RNDr. Marka Rychlého Ph.D. a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

I hereby declare that I developed this term project individually under the supervision of RNDr. Marek Rychlý Ph.D. and I stated all literature and publications I used in this project.

....
Ivo Dlouhý
January 31, 2014

Contents

1 Introduction 3

2 Analysis 5
- 2.1 Standards . . . 5
  - 2.1.1 Object Management Group Standards . . . 5
  - 2.1.2 OCL . . . 5
  - 2.1.3 QVT . . . 6
- 2.2 Tools . . . 6
  - 2.2.1 Visual Paradigm - Model Quality Checker . . . 6
  - 2.2.2 AgroUML . . . 6
- 2.3 Goals . . . 7

3 Design 9
- 3.1 Use Cases . . . 9
- 3.2 Activity diagram . . . 10
- 3.3 Input . . . 10
- 3.4 Pattern Database . . . 10
- 3.5 Output . . . 10

4 Implementation 14
- 4.1 Distribution and Documentation . . . 14

5 Testing 15

6 Evaluation 16

7 Conclusion 17

Bibliography 17

A XMI Format Example 19

List of Figures

2.1 OCL Example 8
2.2 Visual Paradigm Model Quality Checker 8
2.3 Visual Paradigm Model Quality Checker problems [8] 8
2.4 ArgoUML Critic 8

3.1 Use Case diagram - User roles 12
3.2 Use Case diagram - Developer role 12
3.3 Activity diagram 13
3.4 Class diagram - Pattern storage 13

A.1 XMI Example 20

Chapter 1

Introduction

Software design and modeling plays a key role in every software life cycle. The standards and languages and diagrams used for modeling purposes are getting more and more well defined, but they need to be general enough to allow user to model every possible scenario. This ambiguity can cause incorrect usage of the modeling language in cases when the designer is not experienced enough or project is simply too complex to design. Design phases is indisputably the most important part of the software development and heavily influences following phases. Every mistake made in design can have an deep impact on the result software so it is desirable for it to be as correct as possible.

Most widely used tool in the software design area is the UML language represented by the UML Class Diagram in this thesis. The Class Diagram consists of mainly of Classes as a basic tool of Object Oriented design and Associations amongst the Classes. There can be several types of Associations denoting different semantics and it is important that they fit together perfectly to make a working and usable Class Diagram. Most of the rules are controlled by the UML standard however there are still several bad practices or incorrect usage that leads to non valid design. These mistakes in the design can be described in a form of patterns sometimes called incorrectness patterns. Tool created in this thesis should detect these patterns and help user to deal with them. The reader of this text is expected to have basic to advanced knowledge about Object Oriented Software Modeling and UML Diagrams, specifically UML Class Diagram.

Chapter 1 of this thesis is the Introduction. It explains the project background, motivation and contribution the project should make to the modeling community. This should introduce the reader to further chapters and help him understand the problem this thesis solves. Chapter 2 contains introduction to the problem area, basic description of the standards suitable to complete the task and describes similar solutions. Based on problem analysis a tool to check UML correctness is designed in chapter 3. Based on Use Cases all key components of the tool are described and all functional requirements for the complete problem solution are designed. Chapter 4 Implementation focuses on implementing the solution previously designed. It also discusses the documentation and deployment of the tool. Chapter 5 Testing proposes test methods that will be used to test the future tool implementation from different points of view. Chapter 6 Evaluation describes important aspects of the application that need to be evaluated and taken into account so that in can be useful to the modeling community. Chapter 7 Conclusion is the final chapter of the thesis. It sums up the whole thesis based on the solution of the goals defined in Analysis. The future development of the tool and this project is outlined. Appendices contain additional information that do not fit in the mail text, such as code examples.

Chapter 2

Analysis

This chapter will contain a general introduction into the problem of UML correctness where will be relevant standards, languages and tools presented.

2.1 Standards

2.1.1 Object Management Group Standards

The Object Management Group (generally referred to as OMG) is a international, non profit computer industry standards consortium. Standards created by OMG allow visual design, execution and maintenance of software and other processes [7]. Relevant standards are UML, XMI and QVT are covered in following sections.

2.1.2 OCL

Object Constraint Language (referred to as OCL) is another one of OMG standards. OCL is a declarative language that can describe rules that will be applied on UML models so it is a part of the UML standard. It was generalized so that it can be used on the MOF based models. Main purpose of this language is to provide a constraint and object query expressions that can not be expressed with diagram notation.

OCL Constraint

The OCL consists of a set of OCL Constraints. OCL Constraint is basically an OCL expression that results in true or false. The expressions use UML Class Diagram naming - classes in constraint can be addressed by their names, navigation through the diagram is possible by using attributes or roles.

Main parts of each OCL Expression is Context and Invariant. Context specifies the element OCL Expression should be evaluated for. Another important part of OCL are collections, because they allow to work with elements in general.

Here you can find a simple OCL Expression that checks, that for UML Class Diagram at 2.1 Ancestor and Descendant cannot be the same person as the Person OCL is evaluated for (variable self)

```
context Person
inv: ancestors−>excludes(self) and descendants−>excludes(self)
```

As shown in example, OCL is most generally applied to enforce rules for the UML models. However OCL is applied to objects, not the UML Diagram itself. Next step of OCL evolution is the QVT described in 2.1.3.

2.1.3 QVT

QVT is an abbreviation for Query/View/Transformation and as a title indicates, it is a set of languages used for model transformation. As other similar software design standards it is also defined by OMG **??** [6]. Together the languages are capable of Model transformation, important part of Model-Driven Architectures. All languages operate on MOF 2.0 compliant metamodels, such as UML 2.0. The QVT standard also includes OCL 2.0 2.1.2 and extends it with imperative features. That makes QVT more suitable of checking UML correctness.

QVT Operational is first of the QVT languages and covers imperative features capable of unidirectional transformations. QVT Relations is a declarative language that implements model transformations. The transformations can be also run in check only mode, that validates, if the model is consistent according to the transformation. Finally, QVT Core is declarative language that is a target of QVT Relations language.

All of the UML correctness patterns can be specified in this language so it can be efficiently applied to check the UML model correctness similar to approach in [4] and [3]. However the QVT standards are very universal and complex and as such, it is not supported by many tools or libraries. However QVT Relations is implemented by a few tools or libraries that can be used.

2.2 Tools

2.2.1 Visual Paradigm - Model Quality Checker

Visual Paradigm software includes a tool that identifies design problems in the UML diagram [8]. User is presented with problem diagram elements based on the point system. Each problem is assigned a point value and when the sum for a an element exceeds the threshold it is considered as a problem element. A problem can be fixed based on a suggestion or ignored. However the tools checks just for basic problems as you can see in 2.2.1. This fact suggests, that an addition of advanced uml correctness analysis tool would be worthy.

Plugins

Visual Paradigm allows users to extend the functionality by plugins. Plugin is a set of one or more functions implemented in Java [9]. Visual Paradigm Plugin Support [10] exposes an interface for developers and allows processing of model diagrams and diagram elements and invoking of internal functions. Plugin consists of 4 basic components: Model, Diagram, Diagram Element and Action with Action Controller and a plugin.xml metadata file.

2.2.2 AgroUML

AgroUML is . . . Nada offers critique tool, that runs in background and checks for wrong patterns in the model, as described in [2] [1]

2.3 Goals

Goal of this thesis is to implement a tool to check correctness of an UML Diagram, specifically Class Diagram. The core of the tool will work based on the database of bad practices and incorrectness patterns. Input will be an UML diagram and output a log file. The tool should be implemented as a command line application and also as a plugin in the modeling software, preferably Visual Paradigm.

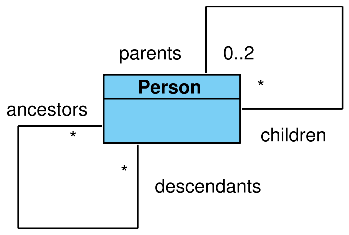
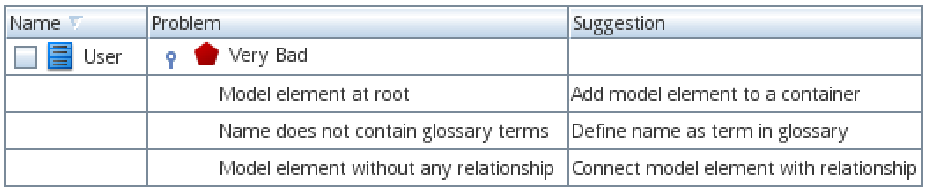


Figure 2.1: OCL Example

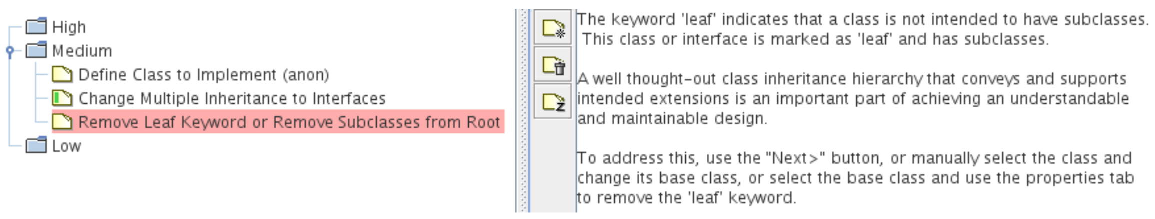


Name	Problem	Suggestion
User	Very Bad	
	Model element at root	Add model element to a container
	Name does not contain glossary terms	Define name as term in glossary
	Model element without any relationship	Connect model element with relationship

Figure 2.2: Visual Paradigm Model Quality Checker

Problem
Model element at root
Name does not contain glossary terms
Non-blackbox pool with no lane nor contained shapes
Task has more than one outgoing flow
Model element without any relationship

Figure 2.3: Visual Paradigm Model Quality Checker problems [8]



- High
- Medium
  - Define Class to Implement (anon)
  - Change Multiple Inheritance to Interfaces
  - Remove Leaf Keyword or Remove Subclasses from Root
- Low

The keyword 'leaf' indicates that a class is not intended to have subclasses. This class or interface is marked as 'leaf' and has subclasses.

A well thought-out class inheritance hierarchy that conveys and supports intended extensions is an important part of achieving an understandable and maintainable design.

To address this, use the "Next>" button, or manually select the class and change its base class, or select the base class and use the properties tab to remove the 'leaf' keyword.

Figure 2.4: ArgoUML Critic

Chapter 3

Design

Software design usually starts with use cases formulation based on the requirements extracted in analysis step.

3.1 Use Cases

The use cases are a way of expressing the functional requirements of a system [5] and as such, it is a starting point for further software design and more UML diagrams.

Usually we start implementing the Use Case diagram with role identification. As stated in Goals section 2.3 in Analysis chapter 2, the application should have two user interfaces - a command line (CLI) and a graphical (GUI).

In Use Case diagram 3.1, this fact is be represented by two roles - Command Line Tool User and GUI Tool User. Most of the abstract functionality is the similar for both interfaces, so most of the Use Cases are associated with generalized Tool User.

Use Cases explained:

- View pattern detail - display all information about a specific incorrectness pattern.
- Update the pattern database - Download updated incorrectness pattern database over the intranet.
- Set output verbosity - User is able to set the level of output logging by setting a minimal severity.
- Display manual - User is able to view help on how to work with the application.
- Ignore patterns - User is able to filter the patterns, that are applied in analysis stage.
- Ignore UML elements - User is able to set UML elements, that are ignore in analysis stage.
- Set input file - User is able to set any XMI diagram file.
- Install/Update the plugin - When considering Eclipse integration, user is able to install/update the plugin through standar eclipse interface.
- Set input model - User is able to choose model, that would be analyzed.

In second part of the Use Case diagram 3.2, you can find role Tool Developer. This represents a role of application developer. During the application lifecycle, developer (or maintainer of the UML incorrectness pattern database in general) should be able to update the pattern database by adding, removing or editing a record and publish the updated database for users to download.

3.2 Activity diagram

Activity diagram 3.3 will demonstrate an application workflow.

3.3 Input

Application performs analysis on a model so model will be the main input of the application. Other form of input will be only application settings - set by config file and/or arguments in CLI application and set in Preferences in GUI mode.

The Model is loaded from XMI file standardized by a XML schema. Application can utilize the XML schema to validate the input and display errors in formatting. This simplifies the parsing of diagram elements, because application can rely on the XMI being well formed.

3.4 Pattern Database

Core part of the tool is the Pattern Database which stores UML incorrectness patterns and their metadata. The database should allow updating the database with new patterns and thus adding new features to the application. Another important feature is interchangeability of the database - patterns can be used by another applications, presented on web or printed to documentation. As described in previous chapter Analysis 2, XML satisfies both requirements. Incorrectness pattern storage is designed in Class diagram 3.4. Root element is Patterns - it contains just basic metadata and all other pattern elements. Each pattern elements consists of

1. **Identification** - attributes name and id
2. **Incorrectness Pattern** - severity, description, code and fix.

This Class diagram clearly describes the XML database design and will simplify implementation of XML Schema.

3.5 Output

The output of the application are incorrectness found in the UML Diagram. Command line application will output the logs to standard output. Log will contain problem name, severity information, identification of problem elements and a hint on how to fix the problem. The output should be well formatted, so that user can process it automatically by a wrapper application or just redirect it to a log file and read later.

Application with user interface will be implemented in a form of a plugin. Plugin will use user interface of the master application to present the analysis results in a form of table. Table will contain the same result set of information as command line application and it should also allow user to navigate to problematic UML Diagram elements if the master application allows this functionality.

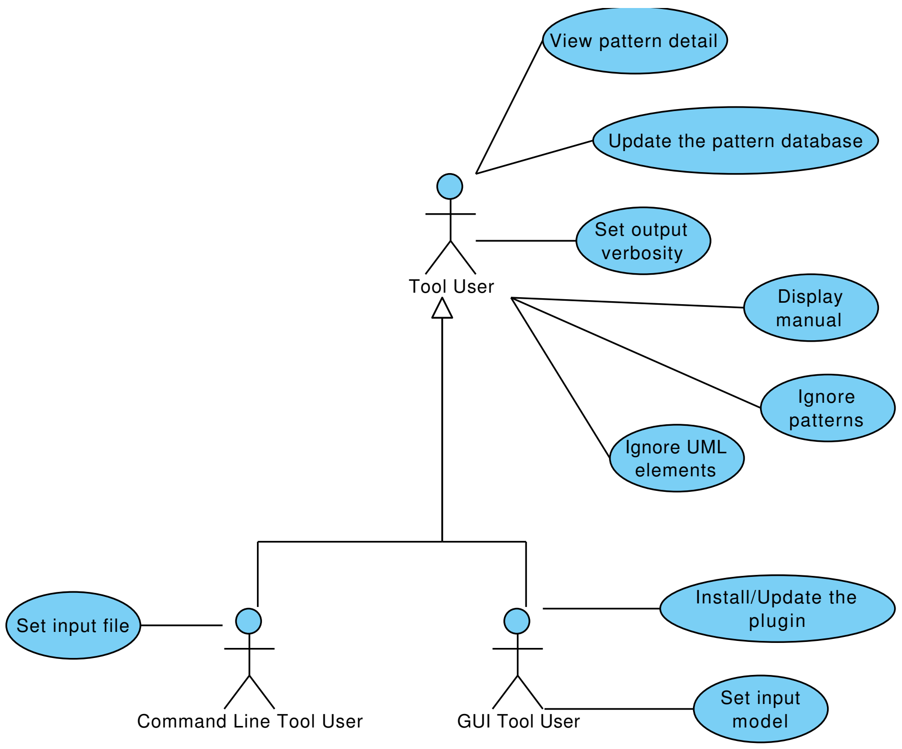


Figure 3.1: Use Case diagram - User roles

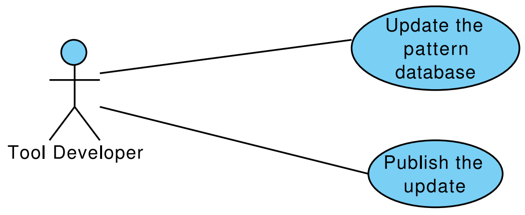


Figure 3.2: Use Case diagram - Developer role

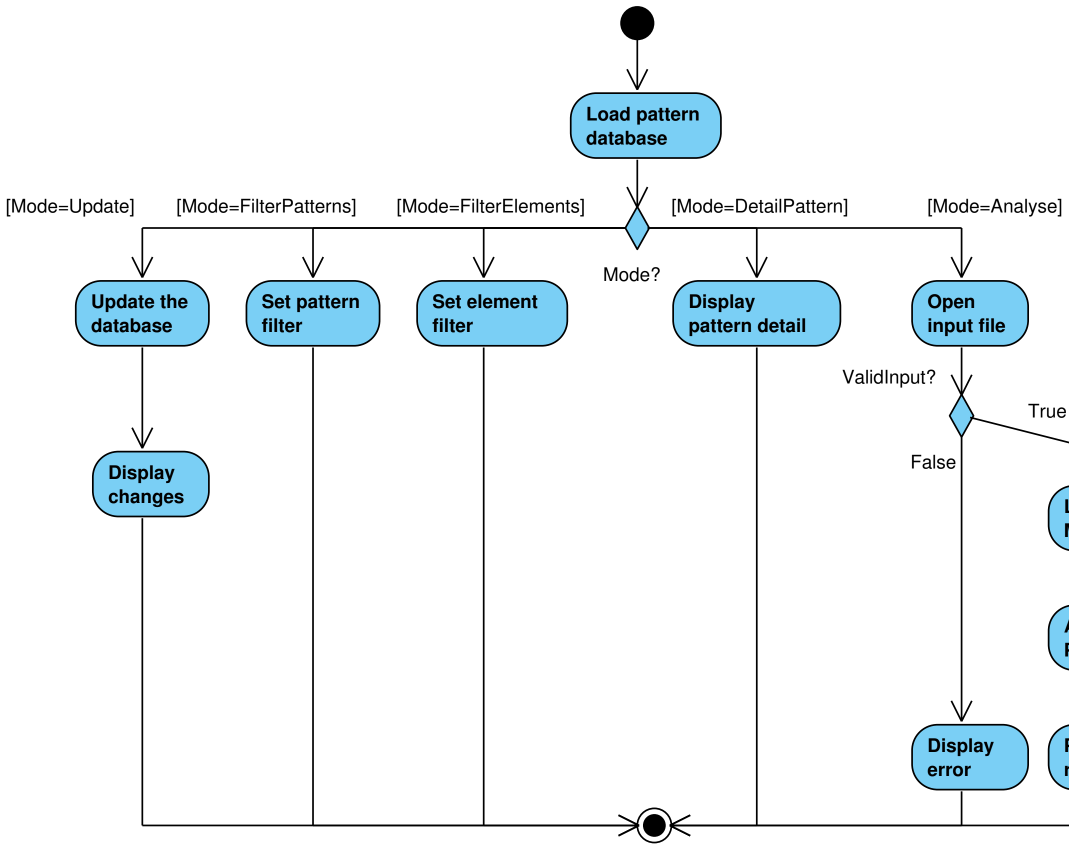


Figure 3.3: Activity diagram

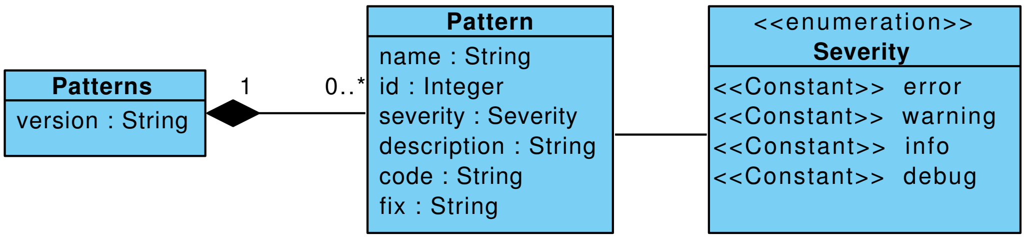


Figure 3.4: Class diagram - Pattern storage

Chapter 4

Implementation

4.1 Distribution and Documentation

The main source of documentation is this report. It will cover both theory behind the application as well as information about implementation and design. Each of the tools will also have a readme file and installation manual. Instructions on how to download and install the files will be available online.

Important part of application nowadays is a distribution. The implemented tool should be downloadable online including installation instructions so that users can easily download and use it. Command line application with all support content will be distributed in archive file and available to download. Both Visual Paradigm and Eclipse have specific requirements for plugins. Visual Paradigm plugin is installed by unarchiving to specific folder and needs to have strict file structure and naming. Plugins for Eclipse are distributed through update sites or Eclipse Marketplace. Pattern database may be distributed separately and will be also downloadable directly from the application. This also enables the database to be updated during application lifetime.

Chapter 5

Testing

This chapter covers testing of the tool. Both application and pattern database components need to be tested for issues so that it can be ensured that it will work properly. Important consideration is also the use of automated testing since the pattern database can be too sizable to test manually. Most of the testing will be realized by a set of input UML diagrams.

There will be used several kinds of tests

1. Test for each incorrectness pattern - several small UML Diagrams
2. Overall test for all patterns in general - one big UML Diagram
3. Application test - different formats of XMI
4. Plugin test

Chapter 6

Evaluation

This chapter evaluates the results of the thesis from multiple points of view. First aspect of evaluation is performance and can be measured by how effective is the tool in detecting collected incorrectness patterns, how many can the tool work with, how many can be automatically fixed Second aspect is the tool user point of view. User should be notified about every failure, so that he is able to fix it. Application controls should be clear. Documentation should be clear and available so that users can learn how to work with the tool. Installation of the software should be easy enough so that users are not discouraged to try it. And finally, the last aspect is the comparison to other available tools and contribution of this application to the traditional Diagram Design workflow.

Chapter 7

Conclusion

In the initial part of this thesis called the *term project* the main task was to understand the problem of correctness of UML Design Diagrams, find and evaluate existing tools and identify main goals of this project. Design of application was created based on the analysis of the problem.

Next milestone in project development becomes the more detailed part of the design so that all aspects of the solution are taken into consideration. It is also possible, that more technologies will need to be analyzed so that the solution will meet all requirements described in Goals and Evaluation section. When the design is complete, implementation of the final solution can start. As stated in the testing chapter, it is important to test both the functionality of the tool and the patterns properly so that it produces correct results. The test set should be continuously improved and automation of test should be considered.

The biggest challenge this thesis faces is to choose the right approach to ensure that the diagram is correct. Using a method that is not generally applicable and fails to detect all patterns can render this project unusable. Another difficulty was to find relevant information. The UML Language and Class Diagrams are widely used, but the same cannot be said for OCL because even some market leading software design tools does not support it and for QVT there is just a handful of existing implementations and some of them do not even support the whole standard.

Provided the core of the application and technologies were chosen correctly an effective, extensible and usable application will be build. If it works as expected it can assist software designers in the process of creating a Class Diagram, improve quality of the result and event validate diagrams by checking for incorrectness patterns and bad practices. This tool can also be used for educational purposes and help to learn how to create a correct UML Diagram from the beginning.

Bibliography

[1] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett, Linus Tolke, and Michiel van der Wulp. *ArgoUML Manual*, 0.34 edition, 2011.

[2] Jason Elliot Robbins. *Cognitive Support Features for Software Development Tools.* PhD thesis, University of California, Irvine, 1999.

[3] Maged Elaasar, Briand Lionel, Yvan Labiche. Specification and Detection of Modeling Patterns: an Approach based on QVT. Technical report, 2010.

[4] Maged Elaasar, Briand Lionel, Yvan Labiche. Domain-Specific Model Verification with QVT. In RobertB. France, JochenM. Kuester, Behzad Bordbar, and RichardF. Paige, editors, *odelling Foundations and Applications*, volume 6698 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2011.

[5] Martin Fowler. *UML Distilled (Third Edition).* Addison-Wesley, 2004.

[6] Object Management Group. OMG QVT, August 2011.

[7] Object Management Group. About the Object Management Group, August 2013.

[8] Visual Paradigm. Identify and Fix Design Issues with the Model Quality Checker. [Online; accessed 2013-12-25], July 2011.

[9] Visual Paradigm. VP-UML Plug-in Development. [Online; accessed 2013-12-21], September 2011.

[10] Visual Paradigm. User's Guide - Introduction to plugin support. [Online; accessed 2013-12-23], December 2013.

Appendix A

XMI Format Example

In this appendix there is an example of simple UML Class Diagram A.1 exported to XMI format. Verbosity of XMI output is different for each modelling tool - Eclipse produces cleanest and most compact output, other tools such as Visual Paradigm uses richer XML and also utilizes xmi:Extension elements to store internal data. This example is exported from Visual Paradigm using XMI 2.1 and UML 2 settings. Note: Generated IDs were replaced by short integers for demonstration purposes.

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20110701" xmlns:xmi="http://www.omg.org/spec/XMI/20110701"
    xmlns:uml="http://www.eclipse.org/uml2/4.0.0/UML" xmi:id="_014" name="model">
  <packagedElement xmi:type="uml:Class" xmi:id="_004" name="Department">
    <ownedAttribute xmi:type="uml:Property" xmi:id="_007" name="name"/>
  </packagedElement>
  <packagedElement xmi:type="uml:Class" xmi:id="_012" name="Employee">
    <ownedAttribute xmi:type="uml:Property" xmi:id="_008" name="name"/>
  </packagedElement>
  <packagedElement xmi:type="uml:Association" xmi:id="_001" name="WorksIn" memberEnd="_002␣
      _003">
    <ownedEnd xmi:type="uml:Property" xmi:id="_002" name="employee" type="_011"
        association="_001">
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_009" value="2"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_005" value="*"/>
    </ownedEnd>
    <ownedEnd xmi:type="uml:Property" xmi:id="_003" name="department" type="_004"
        association="_001">
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_010" value="1"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_006" value="1"/>
    </ownedEnd>
  </packagedElement>
</uml:Model>
```

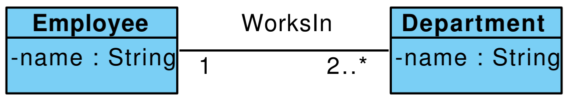


Figure A.1: XMI Example