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

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. IVO DLOUHÝ
AUTHOR

BRNO 2014

NÁSTROJ PRO KONTROLU SPRÁVNOSTI NÁVRHOVÝCH DIAGRAMŮ V UML

A TOOL FOR CHECKING CORRECTNESS OF DESIGN DIAGRAMS IN UML

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. IVO DLOUHÝ
AUTHOR

VEDOUCÍ PRÁCE RNDr. RYCHLÝ MAREK, Ph.D.
SUPERVISOR

BRNO 2014

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2013/2014

Zadání diplomové práce

Řešitel: **Dlouhý Ivo, Bc.**

Obor: Informační systémy

Téma: **Nástroj pro kontrolu správnosti návrhových diagramů v UML**
A Tool for Checking Correctness of Design Diagrams in UML

Kategorie: Softwarové inženýrství

Pokyny:

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.

Literatura:

- Russ Miles, Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, 2006. ISBN 978-0-596-00982-3 [http://my.safaribooksonline.com/0596009828]
- 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]

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT

Datum zadání: 1. listopadu 2013

Datum odevzdání: 28. května 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem diplomové práce je vytvořit nástroj pro kontrolu správnosti návrhových diagramů v UML zvláště diagramu tříd. Práce popisuje jazyk UML a souvisejících standardy, definuje problém správnosti UML a vysvětluje přístup kontroly správnosti UML pomocí databáze vzorů nesprávnosti. Dále navrhuje technologii QVT vhodnou pro implementaci vzorů pro kontrolu správnosti.Problém je rozdělen na více částí, mezi které patří sdílená databáze vzorů chyb v UML spravovatelná přes webové rozhraní, samostatný nástroj pro použití z příkazové řádky a zásuvný modul pro UML návrhový software Visual Paradigm. Všechny navržené části jsou navrženy, implementovány, otestovány a vyhodnoceny. Důraz je kladen na otevřenost a rozšiřitelnost nástroje.

Abstract

Aim of this master's thesis is to create a tool for checking correctness of design diagrams in UML. The work describes the UML language and connected standards, defines the problem of UML correctness and explains the approach of using incorrectness pattern database to check the UML correctness. Furthermore it suggests the QVT language as a suitable for implementing the incorrectness patterns. The problem is decomposed into shared incorrectness pattern database manageable via web interface, standalone tool for use from the command line and a plugin for the UML design software Visual Paradigm. All of the components are designed, implemented, tested and evaluated. The important aspect is the openness and extensibility of the tool.

Klíčová slova

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

Keywords

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

Citace

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

A Tool for Checking Correctness of Design Diagrams in UML

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci 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 master's thesis individually under the supervision of RNDr. Marek Rychlý Ph.D. and I stated all the literature and publications I used in this project.

....
Ivo Dlouhý
May 28, 2014

Contents

List of Figures

Chapter 1

Introduction

Software design and modeling plays a key role in every software life cycle because every error made in design has a deep impact on the final quality of the software. The later the design error is found, the more expensive is to fix it. Therefore it is desirable that design is as correct as possible.

Recent modeling standards and languages are well defined, but they still need to be general and flexible enough to allow user to model every imaginable scenario. This ambiguity can cause incorrect usage of the modeling language when the designer is not experienced enough or project is simply too complex to design.

Unified Modeling Language diagrams are essential part of the software design since they are more clearer and easier to understand than the problem specification and description in the plain text. This thesis investigates the correctness of the UML diagrams, namely the class diagram that denotes classes and their associations. There are several types of association with different semantics and different rules to use. Most of these rules are defined by the Unified Modeling Language standard. However there is still a chance for incorrect usage within the standard. Therefore this work is aimed at creating a tool to support software designers and prevent them from using UML incorrectly.

The analysis chapter covers most of the topics that need to understood before the problem becomes clear. However the reader is expected to have basic knowledge about Object Oriented Software Modeling, Unified Modeling Language, UML Class Diagrams and XML.

The usual software lifecycle consists of following phases: analysis, design, implementation and evaluation. This thesis follows the same pattern in its chapter organization. Chapter 1 of this thesis is the Introduction. It explains the project background, motivation and contribution the project will make to the modeling community. The Introduction brings the reader to further chapters and help him understand the problem this thesis solves. *Analysis* in chapter 2 describes the UML language and connected standards, defines the problem of UML correctness and explains the approach of using incorrectness patterns that will solve the problem. The chapter ends with setting goals for this project. The chapter 3 deals with the *design* of the tool. It forms the use cases according to the project goals and analysis. The problem is decomposed into shared incorrectness pattern database manageable via web interface, standalone tool for use from the command line and a plugin for the UML design software Visual Paradigm. All of the components are designed using UML diagrams. The *implementation* covered in chapter 4 focuses on implementing all of the components designed in the previous chapter. It also discusses the documentation and the deployment of the tool. Chapter 5 proposes *testing* methods and approaches that will ensure, that the tool was implemented correctly. Mix of functional and system tests will be needed so that the tool is tested from different points of view and good quality can be achieved. Chapter 6 *evaluates* the tool from several important aspects of quality. These aspects need to be evaluated and the quality of the tool is examined on the reference diagram. Based on the results the chapter points out the possible use of the tool and suggests the extensions that would be good to implement in the future. *Conclusion* in chapter 7 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. Additional information that are benefitial to the main part of the thesis are positioned in the Appendices.

Main information source for this thesis is the published research concerning UML correctness evaluation with the use of patterns. The most notable research is the catalog of metamodeling anti-patterns by a group of researchers from Carleton University [11]. Other great source is the research of BGU Modeling Group at Ben Gurion University of the Negev [3]. This project finds a way to bring this research in software modeling field and UML correctness evaluation using patterns into practice so that it assist software designers in their every day tasks.

The work on topic of software design will inevitably face some issues. The most apparent of the is the number of standards, languages and their revisions and applications that exist in the world of software design. The standards were created so that they introduce interoperability, however some UML design tool vendors often do not support them completely or just support specific versions. This thesis tries to address these issues with extensible design and allows developers to create their own user interface and integrate the tool into existing system. Another step towards knowledge unification is to allow users to share the validation data. This thesis also tries to be useful as a reference and additional research therefore it is written in English.

The finished project will be able to assist software engineers during their modeling work will benefit to the result software quality and improve the overall software modeling experience.

Chapter 2

Analysis

The analysis is important to correctly grasp the problem so that the best available tools are used to achieve quality results. The information found out in this chapter serve as a foundation for design that directly influences the whole project.

First of all, the chapter introduces the foundation standards and languages that serve as a base for the rest of the analysis - XML that will carry the data with XML Schema for XML validation, UML language with focus on UML Class Diagram and interchangeability via XMI. Based on this knowledge the chapter introduces the terms UML correctness and incorrectness patterns that are key for this project. These terms are the main contribution of the analysis step since the rest of the work is formed on top of them. The chapter continues with more standards and languages that are more specific for the task that lays ahead - first of all the OCL, followed with QVT that is a part of the final solution. Analysis follows with overview of the tools that will help along the way and continues with description of technologies and testing methods that are used in the later chapters of this work, namely Implementation in chapter 4 and Testing in chapter 5. The analysis chapter finally wraps up with the definition of the goals of this thesis.

2.1 Extensible Markup Language

Extensible Markup Language or XML is a flexible markup language that defines a set of rules for encoding data to interchangeable format. The XML language is standardized [1] by the World Wide Web Consortium W3C.

The principle is that the data are wrapped in specialized XML markup that describes their data structure. Thanks to the markup, the XML becomes readable for machines, however it still remain readable for humans as well. The basic expression element of the markup is tag. Each tag can have a set of unique attributes. Tags are organized into a tree structure with one root element. The valid XML document consists of the XML header and one root element.

A XML document validation has two steps: a well formed and valid document. A XML document is considered *well formed* it it complies with the XML standard, tags are well nested and it has one root element. A well formedness is usually checked during parse of the document - if the document is not well formed, it can not be parsed at all. A *valid* XML document needs to conform to a specific schema type definition that describes order and types of tags that can be present so that the document follows the correct structure.

[1]http://www.w3.org/XML/

The validity can be enforced by setting document type definition that should be checked during parsing as well.

Working with XML documents is mostly done using premade XML parsers. In most programming languages, XML parsers are part of the basic standard library or they can be easily included. Most well known approach to manipulate with XML is the Document Object Model or DOM. After validation, a XML parser with DOM interface parses the input file and constructs the XML document tree. Developer can use a DOM library to explore and manipulate with the document contents. Main advantages of this approach are that manipulation with the XML document is simplified but most of all the document is validated before the model is constructed. The DOM can be also built inside the program and than rendered into XML document. This ensures the XML document correctness and well formedness. Alternative to DOM is Simple API for XML or SAX. The SAX interface enables the access to XML elements in the process of parsing. It does not construct any DOM automatically so users have to process the XML in the parsing process. Because it does not construct the XML structure in memory, the SAX methods is more effective and essential when manipulating with sizeable XML documents with thousands of elements. Howerver for simple documents the DOM method is more straightforward to use.

2.1.1 XML Schema

XML Schema or XSD (XML Schema Definition) is widely used language for XML type defintion. Similar to XML, it is also standardized by the W3C. It defines the set of rules XML Document needs to follow to be considered valid. It extends the native Document Typer Definition DTD concept and introduces build in or custom data types and element inheritance. Also one of the advantages is, that XSD is formally described also in XML. The XML schema defines non-atomic types that express the structure of the XML document. As stated in the previous section, if the XSD is defined in the XML document header, the document is validated during the parsing process - parser knows the structure of the XML document. No further validation is needed and user can directly work with the DOM, because validation is enforced. In this project, the XSD schema notation is modeled using UML class diagrams. Application of XSD schema and real examples from this project are listed in the chapter 4.

2.2 Unified Modeling Language

Unified Modeling Language or UML is the standard modeling language for software and systems development standardized by the OMG. A model is an abstraction of the real system. The abstraction removes details that may be irrelevant or confusing so the model becomes simplification image of the real system [13]. This makes the design more understandable. Another criterium of modelling language is that it can not be too detailed like using source code or too verbose and ambiguous like natural language - it needs to be a formal modeling language. [23]

The Object Management Group (generally referred to as OMG) is a international, non profit computer industry standards consortium [31]. Standards created by OMG allow visual design, execution and maintenance of software and other processes. Relevant standards are UML, XMI and QVT are covered in following sections.

UML Class Diagram falls into the category of Logical View diagrams, it models the system parts including classes from static point of view [33]. Main modeling elements are the classes. Each class can have attributes or operations - methods in the UML terminology. Classes are connected with relations of several different types with diferent semantics.

2.2.1 XML Metadata Interchange

XML Metadata Interchange or XMI is a standard by OMG [30] for metadata information exchange using XML capable of carrying any data expressed as MOF. Since UML meta-model is considered to be a MOF meta-model, because it is defined as MOF standard, XMI can be used as a model interchange format for UML. The XMI standard is very complex so this chapter analyzes information about what sections of XMI are essential for this project.

The UML data expressed by XMI fall into two model categories: the abstract model and the concrete model. The abstract model is more important for this project - it holds the UML semantics and instances of UML expressed in MOF. On the other hand, the concrete model contains information about the UML diagrams and their visual [14]. There is also a dedicated part of the XMI standard called diagram interchange. This project checks for correctness of the UML diagrams. The correctness is evaluated only in the model itself, the visual representation is not important.

The XMI standard is very general and complex so as a result, there is several vendor implemented versions and XMI extension attributes that are not specified by the standard are heavily used. As a result some of the XMI are not compatible and tools often allow export in Development of UML modelling software is not an easy task, so most of the tools are created and sold by specific companies. The most well known and open tool is a part of the Eclipse Modelling platform - the Model Development Tools MDT 2.7.

XMI allows exchange of models on two forms: complete and as a transformation. The complete model holds the definition of the UML model - both ways in fragments or complete. On the other hand transformation form expresses just a differences of the model, changes that are to be done in a form of create, edit, delete records. The transformation model can be utilized in situations, where a tool loads a model, makes changes, extends the model - the changes and the source model can be exported separatelly. Although this type of XMI form can be used in cases like. This type of XMI notation is suitable for some software design use cases like using XSLT to transform UML models in a form of XMI [18] as an interchange between tools and as a temporary model representation in Model Driven Architectures MDA, but it is not suitable for checking. However every tool that supports exporting just XMI transformations should also be able to export the whole model.

The latest version of the XMI standard is 2.4.1 released in 2011, with expected up-coming version 2.4.2. However the most widely supported is XMI version 2.1. that is also compatible with the Medini QVT library. The reason for that is, that XMI 2 is largely different from XMI version 1.

2.3 Validity of UML

This section elaborates on the term validity of UML diagrams. Understanding the term correctly is important for next steps in the problem analysis.

The syntax of UML is defined by the OMG standard and it is scrictly set. The input representation for this project is the XMI format that is validated by the XSD. The syntax of the input UML will be always correct.

The introduction suggested that the main problem for validating UML is the semantics. The basic semantics is defined by the OMG standard. However just the recent UML version do have the basic semantic rules specified in the form of OCL because of the merge with MOF. However these rules are not enforced automatically, the enviroment has to support them. Some of the software modeling tools can still produce invalid UML diagrams and export them using XMI.

The other type of UML validity comes from the research on various topic of UML modeling. These sources often represent the invalid UML constructions in the form of patterns. Although some of the semantic errors are yet not a direct part of the UML standard, the UML is still not considered valid.

As with any other development there are some bad practices when using UML. These are not set by any rule or standard, but they follow a recommended way of design. The bad practices certainly do not have the severity to match the true semantic errors, but they still can be relevant to the user.

And finally, the last type of UML validity is the validity against custom set of semantic rules. Some companies or universities can enforce a set of rules, either the UML diagrams or the source code generated from them needs to obey. The other source of custom semantic or syntactic rules may also be a personal preference of the developer or designer.

This project tries to be as open as possible so it needs to take into consideration following sources of UML invalidity:

1. The UML Standard by OMG
2. Pattern databases from research
3. Bad practices, recommendations
4. Custom company or personal requirements

2.4 Incorrectness Patterns

The problems in UML model and class diagram are caused by undesirable interactions or constraints. The constraints in UML language are very diverse and that makes analysis of problematic interactions very dificult. However typical interactions can be identified and sorted into groups by structure. And this is where patterns come in. Each group can be expressed by a incorrectness pattern.

A Pattern can be also viewed as a set of properties. The first one would be the *name* of the pattern, that characterizes the pattern in a few words. Than there should certainly be a *description* of the problem on hand in plain informal text. The description should be accompanied by an example in a form of simple demonstrative UML diagram that serves also as identification structure for the user. Very important component is also a repair advice, notes on how to refactor the diagram so that the diagram is correct.

2.4.1 Pattern Sources

Last section described patterns as a base unit for checking the correctness and defined the pattern metadata and ways to express it. This section continues with the overview of sources, that may contain patterns both directly - patterns can be adapted and expressed to be used within the tool or indirectly - patterns need to be formalized. Some of the pattern sources are online incorrectness pattern catalogues that are results of years of research and experience in software modeling. They are developed mostly for educational purposes or as a reference of design problems, their explanations and repair advice.

One of the aims of this analysis is also to extract pattern categories. Summary of the pattern categories identified in the catalogues is listed in the table 2.4.1.

Patterns, Anti-Patterns and Inference Rules

First of presented pattern sources is an online catalog by BGU Modeling Group from Ben-Gurion University of the Negev that was created as a result of years of research and several articles [5]. It is mainly focused on correctness and quality of class diagram design.

This research defines the class-diagram as a backbone of UML and sees it as a collection of classes, associations, attributes and operations, and constraints that are imposed on them [21, 20]. Instance of the class-diagram is placed into the context of a domain and the symbols are mapped to the elements in the domain. Classes are mapped onto objects, properties are mapped to multi-valued functions over the sets and associations are mapped to relationships between sets of objects.

The work also defines a pattern class diagram [2] for easier pattern notation. This notation is also more suitable for pattern justufication and correctness proof using the catalogue system. However the main and most important source for the tool is text description of the problem and the pattern.

In the pattern catalogue, patterns are categorized into 2 main categories: correctness and quality. *Correctness* category afterwards splits into *incosistency* that represents patterns indicating class diagram can not have an existing instance and *finite satisfiability* patterns that match classes that do not satisfy the multiplicity requirements of the model. *Quality* category patterns indicate errors with lower priority that may be considered more like a warning. Patterns are split into *redundancy* problems describing imprecise specification such as syntactic error, missing information or multiplicity constraints are not specified clearly enough and *comprehension* patterns that lower the clarity of the design [4]. However patterns from the last group are not yet available and usable for the needs of the tool.

Metamodeling Anti-Patterns

Another one of the pattern catalogues is the Metamodeling Anti-Patterns catalogue [10]. One of the authors Maged Elaasar is also a member of OMG and some information and approaches from this catalogue were used by the OMG and UML Revision Task force or UML RTF to validate the UML standard itself, see the work of the UML RTF at [28]. Please note that the terminology anti-pattern or pattern depends on the context and author. Throughout this text there will be explictly used only the term pattern that will cover all errors in UML representable by a pattern.

The catalogue is targeted mainly on the UML language from version 2, but also on MOF [8]. Similarly to the previous catalogue, it represents errors that can occur during the modeling by patterns that are grouped into categories based on the problem they cover. The quality of the patterns are supported by the renowned authors and a fact, that they are used by a group directly in OMG and will have a big influence on the quality of future UML revisions. Each pattern is identified by a informal full text specification, a pattern example and a formal specification using the QVT language [12, 11].

This paragraph elaborates on the relation between UML and MOF. The UML metamodel is tightly bound with the MOF since recent revision 2.4. As [10] describes, the separate MOF will be considered deprecated and it is instead replaced with UML 2.4 metamodel with additional constraints on top. This method will be the future way to define the abstract syntax of modeling languages starting with UML 2.4 itself. This transition is caused by the lesser usage of MOF (mostly used just for import and export of models) and a wide spread of UML that is utilized for model definition. OCL sample [32].

This catalogue differs patterns into following categories [9]: UML, MOF, Sematic, Convention and Notational.

The *UML* category contains patterns that affect the integrity of the model based on the semantics of UML 2.4. It contains some patterns documented in the recent UML standards, but some of them are unique and most definitelly will be a great addition to the project pattern database.

The *MOF* pattern category is similar to UML, but covers pattern that are unique with MOF.

The next category, *Semantic* patterns contain patterns, that are syntactically according to specification, but they may cause confusion or problems when used in the model. These patterns have lower severity and should be assigned appropriately.

Category with even lower severity called *Convention* contains patterns that violate the naming or documentation conventions. This category should be also considered only as warning severity.

And finally the last category, the *Notational* patterns. These patterns cover errors in the diagram itself. Although occurence of these patterns may lower the clarity of the model, they do not have any direct influence on the UML model validity. Moreover the notation to visually represent the diagram in XMI is not unified. There is a Diagram Definition standard by OMG, see [26], but it is not unified throughout the software modeling community and as a result, most tools use their own notation to interchange the visuals of the diagram. Most tools use the capabilities XMI.extension and their own set of elements to denote the placement and formating of diagram elements.

Pattern Category	**Severity**
Correctness - Inconsistency	Error
Correctness - Finite Satisfiability	Error
Quality - Redundancy	Error
Quality - Comprehency	Not Supported
UML	Error
MOF	Error
Semantics	Warning
Convention	Info
Notational	Not Supported

Figure 2.1: Table: Overview of Pattern Categories

2.5 Object Constraint Language

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 2.2 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)
```

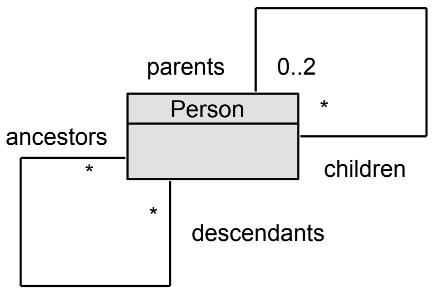


Figure 2.2: OCL Example

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 2.6.

2.6 Query/View/Transformation

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 [27]. 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 2.5 and extends it with imperative features. That makes QVT more suitable of checking UML correctness.

The connection languages from the QVT set have is depicted in the diagram 2.3.

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 articles [12, 10]. 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.

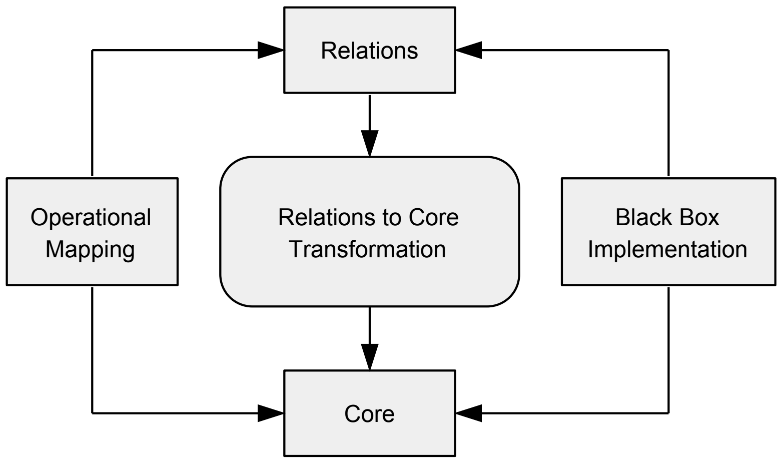


Figure 2.3: Diagram: QVT Language [27]

2.7 Eclipse Modeling Framework

The Eclipse Modeling Framework or EMF is a modeling framework and code generation platform for building tools and application based on a structured data model [34]. It is an optional part of the Eclipse IDE. The tool can descibe the domain model. This meta-model describes the structure of the result model, the result model is its instance. The EMF provides a framework to store the meta-model so that it can be created and defined with various technologies [1]. The advantage of this approach is that the domain model is explicitely defined and clearly visible. When the meta-model is defined, the result model can be automatically generated including interfaces and factories or even the complete Java code. The Eclipse EMF also has a runtime support for the models, persistence via XMI serialization and generic reflective API for manipulating the objects.

2.7.1 Ecore

Ecore is the meta-model for Eclipse EMF. The purpose of meta-model is to describe other models. The meta-model is built from basic elements: object classifications `EClass`, object attributes `EAttribute`, associations between the objects `EReference`, operations and simple constraints [22]. The relationship of Ecore to other models is shown in the figure 2.4. All models are equivalent: the UML Class is transferable to the Ecore EClass, XML Schema Complex type or the implementation of the Class in Java. Because of this, the meta-model can be implemented by any of these languages.

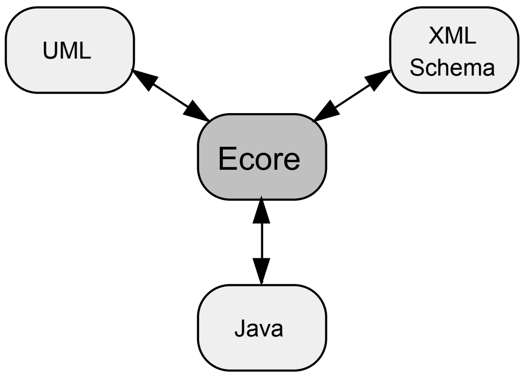


Figure 2.4: Diagram: Eclipse EMF Ecore [22]

2.7.2 Eclipse Model Development Tools

Eclipse Model Development Tools or MDT is a project that focuses on implementation of industry standard metamodels in the Eclise Modeling Framework. The project also provides tools for developing models based on these meta models. The most important part for this thesis is the UML2 implementation Eclipse Model Development Tools offer. The UML2 implements the Unified Modeling Language meta-model. Part of the UML2 project is also a support for common XMI schema interchange. Since the Medini QVT engine (see 2.8) this thesis uses the same EMF platform, the output XMI from Model Development Tool should be the most compatible.

2.8 Medini QVT

Medini QVT is an IDE for model to model transformations [16]. The main feature is the engine to parse and execute the QVT Relations. The IDE includes graphical QVT debugger that also allows developer to step through the relations. The QVT sources are edited in the QVT editor with syntax highlighting and code assistance. The result QVT transformations are be executed and debugged directly in the IDE. All of these features extremely help during the development of QVT transformations. The Medini QVT IDE offers one of the best support for QVT development so it is highly recommended for implementation of QVT patterns.

The QVT engine is build on top of the Eclipse framework and it uses the same library system. The development plan was to release the functionality directly as an eclipse plugin, but the work has not been finished. However it is still possible to use the libraries together with essential Eclipse framework libraries in custom developed applications [17]. The libraries can be integrated to the core of the tool library so that the QVT engine is available to perform the transformations from the UML model to the UML Result model. The Medini QVT engine components are free to use for non-commmercial use and they are relase under the EPL license. The latest version released is Medini QVT 1.7.0 Release Candidate that dates to 2011. It is targeted on the Eclipse 3.6 Helios and uses EMF version 2.6.1.

Bundled with the IDE there is also an example project with a set of QVT relations language that helps during the devlopment as well. Other source of the documentation are also tutorials on the Medini QVT homepage [15]. The tool works generally with any models represented in Ecore.

2.9 Visual Paradigm

Visual Paradigm is a well known, cross-platform and easy to use *visual UML modeling and CASE tool*. Its interoperability with other tools and IDE also contributes to streamline the model driven development process. The tool is capable of using not only all kinds of UML 2 diagrams, but also other notations for the database and business process modeling [38]. This section is focused on the description of Visual Paradigm and UML class diagrams. It also points out a way to extend Visual Paradigm functionality so that it can become a platform for this project.

The default user interface of Visual Paradigm is shown in the figure C.3. On the top side, there is *main menu*. The *toolbar* with icons is placed under the menu. On the left side there are various *panels* for browsing models, diagrams and changing attributes. And finally the most place on the screen takes up the *diagram designer window*.

2.9.1 Visual Paradigm Model Quality Checker

Model Quality Checker is a diagnostic tool that identifies potential design problems in the Visual Paradigm project [36]. User is presented with the errors in the diagram elements. The error severity is based on the point system. Each problem is assigned a point value and when the sum for an element exceeds the threshold it is considered as a problem element. A problem can be fixed based on a suggestion or ignored. The problems are listed in the figure 2.9.1.

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 2.5: Listing: Visual Paradigm Model Quality Checker problems [36]

Outside the Model Quality Checker, Visual Paradigm also checks for constraints that are set by the UML standard. However the plugin implemented by this project can introduce other types of rules and constraints including a lower severity recommendations. There is also a possibility to define entirely custom rules or share them via the database.

Addition of the plugin would greatly enhance the functionality of Model Quality Checker in Visual Paradigm. Next section describes the basic rules of developing a plugin for the Visual Paradigm.

2.9.2 Plug-in Development

Visual Paradigm has an open architecture that results in the possibility of extending the framework through its OpenAPI [35] via plugins implemented in Java. The API allows modification of Visual Paradigm model data, invocation of build-in functions and also implementing a set of custom functions to add features or services to Visual Paradigm. Also the plugin can extend the Visual Paradigm user interface by adding custom menus and tools for triggering the newly added functionality. Since the new functions are implemented in Java, they can implement their own user interface using the standard Java graphical user interface libraries and packages.

Extending Visual Paradigm with plugins is not complicated. The plugins based on the OpenAPI can be developed using the library `openapi.jar` provided by Visual Paradigm as a part of a standard instalation in the lib subdirectory. This library defines the interface between the plugin and Visual Paradigm. The documentation is available in a form of Javadoc that is downloadable from the Visual Paradigm website [2].

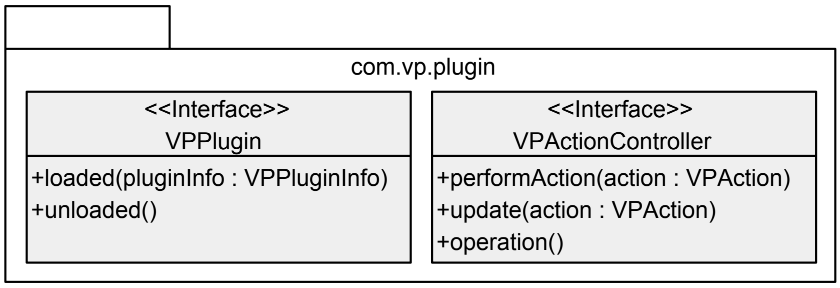

com.vp.plugin
<<Interface>>
VPPlugin
+loaded(pluginInfo : VPPluginInfo)
+unloaded()
<<Interface>>
VPActionController
+performAction(action : VPAction)
+update(action : VPAction)
+operation()

Figure 2.6: Diagram: Visual Paradigm OpenAPI

As explained in the diagram 2.6, the OpenAPI defines two basic interfaces, `VPPlugin` and `VPActionController`. Each plugin must have a class implementing the interface `VPPlugin` that has a possibility to attach the functionality on load and unload of the plugin and at least one class implementing `VPActionController` that handles the actions invoked from the menu, popup menu or toolbar. These classes along with the other classes implementing the plugin logic are stored within a package. The plugin registers itself within Visual Paradigm using a XML file with metadata named `plugin.xml`. The structure of the Plugin XML file is displayed in the diagram 2.7. The top element is `plugin` that holds basic metadata, id and name of the plugin, class attribute is a name of the class that implements interface `VPPlugin`. Plugin is a set of `actionSets`. Each of the `actionSets` can contain its own menu, toolbar and a set of actions. Each of the actions require at least one `actionController` that will be executed [37].

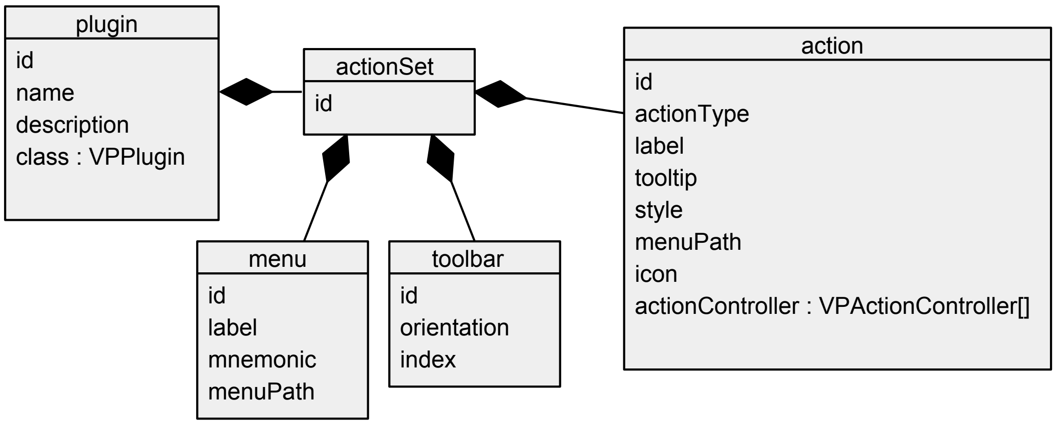


Figure 2.7: Diagram: Visual Paradigm plugin.xml

The plugin is installed just by copying the files in predefined folder structure to the Visual Paradigm instalation directory. The implemented plugin needs to have the correct file and folder structure so that it can be successfully deployed. The file and folder structure

[2] `<http://www.visual-paradigm.com/support/documents/pluginapi.jsp>`

is displayed in the figure 2.8. The root directory is named after the plugin id defined in the `plugin.xml` file. It contains a directory `classes` with all compiled java classes and also the `plugin.xml` file that tells Visual Paradigm how to integrate the plugin into the interface.

```
<plugin>
    classes
        <plugin package>
            <action packages>
            <implemented VPPlugin.class>
            <implemented VPActionController.class>
    plugin.xml
```

Figure 2.8: Listing: The Visual Paradigm Plugin Directory Structure

There are several available licences for Visual Paradigm [3]. The Visual Paradigm Standard license is available for BUT FIT students. There is also a Community version free for non-commercial use [4]. All of the available versions have a plugin capability.

2.10 Technologies for Implementation

This section introduces the technologies, that will be used in the implementation phase. This project is a set if tools including command line application, plugin for Visual Paradigm and a database managed over the web interface. The database stores all the patterns that are used for validating the UML diagram so this component will be described first. Both command line application and plugin for Visual Paradigm work with the database of the patterns and will share some of the functionality.

2.10.1 Database

Patterns should be stored on the server part inside the relational database. This method of storage is commonly used in combination with web interfaces. It also provides a possibility to uniformly store the data so that they can be easily manipulated with. Relational SQL based database also provides enough space for possible extension in the future.

For the implementation of the database the PostgreSQL server was chosen, because it is open source, however it provides business class features [25]. As an open source software it can be easily installed on any system so that the pattern web management system can be also deployed privately [6]. The schema of the database that stores the patterns however does use only standard SQL features and dialect and can be easily ported to any other database system.

2.10.2 The Server

The server part is directly connected to the database and hosts web interface for users to manage the pattern database. The whole set of tools will share a common library that will contain a model (see chapter 3) so it is convenient to implement the server part in Java. The technologies chosen for the implementation of the server part are JSP in combination

[3] `<http://www.visual-paradigm.com/editions/>`

[4] `<http://www.visual-paradigm.com/editions/community.jsp>`

with Java Servlets and Freemarker templates. Java Servlet is simply put a traditional Java class put in the enviroment of the server. Its main task is to handle the request and return the appropriate response. The output web page is rendered using the MVC architectural pattern and Freemarker templating engine. Model View Controller or MVC is a software architectural pattern for implementing user interfaces. As the title suggetsts, its main components are the model that is the interface for accesing the data, controller manipulates with the model and sets the view and finaly View requests information from the model and outputs them for the user. The MVC architectural pattern is displayed in the simple diagram 2.9.

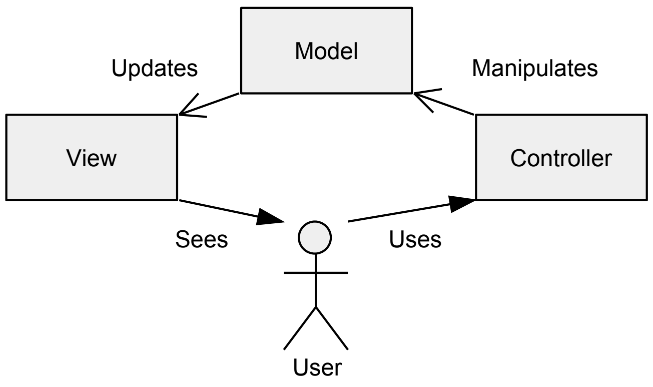


Figure 2.9: Diagram: Model/View/Controller Architectural Pattern

2.11 Testing

Section briefly explains basic terms and testing approaches that apply to this project. The term testing can be defined according to [24] as the process of executing a program with the intent of finding errors. The testing process tries to add a value by raising the quality or reliability of the final product.

The most well know *testing strategies* are Black-Box and White-Box testing. The *Black-Box testing* strategy percieves the program as a black box, the test does not have any knowledge or assumptions about the inner structure of the program. The test data are created directly from the initial specifications and requirements. The workflow usually is to prepare the input test data, execute the program and compare results with expected output.

The second complementary testing strategy is the *White-Box testing*. This approach tests the inner structure and the logic of the application as opposed to Black-Box testing. The test data are derived from the logic of the program or directly from the source code so that possibly every statement in the source code is tested.

None of these testing strategies are perfect. As a result they are mostly executed together in synergy so that maximum coverage and quality enhancement is achieved.

2.11.1 Unit Testing

Unit testing is the process of testing the individual subprograms, subroutines, classes or procedures in a program. The unit testing work with decomposed program and it heavily relies on knowing the *inner structure* and thus it falls into the White-Box testing category. However tests on the inner logic are often complemented by a set of Black-Box tests so that the unit specification is covered. This project makes use of the unit testing to test the inner logic of several key components. The components with a big influence on result product were chosen so that the benefit of the testing is the largest. Since the unit testing is executed on the lower level of the decomposition and is closely tied with the source code, it can be easily *automated.*

2.11.2 Functional Testing

Functional testing tries to find discrepancies between the program and external specification [24]. The first task of functional testing is to identify the *test cases* by analysis of the specification. The function testing usually is a black-box activity so the important part of the test is to prepare the test input and expected result. The test follows what the system does with the input and than compares the real output with the expected one. Functional test are executed according to the test plan. In some cases, it is possible to automate them but *manual test execution* is not an exception.

2.12 Goals

Goal of this thesis is to implement a tool to *check correctness* of an UML Diagram, specifically Class Diagram. The standalone tool for use from the *command line* will work on top of the *interchangeable representation of UML diagrams* and produce *text based* correctness analysis results. Another interface of the tool will be also implemented as the *Visual Paradigm modeling software plugin* that will analyze the currently opened diagram and present the results in the *graphical user interface.* Both of the tools use the same platform wrapped in the *library* and a *shared database of incorrectness patterns* manageable via a web interface. The openess and extensibility of the tool should be emphasized in the implementation process. The result tool will be tested, evaluated and finally *published* online.

Chapter 3

Design

The design phase deals with modeling the system so that there is a clear overview and of what needs to be done in the next step - the implementation.

Design phase usually starts with requirements for the system. The source of requirements for this project is the thesis assignment and goals identified in the section 2.12. These requirements are represented in informal plain text, so the most suitable next step is to represent them in a more formal way so that the requirements can be used as a foundation for the rest of the design process. The requirements will be modeled using UML use case diagrams. After defining the use cases, user roles and tasks become more clearer so that the system can be decomposed into functionally separated components in the section 3.2. At this point, more UML modeling techniques can be applied and the decomposed system can be modeled, at first the input, followed by the pattern data and the logic of the tool and finaly an output.

3.1 Use Cases

The use cases are a way of expressing the functional requirements of a system and are a good starting point fo problem decomposition and further software design [13]. They are created on top of the initial requirements that are set by the assignment and goals identified in the section 2.12.

The use case diagram implementation begins with the role identification. The requirements describe two interfaces for the application targeted on different types of user. The different user interfaces are express by two roles: *Command Line Tool User* and *Visual Paradigm Plugin User*. Since the most of the functionality in both interfaces is similar the general *Tool User* role is also identified. The use cases common for both interfaces are assigned to the Tool User role. The roles Command Line Tool User and Visual Paradigm Plugin User both share the use cases of the Tool User role and add a few use cases specific for the interface and type of use. The complete use case diagram is shown in the figure 3.1.

The pattern database web interface as stated in the requirements is shared and allows management of the patterns. The sharing of the database is modeled by the *Update the pattern database* use case. The management of the pattern database is aimed at more advanced users or developers. This requirement is represented by the role *Tool Power User* that has the use cases of *Managing the pattern database* and also to *Export the patterns.* The use case diagram for Tool Power User role is shown in the figure 3.2.

The use case diagram is a more structured representation of the requirements. However

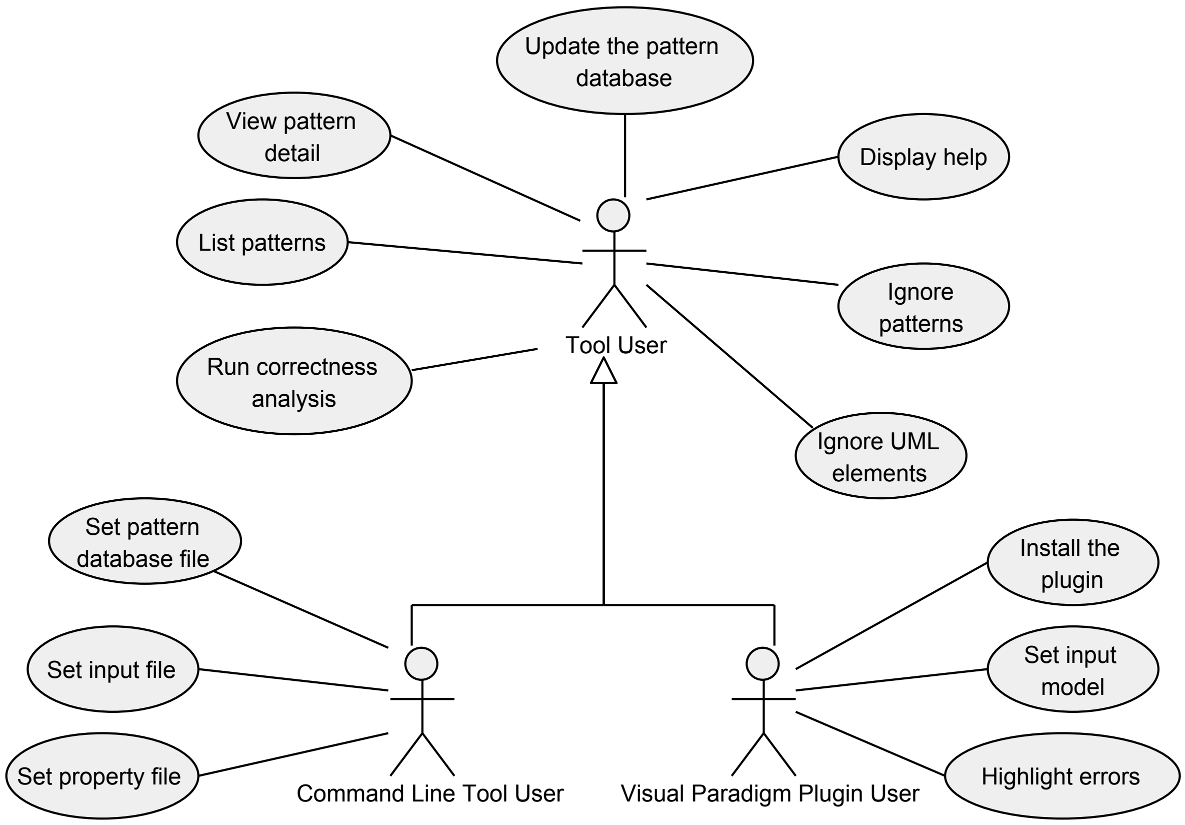


Figure 3.1: Diagram: Use Case diagram - Tool User Roles

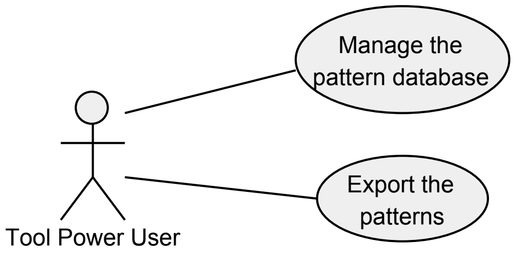


Figure 3.2: Use Case diagram - Tool Power User Role

the use cases need to be described in more detail.

View pattern detail: display all information about a specific incorrectness pattern.

List patterns: list all patterns in the database.

Run correctness analysis: run the correctness analysis and display results.

Update pattern database: download updated database from the server.

Display help: display a short tutorial (plugin) or display arguments (cli).

Ignore patterns: disable specific pattern from the correctness analysis.

Ignore UML elements: disable checking of specific UML element.

Set pattern database file: set file containing the pattern database.

Set input file: set input file with the diagram.

Set property file: set property file with configuration.

Install the plugin: install the plugin using the native method.

Set input model: choose a model to be analyzed.

Highlight errors: highlight an UML element from the correctness analysis results.

Manage the pattern database: create, edit or delete patterns from the pattern database.

Export the patterns: export the whole pattern database.

Use Case: Export the Patterns
ID: 01
Actors: Tool Power User (User)
Preconditions: 1. The pattern database web interface is opened.
Event Flow: 1. User clicks on the *export* menu. 2. System returns a *file for the download.*
Post conditions:

Figure 3.3: Use Case Detail: Export the Patterns

Use Case: Manage the Pattern Database (Create)
ID: 02
Actors: Tool Power User (User)
Preconditions: 1. The pattern database web interface is opened.
Event Flow: 1. User clicks on the *patterns* menu. 2. System displays a *list of all patterns.* 3. User chooses to *create a new pattern.* 4. System opens the *create a new pattern* form. 5. User inputs the pattern metadata and submits the form by clicking submit. 6. System opens the *create a new pattern* form.
Post conditions: 1. Pattern is persistently saved. 2. Pattern is available for update and export.

Figure 3.4: Use Case Detail: Manage the Pattern Database

Use Case: Manage the Pattern Database (Delete)
ID: 03
Actors: Tool Power User (User)
Preconditions: 1. The pattern database web interface is opened.
Event Flow: 1. User clicks on the *patterns* menu. 2. System displays a *list of all patterns.* 3. User clicks the delete link on the pattern line. 6. System opens the *list of all patterns.*
Post conditions: 1. Pattern is deleted and no longer available for update and export.

Figure 3.5: Use Case Detail: Manage the Pattern Database

Use Case: Highlight Errors
ID: 04
Actors: Visual Paradigm Plugin User (User)
Preconditions: 1. The Visual Paradigm with installed plugin is opened. 2. The UML class diagram is currently open. 3. The correctness analysis has completed. 4. The correctness analysis results dialog is opened.
Event Flow: 1. User clicks on the UML Element in the Binding column 2. Plugin highlights the UML Element in the diagram.
Post conditions:

Figure 3.6: Use Case Detail: Highlight Errors

Use Case: Run Correctness Analysis
ID: 05
Actors: Visual Paradigm Plugin User (User)
Preconditions: 1. The Visual Paradigm with installed plugin is opened. 2. The UML class diagram is currently open.
Event Flow: 1. User clicks on *Analyze* in the plugin main menu. 2. Plugin runs the correctness analysis and opens a new dialog with the *analysis results.*
Post conditions:

Figure 3.7: Use Case Detail: Run Correctness Analysis

Use Case: View Pattern Detail
ID: 06
Actors: Visual Paradigm Plugin User (User)
Preconditions: 1. The Visual Paradigm with installed plugin is opened. 2. The tool web interface is accessible via network.
Event Flow: 1. User clicks on *Patterns* in the plugin main menu. 2. Plugin opens a new dialog with the *list of patterns.* 3. User clicks on the *Details* link in the pattern list row. 4. Plugin opens a new dialog with the pattern detail.
Post conditions:

Figure 3.8: Use Case Detail: View Pattern Detail

Use Case: Update the Pattern Database
ID: 07
Actors: Visual Paradigm Plugin User (User)
Preconditions: 1. The Visual Paradigm with installed plugin is opened. 2. The tool web interface is accessible via network.
Event Flow: 1. User clicks on *Update* in the plugin main menu. 2. Plugin updates the local pattern database.
Post conditions: 1. Local pattern database is updated. 2. Next correctness analysis and pattern list will load the updated database.

Figure 3.9: Use Case Detail: Update the Pattern Database

3.2 Decomposition

When the specifications are clear enough, decomposition process can start. The tool decomposition is shown in the diagram 3.10.

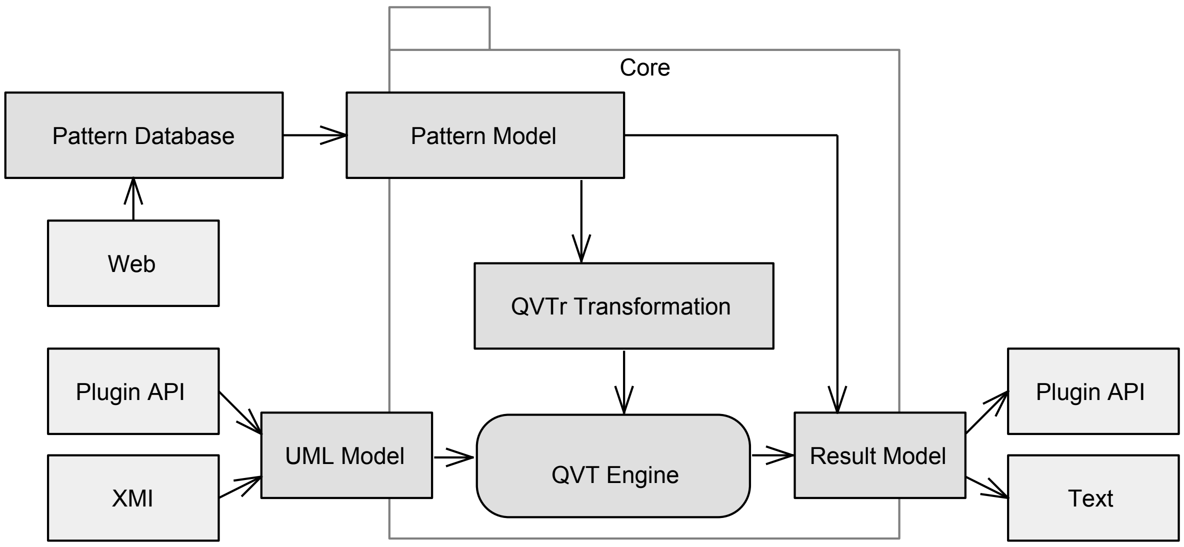


Figure 3.10: Diagram: Tool Decomposition

The *input* of the tool is the UML class diagram. In case of the command line interface, the diagram is stored in XMI. The Visual Paradigm plugin reads the diagram via the plugin API from currently open class diagram. The *output* of the tool are the correctness analysis results. The analysis result consist of the identified pattern, severity and an UML element the pattern was matched on. The command line interface outputs plain text, the Visual Paradigm plugin shows a dialog with the analysis result. On the server, patterns are stored inside the *pattern database.* On the client, the database is stored in the local file. Also it is updateable from the server. The database is manageable via the *web* interface. The client tool functionality is placed inside the *library.* Input of the engine is the UML Model and QVT Script. The *UML Model* is created by client directly from the Plugin API or XMI. The *QVT Script* is generated from the pattern database. The UML Model is transformed by the *QVT Transformation* to the *result model.* The result model is taken by the client application and presented to the user via *plugin API* or *text.*

3.3 Command Line Tool

The command line tool is a standalone application for use from the command line capable of checking correctness of UML diagrams. This section describes the functionality that is unique to this interface and they are not covered by the common library.

3.3.1 Input and Output

The command line tool performs the analysis on the UML model so it will be the main input of the tool.The other input the application needs are the application settings. The settings are be specified in the config file or can be overriden directly by some of the command line application arguments. The model is represented by input *XMI* file described in the section

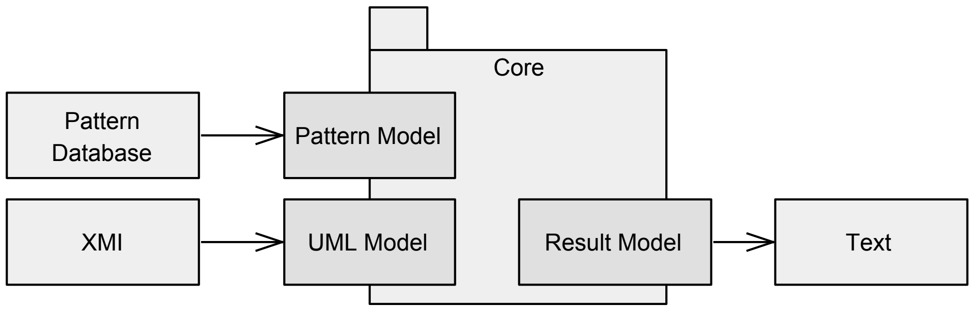


Figure 3.11: Diagram: Command Line Tool Overview

2.2.1. The XMI file is standardized by the XML schema, so it can be validated. This file is parsed into the *UML Model* and passed to the core of the tool that will use it for the analysis.

The main data that the tool works on top of are the patterns. The patterns are loaded from the *pattern database* into the local XML file. The file can be updated anytime. Every time the application starts, the *pattern model* is loaded from the file and passed to the core of the application as well.

The ouput of the command line tool is the results of the correctness analysis. The results come from the core of the application in a form of the *result model*. The model is formated into a table using whitespaces and printed to the standard output as *text*. The ouput contains the pattern id, pattern name, severity, and the identification of the elements. It is simple enought so that it can be processed automatically by a wrapper application or just redirected to the log file and read later. If a more detailed report is needed, the user can individually use the pattern identifiers in combination with the view pattern detail use case to view pattern details including a link to the web that contains more information.

3.3.2 Tool Modes

The the Activity diagrams model the workflow and processes in the software. The activity diagram is in the figure 3.13. The diagram describes the different modes and flow of actions for the command line standalone tool. The typical use of command line application is to set the arguments, application runs, parses the arguments, does the task defined by the arguments and exits. If the user wants to do something else, the application needs to be run again. This paradigm forces the application to behave in several different *modes* according to the argument set.

After being executed, the command line application loads the pattern database every time. After that it analyses the arguments and sets its mode acordingly. The mode defines other activity that the application needs to do and it does reflect the modeled use cases. Another view on the application mode is the command line arguments. The mapping of use cases to command line arguments is shown in the table.

Argument	Use Case
-a,–analyze	Run correctness analysis
-c,–config FILE	Set property file
-d,–database FILE	Set pattern database file
-e,–enable ID BOOL	Ignore patterns
-h,–help	Display help
-i,–input FILE	Set input file
-l,–list	List patterns
-p,–pattern ID	View pattern detail
-u,–update	Update the pattern database

Figure 3.12: Mapping the Use Cases to Command Line Application Arguments

3.4 Visual Paradigm Plugin

The tool with standard desktop user interface is be implemented in a form of a plugin to the Visual Paradigm modeling software. The plugin is directly connected to the user interface. The plugin version of the tool is more targeted on users and it is more intutive and easier to use. The components of the plugin are shown in the figure 3.14.

3.4.1 Input and Output

The input and output of the Visual Paradigm plugin utilizes the OpenAPI that allows plugins to communicate directly with the interface of the Visual Paradigm software.

The *input* is the UML Model. The core of the tool provides the *data structures* for representing and managing the UML Model so the plugin just needs to load the model into the structures and pass down to the core of the application where it will be processed. Thanks to the OpenAPI, the model is loaded directly from the Visual Paradigm and its *active diagram*. The diagram elements are matched on the model and saved into the data structures. The other input data needed is the *patterns* from the pattern database. The patterns are stored in the local XML file and loaded into the *pattern model* data structures when needed.

The *output* of the Visual Paradigm plugin is presented to the user via the standard *desktop user interface*. The Visual Paradigm plugin API allows plugin to use the tabbed text output. When the text output is not sufficient, plugin can invoke a *dialog* with custom content. The plugin uses two basic types of dialog window with similar visual design but different functionality. One of the use cases the plugin supports is to *list the pattern database*. The patterns are displayed in an organized table that is clear and effective for the user to use.The table contains typical pattern metadata defined by other sections. The last column of the table contains a clickable element. This will open a new dialog window with the pattern details loaded directly from the server part.

3.4.2 User Interface

The user interface of the plugin is merged with the user interface of the Visual Paradigm client application. The whole functionality of the plugin is controlled from the main menu of the application. When the plugin is installed a new menu section *UMLLint* appears directly in the main menu. The menu contains these actions: Analyze, Database, Homepage and Setup. The action *Analyze* runs the correctness analysis and displays a window with analysis

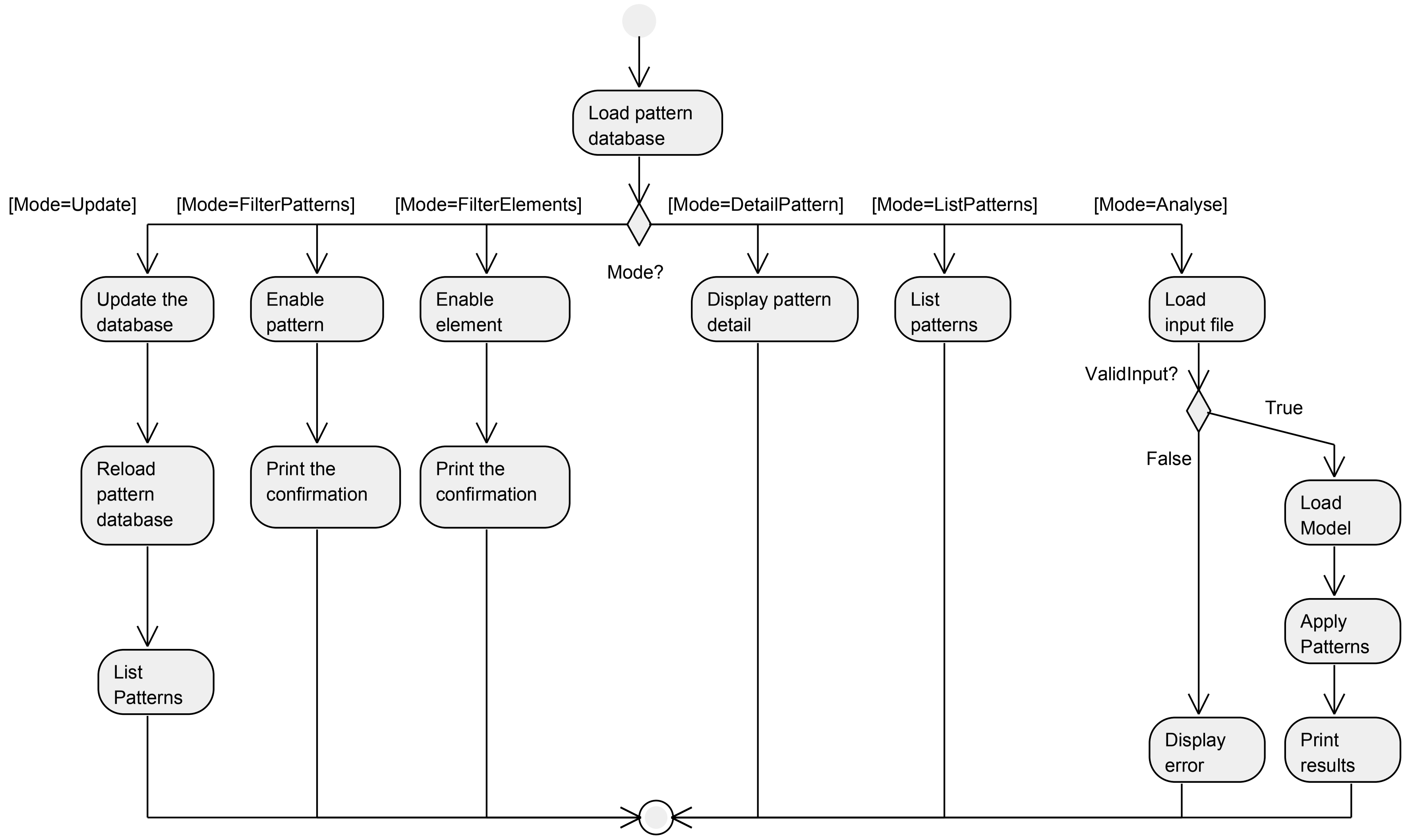


Figure 3.13: Activity Diagram

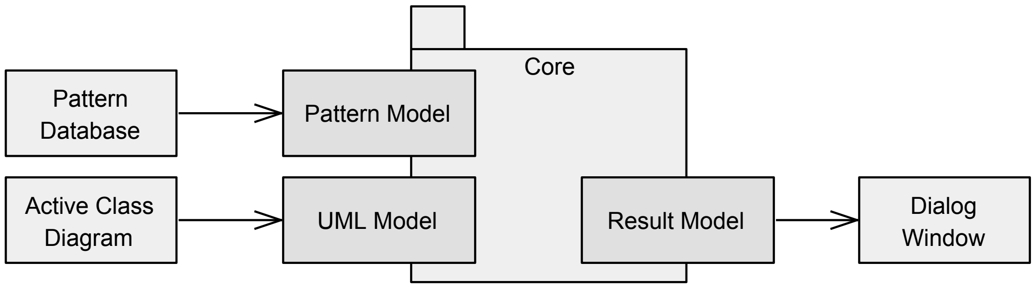


Figure 3.14: Diagram: Visual Paradigm Plugin Overview

results. The *Database* allows user to browse the database. The patters are displayed in a table so that user has an overview about what the plugin uses. The pattern details can be opened by a link in the database pattern list table. The *Homepage* menu action opens the project homepage. The homepage contains more information about the project, but more importantly a user can access the web interface of the pattern database. The user interface of the plugin is shown on the screenshots in appendices.

3.5 Core of the Tool

The core of the tool wraps up the most of logic: the libraries providing the QVT engine, QVTr transformations, the models and a model transformation interface. The component overview of the core is shown in the figure 3.15.

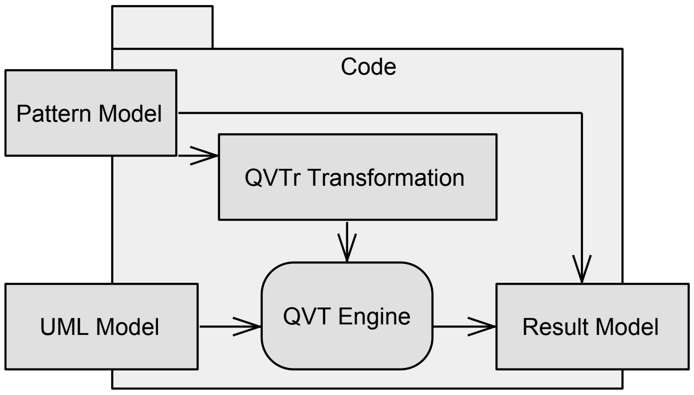


Figure 3.15: Diagram: Logic of the Tool

3.5.1 Models

The interface of the core component with the command line and Visual Paradigm plugin user interface are the models. The user interfaces transfer the input data into the core component and the component also forwards the output data in the form of models.

This project utilizes three models. The *pattern model* does represent the patterns. The pattern defines an incorrectness in the UML. It has a simple structure that characterizes the basic metadata, its severity and category. There is also a place for references. Since most of the patterns comes from a research, the source should be properly atributed. If the user is interested, he can find the origin and read more information about the pattern. The pattern model is defined in the figure 3.16.

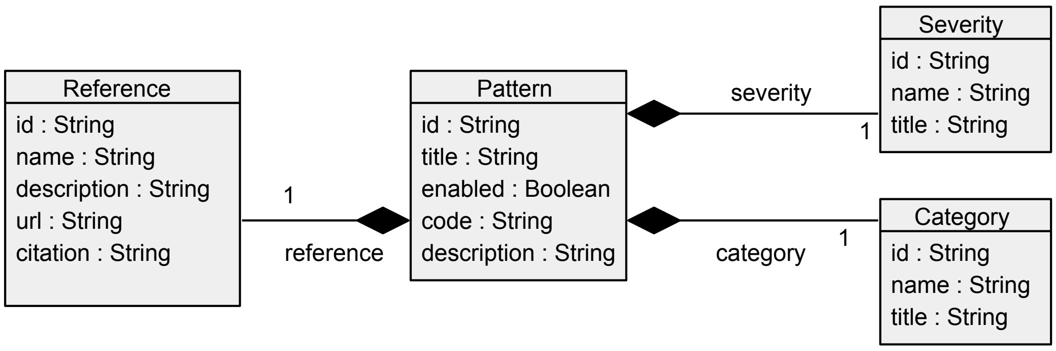


Figure 3.16: Diagram: Pattern Model

The second is the *result model* also known as *umllint model*. This model is a target model of the QVT transformation and represents pattern matching occurence on the UML model. The pattern model is shown in the figure 3.17. It holds the basic metadata about the matched pattern. The Binding represents a relation to the the UML element, where id is the XMI id of the element, name is available for the developer of the QVT relation to store the element identification and type is a type of the element. The Binding is also specified recursivelly. When binding on an association, we may want to bind on the class as well.

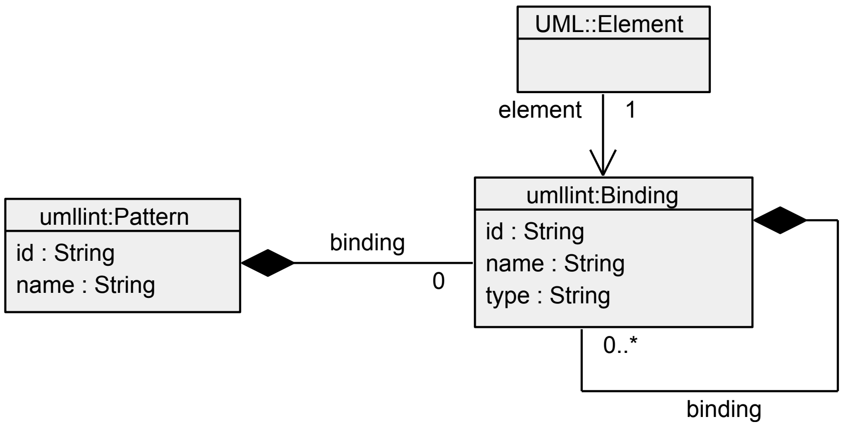


Figure 3.17: Diagram: Result UMLLint Model

The *UML model* is already well known and described by sources [33] or [19]. It is defined by the OMG standard [29].

3.5.2 QVT Transformation

As the analysis suggested, the correctness check will use the model transformation with the addition of patterns. The input UML model the tool will be transformed to the UMLLint result model with the help of the QVT transformation. The QVT transformation is composed from the set of the relations. The relations are provided by the patterns. The QVT transformation applies the pattern matching process and the result UMLLint model contains the pattern occurences. If the result model contains a pattern than the pattern was found in the source model and source model contains an error.

The QVT transformation itself is a complex process and does require several inputs. The overview of the QVT transformation requirements is listed in the figure 3.19. The whole correctness check using the pattern matching process and the QVT transformation can be described as a series of steps:

1. Build a QVT transformation from the QVT relations specified by patterns.
2. Supply the source UML and target UMLLint meta models to the QVT Engine.
3. Supply the source UML model.
4. Supply the QVT transformation.
5. Supply the source model.
6. Set the target model.
7. Set up QVT transformation name and direction.
8. Execute the transformation.
9. Analyze the target UMLLint model.

The transformation provides a pattern matching functionality. If the pattern matches sucessfully, the transformation matches the elements from the UML model to the UMLLint result model structure thus creating and occurence record. The process is displayed in the figure 3.19.

3.5.3 QVT Engine

The QVT Transformation needs a QVT Engine to run. The QVT Engine is a complex component that executes the entire QVT transformation and converts the input model to the output model. This project uses the Medini QVT engine as described in the analysis.

Requirement	
Source model	UML (uml)
Source meta-model	UML 2.1.0 (`uml.ecore`)
Target model	Results (UMLLint)
Target meta-model	UMLLint 1.0 (`umllint.ecore`)
QVT script	`umllint.qvt`
Transformation	`umllint`
Direction	`target`

Figure 3.18: Table: QVT Transformation Requirements

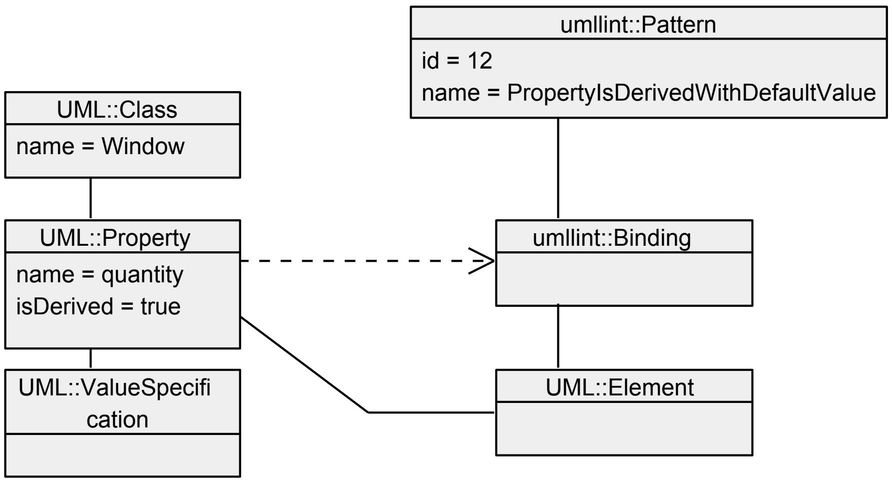


Figure 3.19: Diagram: Result Model Mapping

Chapter 4

Implementation

The implementation phase deals with the development of the command line tool, plugin for Visual Paradigm and the server part with the pattern database accessible via the web interface. The process of implementation follows the same decomposition process as designed and described in the section 3.2. The description of the implementation starts with the pattern database and server, moves towards the core and finally wraps up with the user interfaces.

4.1 Pattern Database

Persistent storage and management user interface is represented by the server part of the application described in appropriate section of Implementation chapter. Patterns are stored in the database implemented according to schema in Entity Relationship diagram. For database PostgreSQL was chosen because of business class features, scalability and availability. In combination with other technologies it allows users to deploy their own server instance thus customizing pattern database.

Patterns can be managed through the server web interface. The web interface offers all standard operations namely create, edit, delete and view. This approach presents an easy way for user to manage the pattern database. Description field can contain arbitrary HTML markup including links and images. Patterns shipped with the application utilize this to show example pattern diagram. Screenshot shows the pattern profile viewed in the web interface.

There are several pattern categories, like incorrectness and recommendations. Similar to category, each pattern is assigned a severity. This represents the impact of finding a pattern. Debug severity is reserved for users defining their own patterns. It can contain notes or status reports. Other degrees are utilized by the tool as well as the users. Info severity degree notes recommendations for design correctnes that have no effect on overall UML diagram validity and correctness. Warning level is basically an error that is however not severe enough to render the UML diagram incorrect. Highest degree is Error. Finding a pattern with Error severity will render the diagram incorrect and invalid. The tool database and library can be easily modified to add more severity degrees, but it must still be clear which severity is applied in which case.

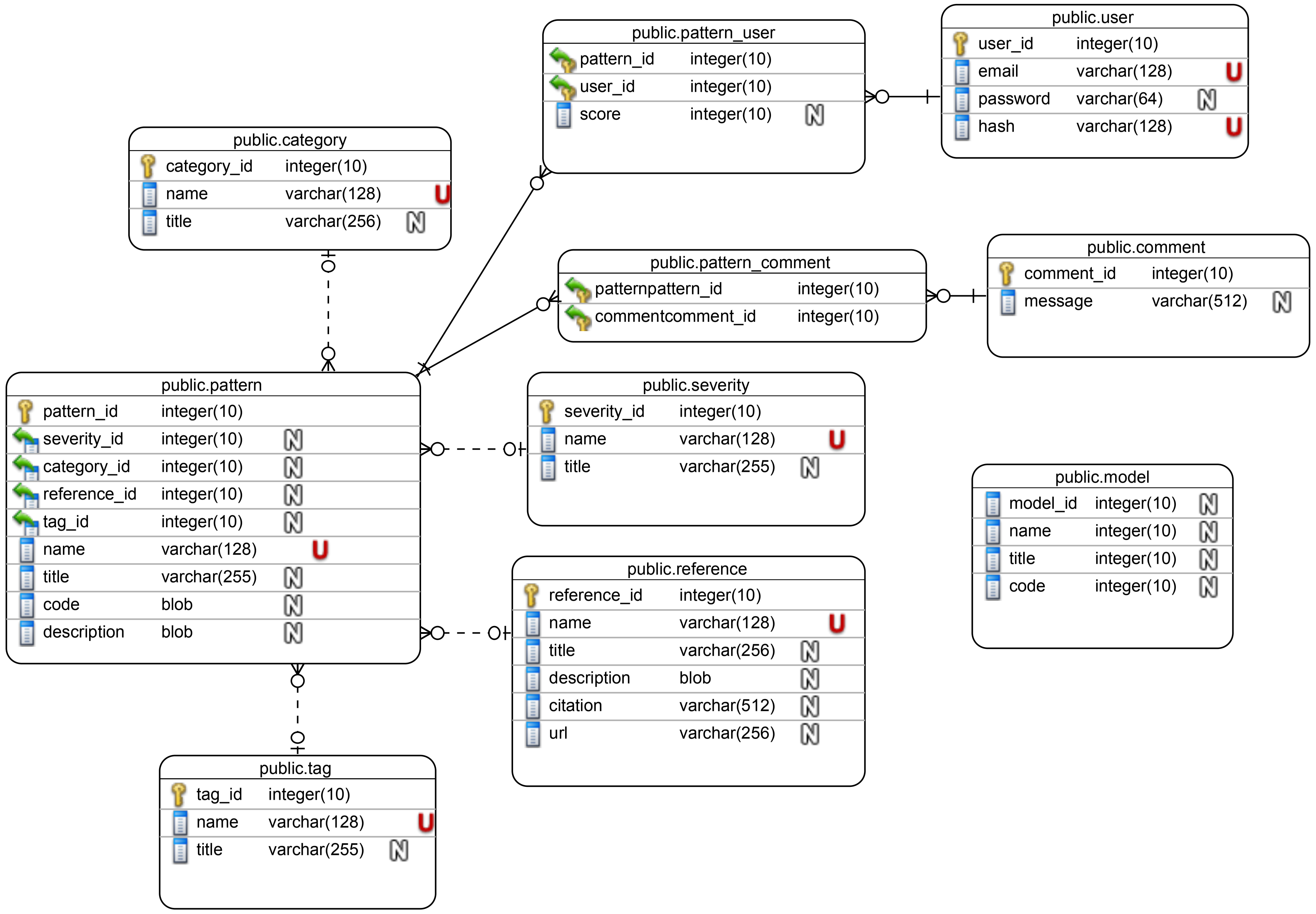


Figure 4.1: Entity Relationship diagram: Database schema

Severity	Consequence
Debug	No Consequence. Distinguish debugging notes.
Info	Important information for the user. Recommendations.
Warning	Error that is not severe enough to render diagram incorrect.
Error	Error causing diagram incorrectness.

Figure 4.2: Severities

Crucial role in the success of this project is the pattern database. Patterns are added to the database from several sources ranging from published articles, UML standard and websites created by renowned authors. Original source most of the time contains additional information often including more examples, theoretical proof and more tips on how to deal with the error. Some of this information is compiled into the *pattern database* however mostly in simplified form due to usability. Part of the pattern metadata is also a reference to the original source of information. Reference contains a title, description, citation and an url. Title is displayed in the application, description holds more information about the source like what is the source focused on. The citation string (like in the publication section in this thesis) is stored in the citation field and displayed every time the patterns is used because the pattern description can contain copyrighted information.

4.1.1 Sharing and Collaboration

There are two possible setups of the server and database part of the application. For the purposes of this project, the application is hosted and publicly available. However users can easily deploy their own server instance on local network or even local computer. Tool can be also easily setup in offline mode to just use pattern XML file and do not connect to server part at all.

From the web nature of the server part of the tool it is apparent that pattern database can be managed by more users that can collaborate on building sizeable pattern database and knowledge base. If the pattern database would be used publicly and extensively, further extensions enabling user management, authorization and user involvment and comments should be considered. The database is already designed to handle the data.

4.1.2 Pattern Set Interchangeability

Both command line tool and Visual Paradigm plugin update the database from the server via XML interchangable document. If using offline mode, this document can be also easily implemented or managed manually directly by the user. It can also be distributed separatelly or the static XML file can be uploaded on the network or web server. Another posibility is to use other system to generate the file.

The content of XML document is specified by XSD schema. The schema is also applied when loading the XML to validate the document. The structure is pretty straightforward - root element patterns contains a set of pattern elements. Each pattern element than contains metadata id, name, title, description (also present at pattern model) and also reference, category and severity. The schema does not directly follow database schema, but it is much more simplified so that it is better acccessible for users that may want to extend the application.

Both the command line tool and Visual Paradigm plugin use the pattern XML file. The file is generated on the server. Data are loaded from the SQL database into the shared

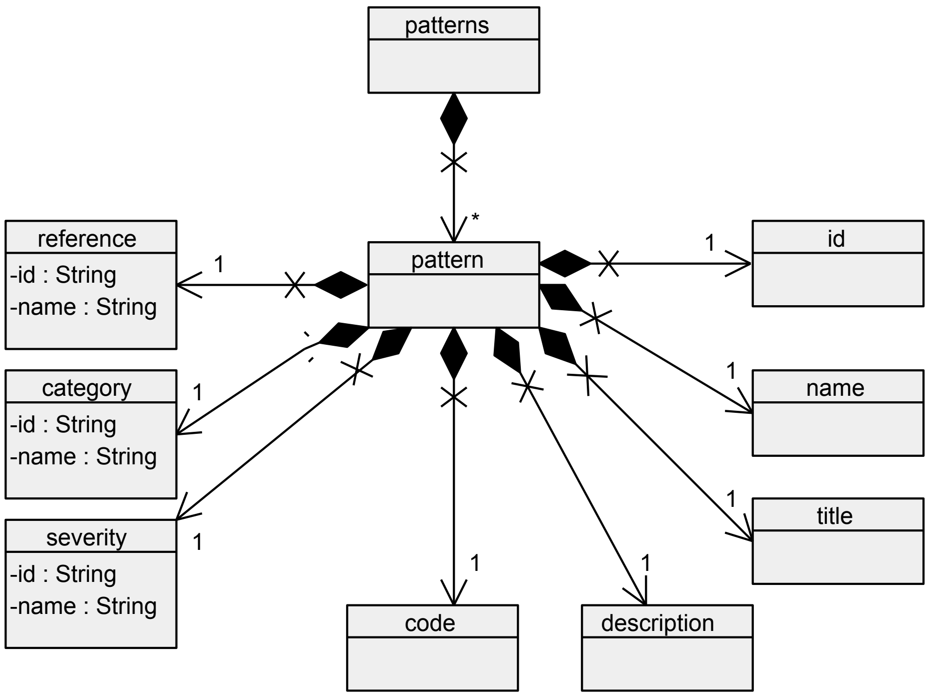


Figure 4.3: Patterns XSD Schema

pattern data model. From the shared pattern data model result XML is build using W3C packages. Result XML is returned to the user. Example XML can be seen on the attached CD.

Pattern XML document is transfered and saved on the client via command line tool or Visual Paradigm plugin. Every time the tool or plugin starts, it is parsed and loaded into the pattern model. The pattern model is further utilized in the process.

4.2 Server

This section describes the implementation of the server part of the tool in more detail. Main task of the server part is to communicate with the database to load the patterns, host web interface to allow users manage the patterns and export patterns via the API.

The server is implemented in Java and is using Java Servlets as a backbone technology. It follows the Model/View/Controller MVC architectural pattern that sets basic rules for user interface implementation. The overall implementation architecture is shown in the figure 4.4. The *model* part of the server is represented by the *pattern model* designed in 3.5.1. Data from the database are loaded into the pattern model through *SQL* persistence. The pattern model data are passed to the view.

View prepares and organizes the data that are presented to the user using both manipulation with the model and the Freemarker *templating engine*. Using the templating engine allows to clearly separate HTML and CSS formatting from the data model. Data are passed to the template as a list of objects and template contains references to the objects. Templates can be also referenced or included amongst each other. This allows to share the common page implementation such as menu and header. They are stored separatelly and the formatting code does not repeat at any point. The rendered page is transfered to the

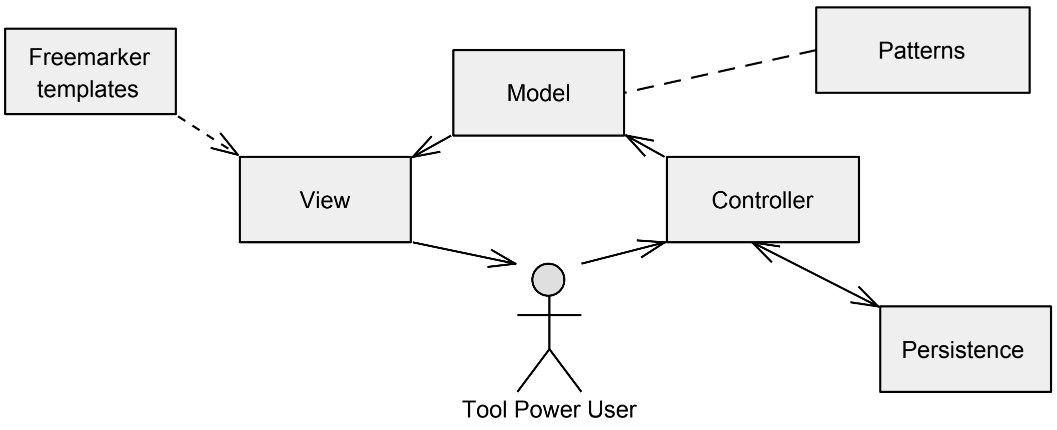


Figure 4.4: Diagram: MVC Architecture

client and displayed in his browser.

This section describes the method of routing pages that is implemented in the tool server. The overview of the routing process is shown in the figure 4.5.

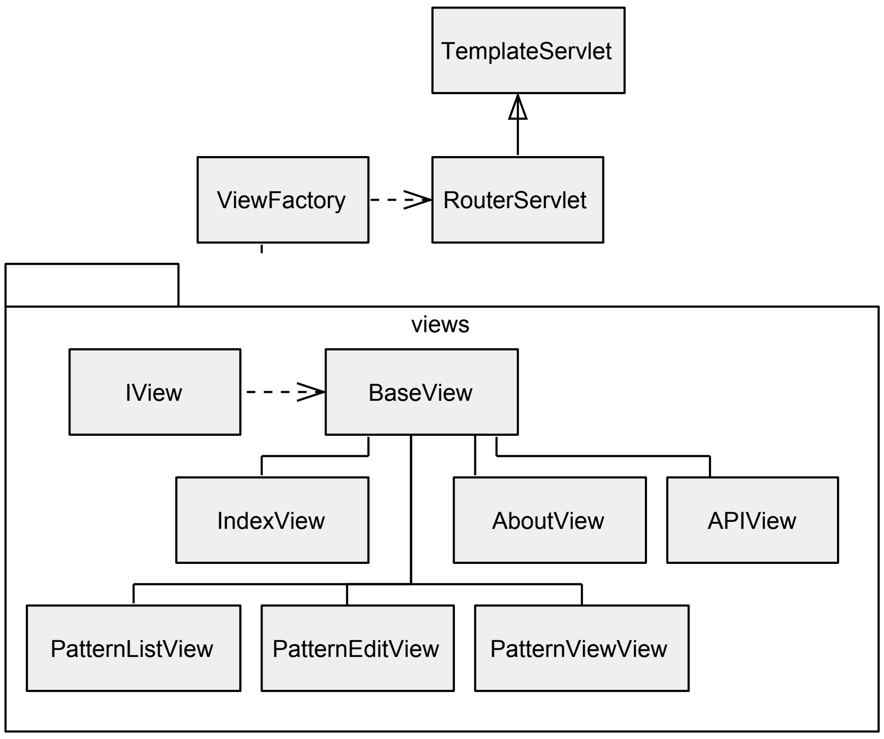


Figure 4.5: Diagram: Servlets and Views

The request to view a page from the user is routed directly to the `RouterServlet`. The servlet parses the request URL and chooses appropripate view using the `ViewFactory`. The views are stored in the separate package. Every view implements the interface `IView`. The core functionality is in the `BaseView`. The views are organized to several layers. The first one is more general and handles the basic pages like `Index`, `About` and the `API`. The `PatternViews` handle the pattern management and offer all standard functions - create, view, edit, delete.

The server part uses templates to separate the HTML and CSS visual attributes from the information. Overview of the template structure is displayed in the figure 4.6. The templating method allows having the same information displayed on more pages with other formating just by changing the template. There are two types of template `page-html` and `page-embedded`. They both share the common `page` so they have the same title and metadata. However `page-html` does contain formating information fo the experience in the web browser. On the other hand `page-embedded` is more minimalistic and displays the content without the menu. This allows to embed the pattern detail page directly in the user interface of Visual Paradigm plugin. The rest of the templates follow similar multi layer approach as views.

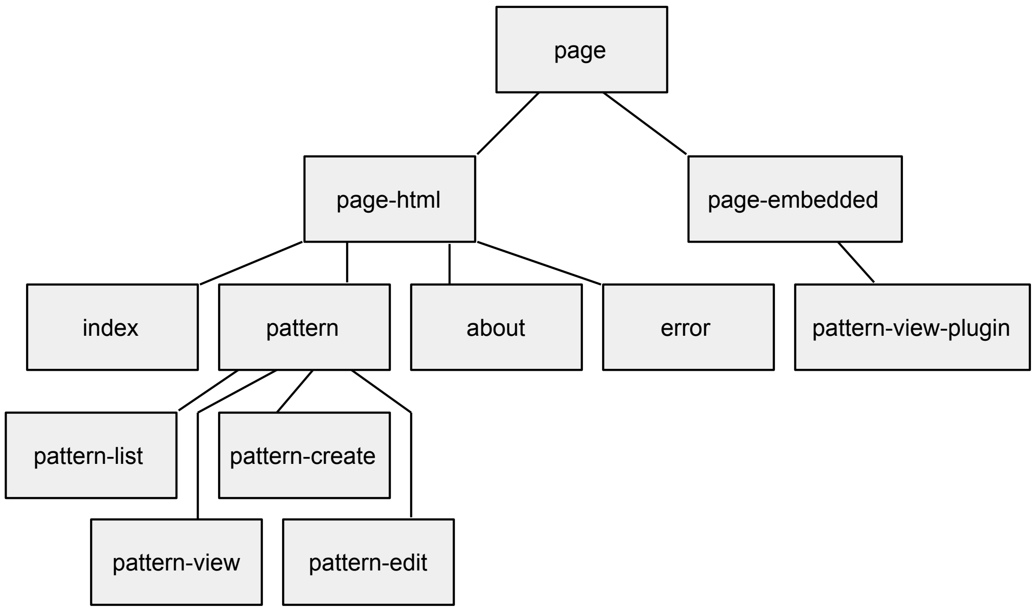


Figure 4.6: Diagram: Templates

The server component is designed and implemented to be easily extensible since it may require some degree of customization when the tool should be used in specific situations like hosting on a local network. The server part of the application is built into and can be distributed in WAR archive. This archive is a package that contains all classes, servlets, templates and other static files. Packages can be easily distributed and uploaded to be hosted on a server.

4.3 Client Tool

The functionality same for both command line and Visual Paradigm plugin is the updateable pattern database. The database distributed with the application is stored in the XML specified in section . Every time the pattern model is needed, the tool parses XML and loads patterns into the model. When the user requests the pattern database update, the tool connects to the server and downloads the up to date XML from the server API URL. The localy stored pattern database is overriden and next time pattern model is loaded it is up to date.

Both command line tool and Visual Paradigm plugin have two main parts - the user interface and the core. The *user interface* communicates with the user, parses and handles the input and presents the output to the user. The main functionality is wrapped in the *core* of the tool. The user interface communicates with the core via a set of models or data structures that carry the data. The core than does the correctness check using the QVT transformation and returns the result model back to the user interface that can present it to the user.

4.3.1 Command Line Interface Tool

Command line tool is designed to process input XMI and return correctness analysis results, see the command line tool design at section 3.3.

The input is the XMI file that need to be parsed before it can be used in the tool. The XMI file is XML so it can be easily parsed using standard libraries. After parsed, the data are stored in the UML model. The UML model can be passed further and processed. The output of the command line tool is the result model that returns from the QVT transformation. It is presented in the form of a table that is formated by whitespace. An example application output is shown in the appendices.

4.3.2 Visual Paradigm Plugin

Visual Paradigm plugin utilizes the Visual Paradigm plugin OpenAPI. The API is wrapped inside of the library `openapi.jar`, that is required to implement and compile the java classes. The plugin API allows the tool to create its own application menu inside the main Visual Paradigm menu. The menu is defined in specially formated XML readeable by Visual Paradigm. The workflow of the plugin is very similar to command line tool workflow.

Plugin API allows display of custom implemented java UI objects. Results are presented in a form of table that contains a binding column. Bindgin columns contains references to real model elements, including an icon. After user clicks on the reference, target model element is highlighted. The plugin needs to implement at least the VPPlugin and one VPActionController interface declared by the OpenAPI as show in the figure 4.7.

The Visual Paradigm plugin is not build as an usual Java application into a JAR. Result of the build process is instead an directory, that contains the files in the correct structure required by the Visual Paradigm. The directory structure is described in the section 2.9.2. The classes directory contains all compiled Java classes. The lib directory contains the Medini QVT engine libraries needed for the tool to work. The plugin configuration is stored in the `plugin.xml` file. The whole directory can be easily packed and distributed as a package. When using locally, the installation is just a simple file copy to the application folder and plugin subfolder.

4.4 Core of the Tool

The core of the client application is wrapped inside the library.

Patterns are stored in the locally available XML. When the pattern database is needed, the XML is parsed and *pattern model* is created. When the correctness analysis is run, the pattern model is passed to the tool core, because it is required by the QVT engine. The core than further processes the pattern model and creates a `QVT transformation`.

4.4.1 Pattern and QVT Relation

Each pattern in the database is defined with a QVT relation stored in the code attribute. The default pattern matching QVT relation can be defined as:

```
   top relation Pattern {

    /* UML pattern */
    checkonly domain source _element:Element {
      name = _name:String {}
    };

    /* Occurence */
    enforce domain target _pattern : umllint::Pattern {
      id = 'ID',
      name = 'Pattern',
      binding = _binding : umllint::Binding {
        name = _name,
        element = _element
      }
    };

    when {
      CONDITION
    }
  }
```

The whole pattern QVT relation is wrapped inside the relation statement. The relation also contains a name of the relation - `Pattern` in the example. The pattern matching QVT relation has three basic sections - the source domain, the target domain and the optional when part.

The `domain source` usualy defines the primary UML element the pattern matches on and binds it onto a variable - element of type Element and variable `element` in the example. It is checkonly, because the presence of the UML pattern in the source model does not need to be enforced.

On the other hand, the `domain target` defines the pattern occurence using the result model. The root element is allways a `umllint::Pattern`. It contains the basic metadata like pattern ID and name. These are important to define correctly, since they will ensure the correct results. The `umllint:Binding` element specifies the occurence in results. It presents a relation to the element from the UML model. The `name` is available to be specified by the pattern developer and the `element` needs to contain a reference on the UML element the pattern matched on - so in the example, there is `element`. The `binding` elements can be recursivelly nested when defining more advanced patterns.

The third optional part of the QVT relation is `when`. It is usually present, if the pattern should be matched only if a condition specified inside is valid.

4.4.2 QVT Transformation

Before the QVT transformation is run in the QVT engine, the single QVT relations provided by the patterns are merged into the QVT transformation.

```
transformation umllint(source:uml, target:umllint) {

  top relation Pattern1 {
    checkonly domain source _classifier:Classifier {
    };
    enforce domain target _pattern : umllint::Pattern {
    };
  }

  top relation Pattern2 {
    checkonly domain source _classifier:Classifier {
    };
    enforce domain target _pattern : umllint::Pattern {
    };
  }

}
```

The QVT `transformation` is identified by a name `umllint` and specifies both `source` and `target` model. The source model is `uml` and the target is `umllint`. When the QVT transformation is passed to the QVT engine, the name of the transformation and the direction needs to be specified. Name of the transformation is `umllint` and the direction is always target - the QVT transformation needs to take UML, match the patterns and produce the result model.

The QVT engine that executes the QVT transformation is provided by the Medini QVT libraries. The authors also documented how to use their QVT engine from custom Java applications [7].

4.4.3 Library

This section describes the contents of project library covers the core functionality of the application. All parts of the application including command line tool and Visual Paradigm plugin wrap the functionality of the library and add a simple logic with user interface.

Command line tool accepts input in a form of XMI files. Tools to parse and generate XMI are a part of the library, so if it is required server part of the tool can be easily converted to accept and process XMI files as well.

Pattern DB is interchaged amongst server and both tools in XML format. Tools to convert between XML and model representation of Pattern model are available in the library as well.

Library also contains all functionality covering the generation of QVT script from Pattern model and wraps model transformation core. This fact allows existence of more user interfaces and also makes possibility to easily create a new interface utilizing the same user interface. New functionality can be also added to server part of the tool so that it can perform the analysis as well. Core of the library is the QVT processing and model transformation engine. For this task, Medini QVT libraries are used.

4.5 Development Process

The project uses source code versioning system GIT. The code of the project is split to multiple repositories, so that it can be managed easily. Also it is clear, what parts the application consists of. Since this report is written in LaTeX, there is also a repository with source codes of this document. Also there is a repository reserved for article and presentation for EEICT student competition that this project took a part in. All repositories are listed in described in the table 4.8. Content of all repositories is available online at umllint.net.

4.5.1 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.

4.5.2 Build System

Most common build systems for all kinds of Java project is Ant. Gradle is a build tool that takes advantage of scripting language Groovy over Ant that uses static XML files. That allows the developer to create customizable and extensible build configurations very easily. Groovy language is based on Java so it is quite familiar already and all libraries can still be downloaded from standard Maven repositories.

Each of the project modules will use its own build script, since they can be distributed and deployed separatelly. Some of the build will have different output artifacts, but that is easily achieved via Gradle plugins so server application is built directly into the WAR archive and library into the standard JAR. Overview of the project builds and result artifacts is listed in the table. More detailed manual on how to run the build is available in appendices or on attached medium.

4.5.3 LaTeX Documents

This document is the exception when using Gradle, since it uses the make tool and Makefile build configuration as suggested by the thesis template. It has been slightly modified so it can handle other tasks as well. As with other projects, the complete reference is available in apendices or on attached medium.

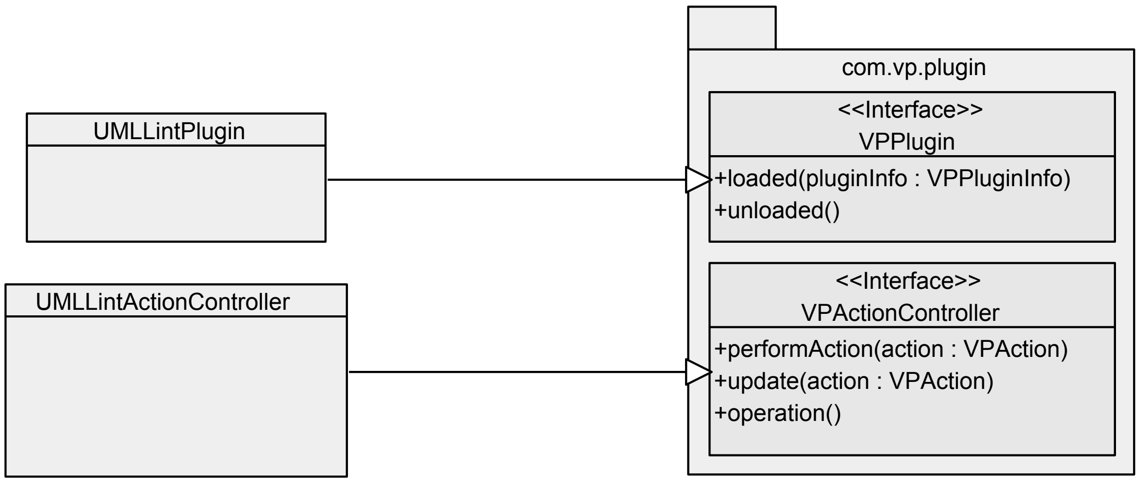


Figure 4.7: Diagram: Plugin Implementation

Repository ID	**Detail**
umllint-eeict	Article and presentation for Student EEICT 2014.
umllint-lib	Project library containing core functionality and models.
umllint-server	Server part of the tool.
umllint-thesis	LaTeX project of this document.
umllint-tool	The command line tool.
umllint-vp	Visual Paradigm plugin.
umllint-web	Static web page containing basic information abouit this project.

Figure 4.8: Table: List of GIT repositories

Project	Build	Build artifacts
umllint-eeict	Make - Makefile	eeict-dlouhy.pdf
umllint-lib	ANT - build.xml	umllint-1.0.jar
umllint-server	ANT - build.xml	umllint-server-1.0.war
umllint-thesis	Make - Makefile	dp-dlouhy.pdf
umllint-tool	AND - build.xml	umllint-tool-1.0.jar
umllint-vp	ANT - build.xml	umllint-vp-1.0.zip

Figure 4.9: Table: Application build systems and artifacts

4.5.4 Libraries

Server part makes use of standard Java web application Servlet libraries. For connection to PostgreSQL database library postgresql 0.3 JDBC4 is used. Templates are handled by Freemarker library. The most important component is the project library that contains the Pattern model and XML persistence utilities.

Application settings such as database connection data are stored in property file. Several propety files can be created for different configuration of the application - during development local.properties represented local development enviroment and production.properties the final hosted server application will be running on.

4.5.5 Libraries and License

This project use the Medini QVT plugin and related required Eclipse libraries. All of the libraries are licensed under the license Eclipse Public License and are free for non commercial use.

```
de.ikv.medini.metamodel.xsd2ecoreutil.edit_1.0.0.25263.jar

de.ikv.medini.metamodel.xsd2ecoreutil.editor_1.0.0.25263.jar

de.ikv.medini.metamodel.xsd2ecoreutil_1.0.0.25263.jar

de.ikv.medini.qvt.debug.core.jar

de.ikv.medini.qvt.debug.ui.jar

de.ikv.medini.qvt.examples_1.0.0.25263.jar

de.ikv.medini.qvt.help_1.2.0.25263.jar

de.ikv.medini.qvt.plugin_1.4.0.25263.jar

de.ikv.medini.qvt.product_1.6.0.25263.jar

de.ikv.medini.qvt.ui_1.2.0.25263.jar

de.ikv.medini.util.core.plugin_1.2.0.25263.jar

de.ikv.medini.util.eclipse.nls_1.2.0.25263.jar

de.ikv.medini.util.eclipse_1.2.0.25263.jar
```

```
org.eclipse.core.expressions_3.4.200.v20100505.jar

org.eclipse.core.runtime_3.6.0.v20100505.jar

org.eclipse.core.variables_3.2.400.v20100505.jar

org.eclipse.emf.common_2.4.0.v200808251517.jar

org.eclipse.emf.ecore.change_2.5.1.v20100907-1643.jar

org.eclipse.emf.ecore.xmi_2.4.1.v200808251517.jar

org.eclipse.emf.ecore_2.4.1.v200808251517.jar

org.eclipse.emf.edit_2.6.0.v20100914-1218.jar

org.eclipse.emf.transaction_1.4.0.v20100331-1738.jar

org.eclipse.emf.validation_1.4.0.v20100428-2315.jar

org.eclipse.equinox.common_3.6.0.v20100503.jar

org.eclipse.equinox.registry_3.5.0.v20100503.jar

org.eclipse.osgi_3.6.1.R36x_v20100806.jar
```

Chapter 5

Testing

This chapter covers testing of the tool, specifies the enviroment and library versions that the tools was tested on and also establishes a conclusion based on the test results. The project uses several levels of testing.

1. Unit testing
2. Functional testing
3. System testing

First level covers testing of important inner components and logic using unit testing. Second level is the functionality testing. Final level of testing uses the complete product and performs system tests. Important consideration is also the use of automated testing since the pattern database can be too sizable to test manually.

5.1 Unit Testing

BeforeClass and AfterClass annotation can be appended to classes that do not need to be static. Annotations are more clear in general. Test suite in TestNG can be defined easily by XML files that specify which classes are part of the test suite. Test classes can be also grouped together using group annotations. Test management in TestNG is very flexible and allows detail configuration of test runs. TestNG also introduces test dependecy - although it is generally considered bad practice since it is against on of unit testing principles it might be usefull in some situation. All these reasons speaks for using TestNG over JUnit so the choice for this project is clear.

In this project tests implemented in TestNG are located in dedicated package. They cover all units (classes) that contain critical application logic so that it can be ensured that the core of the application works correctly. There is no reason to cover data transfer objects or classes that belong to the model that do not contain any application logic. The whole unit test package can be executed to make sure, that the application core is correct.

The Gradle build system allows integration of test to the build plan. So every time the application is build, all the tests are execute as well. Result of the test analysis is not just a summary printed by the build system, but also a detailed report including possible error detection and stack trace.

All of this helps not just to verify the final product but also during development to fix bugs, but also some of the parts or logic is easier to implement using test driven development.

Project	Section	Coverage
Server	Persistence and DB	Login data property file, connection, running SQL
	Routing	Parse the request path, routing to view
	Views	View Factory returning views based on the routing info
	XML	Parsing and rendering from/to XML
Tool	Input	XMI Parsing and rendering
	Pattern Database	Update and load the pattern database
	Output	Output rendering
	Result model	Result model rendering

Figure 5.1: Table: Unit Test Coverage

5.2 Functional Testing

Functional Testing that covers a specific component by the specifications. These tests will focus just on the UML pattern search part of the system. As described in , functional testing test cases most often specify input and expected output. This translates to the UML diagram input in a form of XMI files and pattern matching result output. Each isolated pattern will be tested by a specific UML diagram, or part of the diagram specified in XMI. Expected output of the tool is the matched tested patterns. This approach can be also easily automated by preparing XMI files and match results with prepared template output files.

The similar testing will be also performed on the Visual Paradigm plugin, but it needs to be done manually since Visual Paradigm does not offer an API for tests. However if all tests in the tool pass, Visual Paradigm plugin uses the same core component that is already tested by the automated functional testing.

Another functional test will be performed on the XML database, that is generated on the server part of the tool. The XML validity can be easily verified by attached XSD file. XSD can be also utilized to verify the input XMI tests.

5.3 System Testing

System testing main purpose is to test, if the integrated system complies with the specified requirements, see Chapter 2. The requirements are represented by the Use Case diagram. Based on the requirements, set of test cases were defined. System testing is a black box type of testing that covers testing of the system from the user point of view, without knowing its inner structure. System testing is performed by manually executing predefined test cases based on the use case specification.

5.4 Application Versions

This section describes a reference enviroment - list of all libraries and utilities the application was tested on. Application versions and minor application versions not specified in the list default to most up to date version as of 2014-05-20.

	Version
Java	1.7.0 51
Freemarker	2.3.20
PostgreSQL JDBC4	9.3-1101
PostgreSQL Server	9.3
Apache Tomcat	7.0.52
TestNG	6.8.8
Hamcrest	1.3
Gradle	1.1.1
XMI	2.1
XML	2.1
UML	2
Git	1.8.3
JetBrains IDEA	13.1.2
TexLive	2013
Medini QVT IDE	1.7.0
Visual Paradigm Standard Edition	10.2
Visual Paradigm Community Edition	11.2
Windows	7
CentOS	6.5

Figure 5.2: Table: Reference Enviroment Versions

5.5 Test result evaluation

The tests were heavily used during the development of the tool. Unit tests can be included into the development process via test driven development, as demonstrated on the example of Route parsing problem. First of all the route parser class and interface was created. After this, unit test covering all possible route formats can be created so that all cases are covered. As mentioned in the unit tests sections, unit tests were also a part of the build so having a successfull build signifies that all tests passed. Before any release of the tool version, but mainly the delivery all test need to pass.

Chapter 6

Evaluation

This chapter evaluates the result of this thesis from multiple points of view. Initial goal was to create a tool for checking the correctness of UML design diagrams. This project introduces a multi component tool that allows users to use command line interface or Visual Paradigm plugin. Both interfaces share the same validation data that can be edited via a web interface.

The pattern matching approach for UML correctness validation was chosen, because it is generic enough for using in different scenarios, yet it can be specific enough to handle the advanced problems as well. The patterns are also more accessible to the user and this allows them to define their own.

The assignment suggested the use of OCL for the correctness check. However the analysis revealed, that the QVT language is more powerfull especially in combination with the incorrectness pattern approach and that makes it much more suitable for the task. Since QVT works on top of the UML model, any incorrectness pattern that can be expressed in the UML can be defined in QVT and used in the tool. Using QVT as the pattern definition platform. But using QVT in the tool has a downside as well. There are not many good QVT engines that can be integrated in custom tools.

There are plenty of tools that support the XMI format. However due to the changes, different revisions and specification versions some of them may not be compatible with this tool. On the other hand many of the available software modeling tools are based on the Eclipse platform and use the same components to work with XMI as well and that ensures compatibility, because this tool is tailored mostly on the Eclipse style output.

Because the XMI ambiguity and compatibility the Visual Paradigm plugin was introduced. The plugin extracts the basic UML model directly from the active diagram, so it is not dependent on the XMI syntax.

The results of this project are published publicly as open source and are available to anyone. For this purpose the home page of the project was created [1]. It contains all components of the tool including a manual and this document as a complete reference. The pattern database is also integrated inside the page, so everyone can test the software without installing their own server or manually editing the XML pattern database. The instalation of the software is easy enough. The command line tool can be distributed directly in the binary jar and the instalation of the Visual Paradigm plugin means just unpacking it to the right directory.

This thesis is written in English, because the software design community intrested in the

[1] `<http://umllint.net>`

UML correctness validation is not very big. All project documentation, homepage, manuals are written in English as well. The project was also presented on the Student EEICT 2014 and the article was published in the conference proceedings.

Both user interfaces of the tool are as simple as possible so that the tool is intuitive and easy to use. The Visual Paradigm plugin offers one click correctness analysis that is also reasonably fast so that it can be used during the development without disrupting the workflow. It can assist designers in their everyday task to check if they did not miss an error or as a learning aid to beginners to guide them into the world of UML. Thanks to the interface and argument layout, the command line application can be also used as a part of the script that will integrate the tool into another system. In the final application, it can be a part of continuous integration tools or just run on top of XMI to validate the UML model or just check that the UML model applies to custom company policies.

The shared pattern database encourages the development of new patterns and new cases of UML incorrectness. It also enables everyone to update and use recent validation data without any additional manual installation. When required, the patter database can be also deployed on a custom server or local network. The database schema and server architecture enable easy extension. Due to the general design and technology used, the tool is capable of general pattern detection on MOF based models.

Due to the component composition, extension of the tool is not difficult. Future development may bring yet another user interface, possibly with more reliable input. The server and database component should be extended with the user management system, if the tool is widely deployed. This would also introduce a possibility to true custom patterns, pattern rating, recommendation and comentary. More research in the future will show, if there is also a possibility to use the tool QVT transformation to directly fix the incorrect patterns.

Chapter 7

Conclusion

The goal of this thesis was to create a tool for checking correctness of design diagrams in UML modeling language. After initial analysis of UML and related languages and standards the approach of incorretnes patterns was chosen to be used for the correctness evaluation. The incorrectness patterns are stored in the SQL database. The database is shared and allows users of the tool to collaborate on creating and extending the quality of the corretness validation. Power users can manage the patterns via web interface provided by the server part of the tool, the changes are reflected to the client tools.

Two client user interfaces for correctness evaluation were implemented. The first one is targeted on systems using command line, accepts the UML model in interchangeable format and results of the analysis are published in text. The second client user interface is implemented as a plugin to widely used Visual Paradigm UML modeling software. Both of the user interfaces share common functionality in the shared library. The shared library also allows users to easily create another user interface or adapt the tool to be used in existing system. The tool is published on its own website that contains all information and tutorials on how to use it.

The result tool is very flexible so that it can be used in many different scenarios. Also the evaluation shows, that if the patterns are defined correctly and thoroughly, it does provide valuable results and feedback. The tool makes also use of existing pattern catalogues that were created after years of research for mainly educational purposes and provides a way to utilize them in every day software modeling. Moreover due to architecture and inner logic it is very univerzal and can be easily extended or customized for different scenarios. The design is so general, it can handle not only correctness check, but basically any pattern matching task on UML diagrams. This project was also presented on the Student EEICT 2014 and was published in the conference proceedings.

Main problems and obstacles during the development were mostly caused by the ambiguity of specifications. The UML language including XMI interchange format and the QVT language is specified by the Object Management Group, however most of the vendor tools do heavily customize the XMI export. The support for exchanging the visual side of UML diagrams is freshly standardized and most commercial software modeling products will take years before it is fully supported. Also the QVT set of languages that is very capable and can be used for model transformation is not very widely used and there is no complete reference implementation as library that allows the integration to user tools. The best available tool to work with QVT and integrate it in custom applications is the Medini QVT extension, however it is slowly becoming outdated, since the evolution of XMI and UML will go forward in the future.

Due to the fact, that the tool uses common library it is not so difficult to customize it and use another QVT evaluation engine in the future. Also the models and parsers can be customized so that they support recent versions of XMI and UML. More ambitious future extension would be to utilize the model transformation not only to detect the incorrectness patterns in UML, but also to fix the errors directly in the model. The tool could be also extended to evaluate the visual side of the UML diagram, since the representation standardized recently by OMG also uses the XMI model. The server and database component of the tool can also be extended with the user management allowing personalized custom patterns, pattern rating, communication and other methods that support collaboration. But the main opportunity to contribute to the tool is and always will be the pattern database. The more patterns are in the database, the more quality result the tool produces.

The main use of this project is to assist the software engineer during the software modeling proces. There are several ways to do that. The plugin for Visual Paradigm can be utilized during the diagram design process so that it continually improves. Together with the command line interface it can also validate complete models as a part of continuous integration during the iterations or to check the final UML design. Extending the tool with the set of custom patterns can also help enforcing the company policies and rules of design. It can also be integrated to he univerzity learning system to assist junior software designers during their education. Thanks to the general logic and pattern based approach it can be customized to any number of different uses.

Modeling certainly is an important part of the software development process since it does heavily influence the following implementation and result product quality so it should not be neglected. This tool offers the help during this process and is a good starting platform for better software design.

Bibliography

[1] Rodrigo F. Araujo, Vinicius H. S. Durelli, and Rafael M. Teixeira. *Getting Started with Eclipse Juno.* Packt Publishing, 2013.

[2] Mira Balaban and Azzam Maraee. The Pattern-Class-Diagram (PCD) Language. 2013.

[3] Mira Balaban, Azzam Maraee, and Arnon Sturm. Management of Correctness Problems in UML Class Diagrams - Towards a Pattern-Based Approach. *International Journal of Information System Modeling and Design*, 2010.

[4] Mira Balaban, Azzam Maraee, Arnon Sturm, and Pavel Jelnov. A Pattern-based Approach for Improving Model Quality. *Software & Systems Modeling*, pages 1–29, 2014.

[5] BGU Modeling Group. Patterns, Anti-Patterns and Inference Rules Catalog for UML Class Diagrams. `<http://www.cs.bgu.ac.il/~cd-patterns/>`, 2014. (accessed October 25, 2013).

[6] Korry Douglas and Susan Douglas. *PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases.* SAMS publishing, 2003.

[7] Hajo Eichler. How to run transformation from my Java application? `<http://projects.ikv.de/qvt/wiki/integration>`, 2007. (accessed February 12, 2014).

[8] Maged Elaasar. *An Approach to Design Pattern and Anti-pattern Detection in MOF -based Modeling Languages.* PhD thesis, Ottawa, Ont., Canada, Canada, 2012. AAINR93678.

[9] Maged Elaasar, Lionel Briand, and Yvan Labiche. A Metamodeling Approach to Pattern Specification. In *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 484–498. Springer Berlin Heidelberg, 2006.

[10] Maged Elaasar, Briand Lionel, and Yvan Labiche. Metamodeling Anti-Patterns, 2010. (accessed October 12, 2013).

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

[12] Maged Elaasar, Briand Lionel, and Yvan Labiche. Domain-Specific Model Verification with QVT. In Robert France, Jochen Kuester, Behzad Bordbar, and

Richard Paige, editors, *Modelling Foundations and Applications*, volume 6698 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2011.

[13] Martin Fowler. *UML Distilled.* Addison-Wesley, third edition, 2004.

[14] Timothy Grose, Gary Doney, and Stephen Brodsky. *Mastering XMI: Java Programming with XMI, XML, and UML.* Wiley, April 2002.

[15] Lex Heerink. Medini QVT Relations Tutorial. `<http://projects.ikv.de/qvt/wiki/tutorial>`. (accessed February 15, 2014).

[16] IKV++ Technologies. Medini QVT. `<http://projects.ikv.de/qvt>`, 2012. (accessed February 10, 2014).

[17] Jörg Kiegeland and Hajo Eichler. Medini QVT Workshop. `<http://projects.ikv.de/qvt>`, 2007. (accessed February 12, 2014).

[18] Jernej Kovse and Theo Härder. Generic XMI-Based UML Model Transformations. In *Proceedings 8th of International Conference on Object-Oriented Information Systems OOIS'02*, pages 192–198. Springer-Verlag, 2002.

[19] Kevin Lano. *UML 2 Semantics and Applications.* Wiley, 2009.

[20] Azzam Maree and Adiel Ashrov. Class Diagram - Syntax and Semantics. 2013.

[21] Azzam Maree and Adiel Ashrov. Class Diagram - Semantics. 2014.

[22] Ed Merks. Introduction to the Eclipse Modeling Framework. `<http://mdsd08.techjava.de/emftutorial31.pdf>`, 2008. (accessed May 1, 2014).

[23] Russell Miles and Kim Hamilton. *Learning UML 2.0.* O'Reilly Media, 2006.

[24] Glenford Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* Wiley, third edition, 2012.

[25] Regina Obe and Leonard Hsu. *PostgreSQL: Up and Running.* O'Reilly Media, 2012.

[26] Object Management Group. Diagram Definition Standard. `<http://www.omg.org/spec/DD/1.0/>`. (accessed March 10, 2014).

[27] Object Management Group. Meta Object Facility Query/View/Transformation. `<http://www.omg.org/spec/QVT>`, August 2011. (accessed March 21, 2014).

[28] Object Management Group. UML Revision Task Force. `<http://www.omgwiki.org/uml2-rtf/doku.php?id=start>`, 2011. (accessed May 2, 2014).

[29] Object Management Group. Unified Modeling Language^{TM} Standard. `<http://www.omg.org/spec/UML/>`, 2011. (accessed January 5, 2014).

[30] Object Management Group. XML Metadata Interchange. `<http://www.omg.org/spec/UML/>`, August 2011. (accessed February 5, 2014).

[31] Object Management Group. About the Object Management Group. `<http://www.omg.org/gettingstarted/gettingstartedindex.htm>`, August 2013. (accessed March 15, 2014).

[32] Object Management Group. Meta-Object Facility Standard. <http://www.omg.org/mof/>, 2014. (accessed March 11, 2014).

[33] Dan Pilone. *UML 2.0 in a Nutshell.* O'Reilly, 2005.

[34] The Eclipse Foundation. Eclipse Modeling Framework Project EMF. <http://www.eclipse.org/modeling/emf/>, 2014. (accessed May 2, 2014).

[35] Visual Paradigm. Features. <http://www.visual-paradigm.com/features/miscellaneous/>. (accessed December 21, 2013).

[36] Visual Paradigm. Model Quality Checker. <http://www.visual-paradigm.com/product/vpuml/tutorials/modelquality.jsp>. (accessed December 19, 2013).

[37] Visual Paradigm. Plug-in Development. <http://www.visual-paradigm.com/product/vpuml/tutorials/plugin.jsp>. (accessed December 17, 2013).

[38] Visual Paradigm. User's Guide. <http://www.visual-paradigm.com/support/documents/vpuserguide.jsp>. (accessed December 15, 2013).

Appendix A

Contents of the CD

umllint-thesis/	Documentation.
umllint-thesis/src/	Documentation source code.
umllint-thesis/dlouhy-dp.pdf	This document.
umllint-lib/	Library with common functionality.
umllint-server/	Server pattern management system.
umllint-database/	The pattern SQL database.
umllint-tool/	Command line tool.
umllint-vp/	Visual Paradigm plugin.
umllint-web/	Project home page.
umllint-eeict/article/	Published article for EEICT 2014.
umllint-eeict/leaflet/	Leaflet for EEICT 2014.
umllint-eeict/presentation/	Presentation for EEICT 2014.
README	Contents of the CD.

Appendix B

Glossary

As the text of this thesis shows, the software modelling world use several various languages, standards, methods and applications. Most of them are commonly referenced and known just by abbreviations of their name. To avoid confusion and provide a reference for the reader, the ones used in the text of this thesis are translated below.

ANT	Tool build process automation
CLI	Command Line Interface
CSS	Cascading Style Sheets
DD	Diagram Definition
Ecore	Metamodel in EMF
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
Gradle	Project automation tool
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JAR	Java ARchive
JRE	Java Runtime Enviroment
JSP	Java Server Pages
Medini QVT	A tool set for model to model transformations
MOF	Meta-Object Facility
MVC	Model/View/Controller
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query/View/Transformation Language
QVTr	QVT Relations Language
TestNG	A testing framework
UML RTF	UML Revision Task Force
UML	Unified Modeling Language
UML	Unified Modeling Language
WAR	Web application ARchive
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema

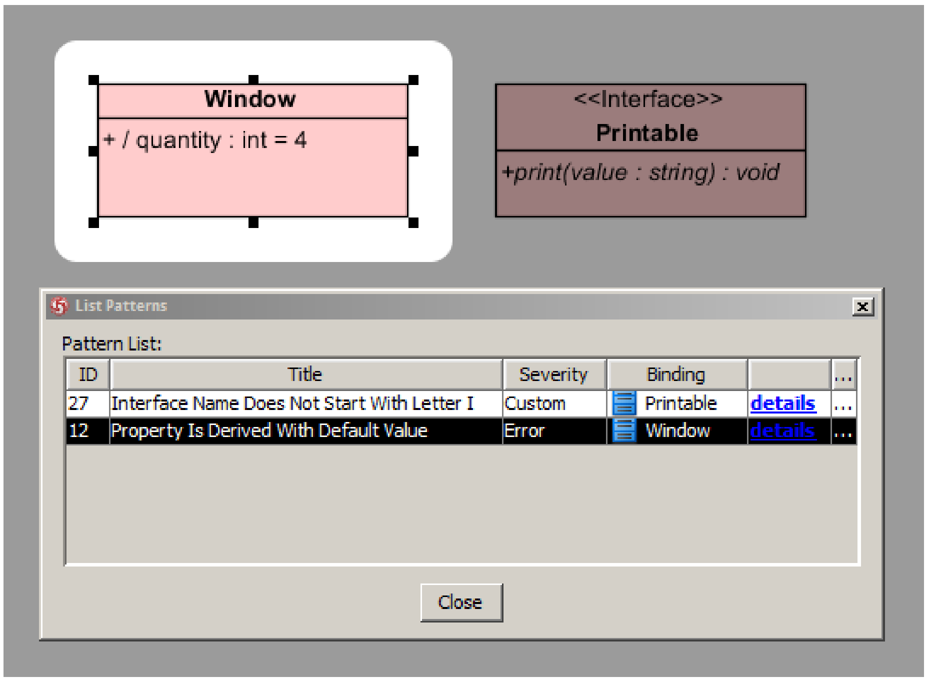
Appendix C

Screenshots

This appendix contains screenshots to put the text into a visual context. The screenshot C.3 describes the Visual Paradigm modeling software.

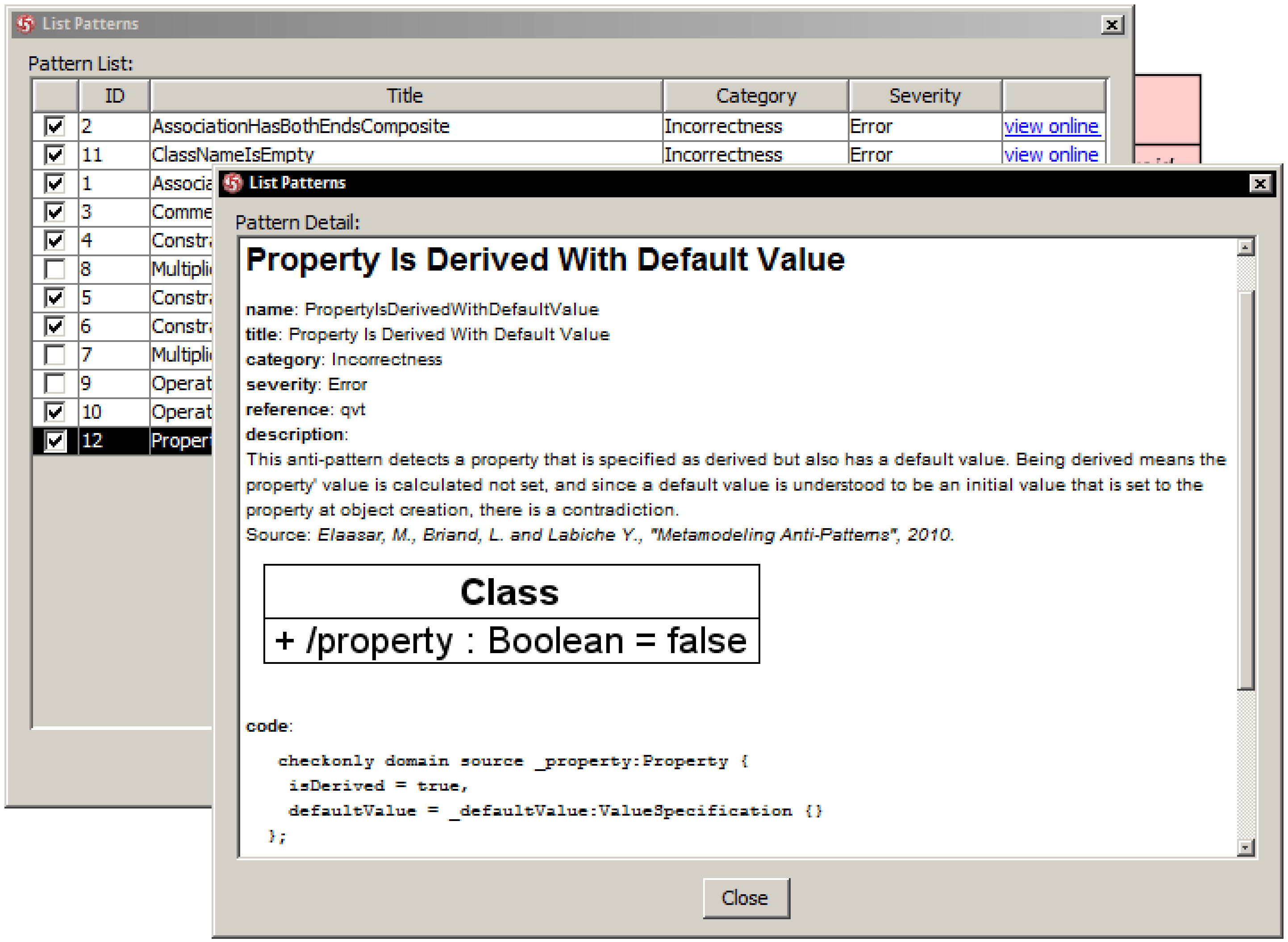
The screenshot C.2 shows the pattern database contents in the Visual Paradigm plugin. In the background, there is a table that lists all the patterns in the database. Each of the patterns is represented by ID, title, category and severity. The view online link opens a new dialog, that loads the pattern details directly from the server. The pattern details contain more useful information including possible advices on how to fix it.

And finally the screenshot C.1 shows the results of the correctness analysis. The output is similar to pattern database listing, but there is also a binding column that refers directly to the diagram element. When the column is clicked, the diagram element that contains an error get highlighted. Similar to the database listing, user can view the pattern details online.



ID	Title	Severity	Binding		...
27	Interface Name Does Not Start With Letter I	Custom	Printable	details	...
12	Property Is Derived With Default Value	Error	Window	details	...

Figure C.1: Screenshot: Visual Paradigm Plugin - Analysis Results



	ID	Title	Category	Severity	
✓	2	AssociationHasBothEndsComposite	Incorrectness	Error	view online
✓	11	ClassNameIsEmpty	Incorrectness	Error	view online
✓	1	Associa			
✓	3	Comme			
✓	4	Constr			
	8	Multipli			
✓	5	Constr			
✓	6	Constr			
	7	Multipli			
	9	Operat			
✓	10	Operat			
✓	12	Proper			

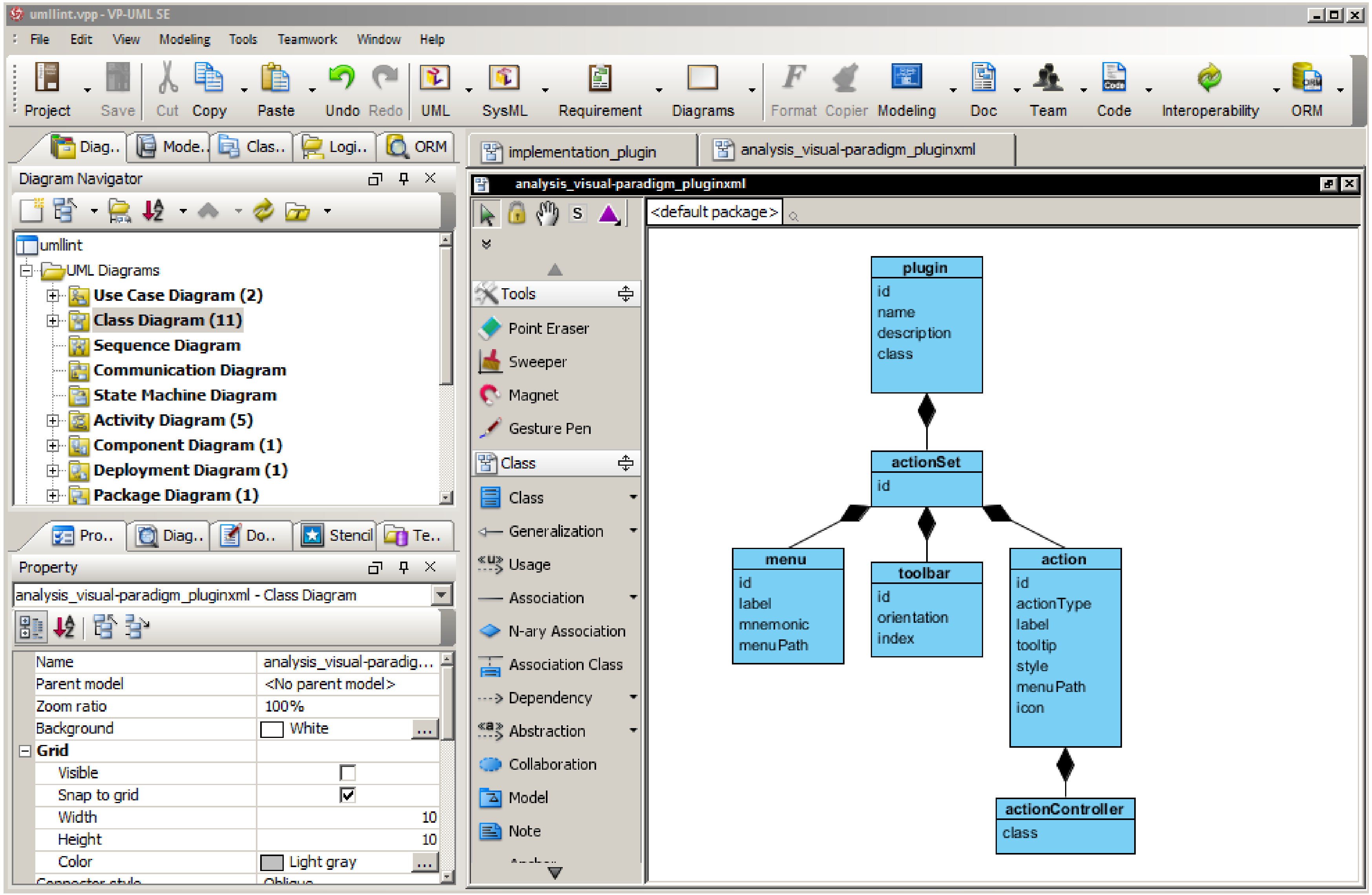
Property Is Derived With Default Value

name: PropertyIsDerivedWithDefaultValue
title: Property Is Derived With Default Value
category: Incorrectness
severity: Error
reference: qvt
description:
This anti-pattern detects a property that is specified as derived but also has a default value. Being derived means the property' value is calculated not set, and since a default value is understood to be an initial value that is set to the property at object creation, there is a contradiction.
Source: *Elaasar, M., Briand, L. and Labiche Y., "Metamodeling Anti-Patterns", 2010.*

code:

```
 checkonly domain source _property:Property {
  isDerived = true,
  defaultValue = _defaultValue:ValueSpecification {}
 };
```

Figure C.2: Screenshot: Visual Paradigm Plugin - List Patterns



analysis_visual-paradigm_pluginxml - Class Diagram

Name	analysis_visual-paradig...
Parent model	<No parent model>
Zoom ratio	100%
Background	White
Grid	
Visible	☐
Snap to grid	✓
Width	10
Height	10
Color	Light gray

Figure C.3: Screenshot: Visual Paradigm