MetricView Evolution

Monitoring Architectural Quality

by:

Martijn Wijns

Supervisors:

Dr. M.R.V. Chaudron (TU/e)
Ir. C.F.J. Lange (TU/e)

within the EmpAnADa project

Eindhoven, March 2006
Abstract

Current tooling used to analyze the quality of a software architecture focuses on a single version. The results of the metrics involved in this analysis are often visualized separately from analyzed artefact. With the creation of the MetricView tool [Ter05] it was shown that the Geographic Information System (GIS) approach to metrics visualization, in which metrics and artifact visualization is combined, is a viable alternative. We suggest two main extensions to this tool:

- Abstraction from metrics using Quality Models
- Enabling evolution analysis by supporting analysis of multiple versions of a model

In this document the process of extending this tool and the extension itself are considered. Furthermore, the results of a usability experiment, in which the tool was tested, are presented. This experiment showed that participants found the tool useful for program understanding, quality evaluation and maturity/completeness evaluation. Although there is room for improvement with respect to usability, it confirms our beliefs that MetricView Evolution is a useful contribution.
Preface

After the final presentation of my internship project on measuring operating system real-time performance I was approached by Dr. Michel Chaudron. He asked me whether I was interested in doing a masters project on the subject of visualization of quality of software architectures. Because visualization and software architecture both are areas of my interest it took me little time to decide that this was something I wanted to work on, especially after I was shown the MetricView tool [Ter05]. It was innovative and clearly had a lot of potential.

During my masters project I have tried to exploit this potential. This would not have been possible without the guidance and support of Dr. Michel Chaudron and ir. Christian Lange for which I am very grateful. I would like to thank Dennis van Opzeeland MSc., Marcel van Amstel and Rene Ladan for their help and ir. Louis van Gool for providing me with a good excuse to spend some time learning Python, which turned out to be very useful. Furthermore, I would like to thank all participants in the usability experiment for their time, effort and useful remarks. Last but certainly not least, I would like to thank Dr. Kees Huizing and Dr. ir. Reinder J. Bril for taking place in my examination board.

Martijn Wijns

System Architecture and Networking (SAN)
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Contents</td>
<td>vi</td>
</tr>
<tr>
<td>List of figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of tables</td>
<td>ix</td>
</tr>
<tr>
<td>Acronyms</td>
<td>1</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Goal</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Outline</td>
<td>4</td>
</tr>
<tr>
<td>2 State of the Art</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Existing Work</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Existing Tools</td>
<td>6</td>
</tr>
<tr>
<td>3 Architectural Evolution</td>
<td>9</td>
</tr>
<tr>
<td>3.1 Software Architecture</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Quality</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Evolution</td>
<td>12</td>
</tr>
<tr>
<td>4 Towards a Solution</td>
<td>15</td>
</tr>
<tr>
<td>4.1 Prototyped Ideas</td>
<td>15</td>
</tr>
<tr>
<td>4.2 The notion of tasks</td>
<td>23</td>
</tr>
<tr>
<td>5 MetricView Evolution - Requirements</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>5.2 General description</td>
<td>29</td>
</tr>
<tr>
<td>5.3 Specific Requirements</td>
<td>33</td>
</tr>
<tr>
<td>5.4 Use cases</td>
<td>35</td>
</tr>
<tr>
<td>5.5 Software requirements</td>
<td>40</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Screenshot of SDMetrics showing a Kiviat diagram</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Screenshot of MetricView</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of correspondence visualization</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>Metric Visualization for multiple versions: (prototype on the left, final implementation on the right)</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>Example Quality Tree</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Timeline showing trend in ‘Coupling between Objects’ values for 3 versions of a model</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>Lange Diagram showing inter-diagram relations</td>
<td>22</td>
</tr>
<tr>
<td>4.6</td>
<td>Context Diagram showing all the children of a single class</td>
<td>23</td>
</tr>
<tr>
<td>4.7</td>
<td>Lange Diagram showing clustering</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Tool environment</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Use cases: input</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Use cases: output</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>Use cases: functionality</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Global processing MetricView</td>
<td>41</td>
</tr>
<tr>
<td>5.6</td>
<td>Structure of the visualizer subsystem</td>
<td>42</td>
</tr>
<tr>
<td>5.7</td>
<td>Visualization uses visitor pattern</td>
<td>42</td>
</tr>
<tr>
<td>5.8</td>
<td>Model manager structure</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>A global analysis model of MetricView Evolution and its context</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Decomposition</td>
<td>51</td>
</tr>
<tr>
<td>6.3</td>
<td>Internal structure of Control, main classes</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>Focus interaction scenario using Model-View-Presenter pattern</td>
<td>58</td>
</tr>
<tr>
<td>6.5</td>
<td>Selection of a Diagram Element/Model Element</td>
<td>58</td>
</tr>
<tr>
<td>6.6</td>
<td>Internal structure of DataStorage</td>
<td>61</td>
</tr>
<tr>
<td>6.7</td>
<td>Classes related to QualityFunction</td>
<td>65</td>
</tr>
<tr>
<td>6.8</td>
<td>The evaluation of the QualityFunction of a Quality Attribute</td>
<td>66</td>
</tr>
<tr>
<td>6.9</td>
<td>Internal structure of Visualization</td>
<td>68</td>
</tr>
<tr>
<td>6.10</td>
<td>Refreshing the Canvas</td>
<td>70</td>
</tr>
<tr>
<td>6.11</td>
<td>Internal structure of Instrumentation</td>
<td>73</td>
</tr>
</tbody>
</table>
8.1 Usefulness for correctly doing a task ................................. 83
8.2 Usefulness for efficiently doing a task .............................. 83
8.3 Search and Highlight usability rating ............................... 84
8.4 Context View usability rating ......................................... 85

B.1 The Model View Controller pattern ............................... 102
B.2 The Model View Presenter pattern ................................. 102
B.3 A screenshot of a prototype of QualityTreeEditor .......... 105
List of Tables

5.1 Priority Levels ............................................................ 33
5.2 Artefact data ............................................................... 34
5.3 Analysis data ............................................................... 35
5.4 Output requirements ..................................................... 36
5.5 Functionality requirements ............................................. 44
5.6 Editing functionality requirements .................................... 44
5.7 Evolution view requirements .......................................... 45
5.8 Platform requirements .................................................. 45
5.9 Quality requirements .................................................... 46

8.1 Basic View size for various cases ..................................... 77
8.2 Basic Model size for various cases .................................... 78

C.1 Summed usefulness results. C: correctly doing a task, F: doing a task fast .... 110
C.2 Usefulness for correctly doing a type of task .......................... 110
C.3 Usefulness for efficiently doing a type of task ...................... 110
C.4 Understandability ratings ............................................... 111
C.5 Intuitivity rating ........................................................... 111
Acronyms

GIS  Geographic Information Systems
MVC  Model View Controller
MVP  Model View Presenter
SAAT  Software Architecture Analysis Tool
SQL  Structured Query Language
UML  Unified Modeling Language
XMI  XML Metadata Interchange
XML  eXtensible Markup Language
XSL  eXtensible Stylesheet Language
Chapter 1

Introduction

This introduction chapter consists of three sections. In the first section the motivation for doing the project is presented. Based on this motivation a goal is defined in the second section. The third an final section describes the structure of the rest of the document.

1.1 Motivation

There are two main themes at the base of this thesis, the first of which is the evolution of Software Architectures. A software architecture is a high level description of a software system. Typically this description is developed incrementally. It might start at a very high abstraction level to which more detail is added until a sufficient level is reached for detailed design and implementation. After the implementation, during the maintenance phase it is likely that the system will at some point be changed to meet some additional requirements. If these changes are severe enough the architecture itself has to be adapted to remain an accurate description of the system. The important thing to remember here is Software Architectures evolve.

The second theme for this thesis is the concept of software Metrics. Software Metrics are specifications for quantification of software attributes. They are considered a good thing because they can help with understanding, provide control and be used to make predictions [Fen96]. Metrics can help with understanding by offering a means to find parts of a system that have a specific property. Metrics can also provide control by making it possible to define rules and guidelines. An example could be: “Metric A should be below 7 at all times”. Finally, metrics can help with prediction. If metric data is collected for multiple versions of a model trends in this data could be extrapolated to make predictions about future versions.

Combining the information that software architectures evolve and that metrics can help with controlling and improving changing artifacts leads to the idea of using metrics to support the evolution of software architectures. The benefit of using metrics in the architectural phase of a software project is that it is much cheaper to fix problems there than it is to fix them during implementation. We expect that by looking at evolution
data one could get a better understanding of a software architecture and obtain greater control over its development and final quality. Doing this at the moment, however, is not an easy task. The tools that are available are not suited, which forms the main motivation for this project. For a more thorough explanation of the problems with the current tooling see chapter 2. For a more thorough explanation of the motivation of this project see chapter 3.

1.2 Goal

The main goal of this project is to develop a tool that supports the use of metrics to monitor the quality of evolving software architectures. Part of the project consists of exploring new ways to present data the user is likely to consider of interest in a helpful manner. The resulting tool should encourage the use of metrics by an attractive and user-friendly interface, allowing the user to find and interpret the data he needs without getting lost.

1.3 Outline

The rest of this document is structured as follows. The next chapter 2 discusses the current state of the art with respect to architectural evolution and metrics. This includes related work and existing tools in this area. Chapter 3 delves deeper into the domain of Architectural Evolution and discusses current challenges. After that, in chapter 4 a strategy is developed to tackle some of these challenges. Applying this strategy resulted in a tool called "MetricView Evolution". Chapter 5 contains the requirements that have been defined for the tool. In chapter 6 a design is presented aimed at fulfilling these requirements. Some notes about the implementation of this design can be found in chapter 7. MetricView Evolution has been applied in academic and industrial context and tested in a small usability experiment. The results of this are discussed in chapter 8. The last chapter consists of an evaluation of the project and the conclusions.
Chapter 2

State of the Art

This chapter consists of 2 sections, in which related work is described. The first section is about related research, the second section about related tooling.

2.1 Existing Work

In this section related research is described. This is split up into three topics: evolution of software architectures, metrics and quality models.

2.1.1 Software Architecture Evolution

The field of studying software evolution is traditionally said to have started with Meir Lehman’s study of the evolutionary characteristics of an IBM mainframe operating system \[\text{Leh69}\]. From this point onwards quite some research has been going on in this field, ranging from measurements and metrics (\[\text{BM99}, \text{TGK99}\]) that have been used to establish laws (\[\text{LRW}^+\text{97}\] and models (\[\text{Per94}\]), to analysis of changes in the history data contained in software repositories (\[\text{GJKT97}, \text{GT00}\]). Most methods are in some way assisted by visualizations (\[\text{GJR99}, \text{HP96}, \text{KAG99}\]) and some include ways to make predictions (\[\text{GKMS00}, \text{RL00}\]).

In \[\text{TG02}\] an approach is described for studying architectural evolution. The authors recognize three main challenges in this field of research: coping with the large amounts of data, integrating the existing techniques into one platform and the analysis of structural changes in software systems.

An interesting approach to software evolution is taken in \[\text{G05}\]. In his thesis, the author describes a way to explicitly model the history of an artefact. Once this model is established it opens up the possibility to define metrics over a history.

2.1.2 Metrics

Over the years several metrics suites have been proposed. At the code level, some well-known metrics are Lines of Code, McCabe’s cyclomatic complexity number \[\text{McC76}\] and function points \[\text{Alb79}\]. For Object-Oriented systems contributions have been made
in [CK91] and [CK94]. In these paper several metrics, among which the well-known ‘Coupling between Objects’ and ‘Depth of Inheritance Tree’, are introduced. One of the classic books on software metrics is “Software Metrics: A rigorous and practical approach” [Fen96]. It contains a thorough dissertation on the background of metrics, an explanation why they are important and a description of how metrics should be used in practice. All this knowledge is applied later on in the book to the area of software engineering itself and the processes that are used in this area.

An interesting application of metrics can be found in [GVG04]. In this paper a case study is presented in which the evolution of a financial web-service system is monitored using several OO-metrics.

2.1.3 Quality Models

The most well-known quality models are described in [JM77], [BBL76], [BB78]. Quality Models are described in more detail in section 3.2.3.

2.2 Existing Tools

In this section some related existing tools are discussed.

2.2.1 SDMetrics

SDMetrics [Wüs05] is described as “The OO design measurement tool for the UML”. It can analyze a UML design using metrics and design rules. These rules and metrics can be custom defined. The results are shown in tables, (cumulative) histograms or Kiviat diagrams. It accepts UML models described in XMI, and is fast in analysing these. The latest version supports version UML 2.0 and is capable of directly comparing two versions of a model. Figure 2.1 shows a screenshot of SDMetrics in action.

2.2.2 MAISA

MAISA stands for “Metrics for Analysis and Improvement of Software Architectures”. This project ran from September 1999 until December 2001 at the Computer Science department of the University of Helsinki, Finland. The MAISA tool supports extraction of metrics and design patterns from multiple versions of a design model. Several statistics can be computed over these metrics and design patterns and it is even possible to make predictions. Unfortunately no example models were available for download and supporting documents were largely in Finnish making it very difficult to evaluate the tool besides from the user manual and some papers in which it is referred to [LN00], [VGNP00].

2.2.3 SAAT

SAAT or “Software Architecture Analysis Tool” is developed at the Eindhoven University of Technology. Initially by Johan Muskens [Mus02] and later extended by Christian
2.2. EXISTING TOOLS

Lange [Lan03] the Software Architecture Analysis Tool, or SAAT, is a metrics calculation tool that analyzes UML models described in XMI. It has been used in several case studies [Mus02], [CL04].

SAAT is a command line tool consisting of a set of scripts. These scripts load an UML model described in XMI into a database, analyze this model using a set of metrics and output the results of this analysis to HTML. A browser can then be used to access the results. The benefits of SAAT are that it is easy to add new metrics or extend the tool itself with for instance a new parser. The main downside is the installation procedure which involves many steps and is rather tricky. Also the visualization options are rather limited.

2.2.4 MetricView

MetricView is a tool created by Maurice Termeer aimed at the visualization of the (large amounts of) data generated by SAAT [Ter05]. It uses a Geographic Information System (GIS) approach, combining the visualization of the architecture and the visualization of the metric data [LC05]. Metric calculation does not take place inside MetricView. Instead the output from SAAT is used as a metric source.

Figure 2.2 shows a screenshot of MetricView.
2.2.5 GASE

The GASE (Graphical Analyser for Software Evolution) tool [HP96] is aimed at the visualization of software evolution. It shows changes that are made to different modules using colors. These colors indicate how recent the changes are.

2.2.6 Beagle

In [TG02] a tool is described for studying architectural evolution. Beagle, as it is called, integrates the use of metrics, visualization and so called origin analysis into an extensible environment.
Chapter 3

Architectural Evolution

In this chapter the main topics that are important in relation to the evolution of software architecture are discussed. The first section is about the software architecture itself. The following section discusses the notion of quality and how to define it. The final section talks specifically about evolution.

3.1 Software Architecture

A software architecture is a high level description of a software system explicitly stating the concerns of the stakeholders that are involved. It is a vehicle for communication among those stakeholders. In it design decisions are explicitly stated. These design decisions involve a trade-off that is also documented. The level of abstraction in the architecture enables it to be reused when developing similar systems. (free after [BCK98]).

3.2 Quality

Quality in general is dependent on the observer. Different observers, that each have their own background and conceptions about quality, will perceive the quality of an artefact differently. Quality can be said, just as beauty, to be in the eye of the beholder. In this section the concept of quality in general and its relation to Software Architectures will be discussed. The first topic will be quality itself. After that we will take a look at how Metrics and Quality Models can help in alleviating the problem of quality being relative to the observer.

3.2.1 Quality Attributes

Without going into the philosophical details of quality it seems useful to provide a basic explanation of the concept of quality and its relation to the subject of software architectures. Quality in the broad sense is sometimes defined in terms of meeting requirements or expectations. In the context of Software Architectures this is usually not what is meant with the term quality. Instead of trying to come up with a better
CHAPTER 3. ARCHITECTURAL EVOLUTION

definition at once, let's take a look at the different aspects of quality in relation to software in general. From this the relation with software architectures will become clear.

The starting point for this short trip into quality will be the definition of the ISO9000 standard: “(Quality is the) degree to which a set of inherent characteristics fulfills requirements.” There are (at least) two important concepts used in this definition, inherent characteristics and requirements.

Inherent Characteristics are those properties of an artefact of which the origin lies in the nature of the artefact itself. This in contrast with Assigned Characteristics of which the origin lies somewhere else.

The concept of requirements is clear in relation with software. These requirements are usually stated in a document that acts as a kind of contract between the client and the developers. Some requirements that are important for the quality (as perceived by the client) might not be explicitly stated. We will not delve into this, but this could cause problems when applying models to create a definition of what quality is for a certain software project or system. In practice these models will only capture a part of the perceived quality. This does not mean that it is not useful to apply these models, just that we keep in mind the limitations. The requirements that are explicitly stated are split into two categories most of the time: functional and non-functional (sometimes called extra-functional) requirements. Both are important, the focus in the rest of this document however, lies on the second category. The reason for this is that evaluation of ‘functional’ quality is in a much further state of research and already a lot of techniques are available to help in the process of verifying this part of quality.

The term quality will from this point be used to denote the non-functional aspect of quality. The functional aspect of quality will be denoted by the term functionality. Functionality is orthogonal to quality. It is almost always possible to change the quality of a software artefact without changing its functionality. This does not mean, however, that quality can be changed at will to the extremes.

Another thing that is important with respect to quality is that it is decomposable into quality attributes. Examples of attributes that are often used are Reliability, Availability, Maintainability and Performance. The inherent difficulty of improving quality is that these attributes are not independent. A good example is the tension that exist between Performance and Maintainability. Maintainability will require a strict decomposition of the system with as few dependencies as possible. The overhead that this causes will hurt the Performance however. The tension between these attributes and the related tradeoffs that need to be made lie at the core of the challenge of creating good software architectures.

3.2.2 Metrics

Metrics are quantifications of properties or attributes and can be used as an instrument to make quality explicit. To see how this works let’s take a look at an example. Say, you wanted to buy an mp3-player and one of the requirements that you have is that it must be able to play music during the 8 hour trip that you are going to make. An
A quality model in general is a model that provides a structure to think about quality. It presents identified factors that influence quality and their relation. Such a model can be a great help to improve quality in a controlled manner.

As stated in the last subsection, the relation between metrics and quality is not always clear in the context of Software Architectures. So called decompositional Quality Models make the relation explicit. They offer a decomposition of quality into attributes, (possibly several levels of) sub-attributes down to the level of metrics. This makes it possible to trace the contribution a metric has to the overall quality of a system. The most important benefit that quality models offer is abstraction. Abstraction enables people to understand and manage complex artifacts.

Several fixed, or predefined, quality models have been created in the past, some of the most well known are:

3.2.3 Quality Models

A question that might arise when considering the use of metrics for software architectures is “Why not use code metrics?” The answer to this question is that the earlier in the project metrics are used, the earlier they can help to indicate problems. It is well known that it is (much) cheaper to fix problems in the earlier phases of a project then it is to fix them in the later phases.
• McCall. Described in [JM77], uses a decompositional approach to quality. It is structured around four levels. At one end there are low-level, directly measurable, metrics that are left undefined. Above these metrics are what are called ‘Criteria’ like Traceability, Completeness and Consistency. These ‘Criteria’ form the basis for ‘Factors’ like Reliability and Maintainability. At the other end three ‘Uses’ are identified: Product Operation, Product Revision and Product Transition.

• Boehm. First appeared in [BBL76] and [BB78]. Also uses a decompositional approach, in this quality model 4 levels are discerned as well. Again, low-level metrics. The second and third level are called ‘Primitive Constructs’ and ‘Intermediate Constructs’ respectively. The top level is called ‘Primary Uses’ and is composed of: ‘General Utility’, ‘As is Utility’ and ‘Maintainability’.

• ISO9126. The McCall quality model has been used as a basis for a ISO standardized model that goes by the name “Software Product Evaluation: Quality Characteristics and Guidelines for their Use”.

• Lindland. A rather different approach is taken in [KLS95]. Instead of decomposing quality into several attributes. The quality of a model is evaluated in three different ways: syntax, semantics and pragmatics. This evaluation consists of comparing four sets of statements: model, language, domain and audience interpretation. Semantic quality is defined on the relation between domain and model, syntactic quality on the relation between model and language, pragmatic quality is defined on the relation between model and audience interpretation. An extension to this model also takes the knowledge of a participant into account that influences the audience interpretation by a relation on which perceived semantic quality is defined. It also adds the notion of social quality that is based on the agreement on the audience interpretation of a model.

It is no coincidence that more than one quality model exists. It is often not desirable to use a ‘one-size-fits-all’ quality model that can be generically used. Software development projects and the systems that are developed differ to much for this. Usually a custom quality model is created for a project or a standard model is adapted to cater for the specific needs of that project.

In the context of the EmpAnADA project attempts have been made to come up with a quality model for the UML that can easily be adapted to suit most projects. At the moment of writing, however, this is still ongoing work.

### 3.3 Evolution

A software architecture is not a static entity. It changes throughout its lifecycle, not only does it change during development, but also afterwards during the maintenance phase of a software project. These changes in the architecture, that will be made with improvement in mind, together are called its evolution. If the quality of a software architecture is evaluated this is currently being done by reviewing a single version. In
this way an architecture is regarded as a static entity and much information is not taken into account, namely the trends in its evolution. We expect that by also looking at evolution data one could get a better understanding of a software architecture and obtain greater control over its development and final quality.
Chapter 4

Towards a Solution

As the name suggests, MetricView Evolution is a successor to MetricView. The core functionality of the original MetricView application, visualizing diagrams combined with metrics, would also be needed in this new version. The author of the original version has paid good attention during design and implementation to enable reuse. This has made it possible to build upon large parts of the original code without too much trouble. Because of the exploratory nature of the project the approach that was taken to develop MetricView Evolution consists largely of prototyping with in between phases of design refactoring. As stated above the basis for these prototypes was formed by the implementation of the original version of MetricView. The ideas that are prototyped have their own background and motivation, which are described in the first section of this chapter. The second section describes the road that was taken during prototyping to come to a tool that helps tackling the challenge of controlling the quality of evolving software architectures.

4.1 Prototyped Ideas

4.1.1 Force-Directed Layout

The first idea that was prototyped was a layout adjustment algorithm for MetricView. The purpose of this was to get familiar with the MetricView design and implementation in a useful manner. As the name suggests, layout adjustment is adjusting a layout. The idea behind this is that sometimes a layout is available but due to a change in a diagram (e.g. adding or removing something) the layout is no longer sufficient. It would be possible to generate a completely new layout. When the original diagram is created by a person, however, it might be useful to preserve some of the structure of the diagram. The designer has a mental map of the diagram and by preserving the structure the adjusted diagram is easier to recognize than the same diagram with a completely new layout. The layout adjustment is in this cased performed by a force-directed algorithm. The idea for this type of algorithm is already quite old and originates from a technique for component placement in circuit boards [QB79]. The algorithm used in our prototype
(described in [FR91]) simulates a mechanical system with springs and rings. It uses Hooke’s law, which is an approximation of the behavior of springs, to apply forces on the diagrams. These forces will cause the distances between the diagrams to converge to a predefined optimal distance that prevents the from overlapping. [MELS95]

4.1.2 Correspondence Visualization

During the first half of this project, another masters project was running with the subject of correspondence between models [vO05a]. Initially the idea was to test the Force-Directed layout to generate a unified layout for several versions of a diagram. The diagrams are stacked on top of each other using the third dimension for the different versions. The layout algorithm makes sure the diagram elements that represent a different version of the same model element have the same 2D position, they are only on a different layer. Additions or removals can easily be spotted this way, also because they are given a different color. The problem with this visualization is that it does not scale well to many versions or large diagrams. Of course some filtering can be applied to only visualize the versions that one is interested in. An example correspondence visualization is visible in figure 4.1.

4.1.3 Metrics over Time

The correspondence visualization combined with the idea of visualizing metrics on top of diagram elements leads to the idea of displaying the evolution of a metric over time on top of a diagram element. In a prototype this was tried out by drawing a graph on top of each stack of diagram elements. One variation used an ordinary (flat) way to draw the graph, a second variation draws the graph in 3 dimensions connecting each point in the graph to the layer related to the version that the point represents. Basic hallway usability testing showed that the second variation did not add that much value and confuses the user.

Figure 4.2 shows the prototype implementation on the left and the final implementation on the right. The final implementation shows the metric graph only when the user hovers with the mouse pointer over a diagram element. We assume that the mouse pointer is at the users locus of attention. Drawing all graphs at once would obfuscate the diagram elements beneath the graphs making them harder to read. On the other hand, showing the metric graph only for one entity disallows seeing relations between the metric values for different entities.

4.1.4 Quality Tree

The metric information visualized in the original MetricView shows a low-level view on the quality of the system. In an effort to abstract from this a prototype was created that includes a quality model. This quality-model aggregates the metric information into quality attributes that can themselves be aggregated into higher-level quality attributes. Aggregation is done using functions that range from basic counting or taking average to
4.1. PROTOTYPED IDEAS

Figure 4.1: Example of correspondence visualization
Figure 4.2: Metric Visualization for multiple versions: (prototype on the left, final implementation on the right)
4.1. PROTOTYPED IDEAS

Figure 4.3: Example Quality Tree

custom tailored ones. So a quality attribute can be thought of as the result of a function of other quality attributes or metrics. The advantage of having an explicit quality-model in a tool is not only automating evaluation of quality attributes but it also supports for tracing dependencies between high-level attributes and low-level metrics. It can show which metrics contribute e.g. to ‘maintainability’ or which attributes depend on the metric ‘coupling between objects’. The idea of the Quality Tree is to enable users to define their own quality model that is best suited to their project(s). This model can then be used to quickly evaluate the main quality attributes that are deemed important. After identification of attributes with suspicious values the dependency tracing can be used to identify the metrics that are responsible for this. This works by highlighting all upstream and downstream dependencies for a given node.

Figure 4.3 shows an example quality tree consisting of 12 quality attributes (red) and 12 metrics (green). The values of the quality attributes and metrics are in this case visualized using 3d bars. The functions that aggregate the data from one or more metrics or quality attributes to another quality attribute are depicted by black circles.
4.1.5 Timeline

Something that is lacking in the Quality Tree is the ability to spot trends in the change of metric and quality attribute values. To solve this the Timeline was introduced. The Timeline consists of a small calender on which the dates are marked for which a snapshot of the model is loaded into the tool. The time is denoted by the x-axis, the y-axis is used when a metric or quality attribute is selected. For this selected metric or attribute a graph is shown, depicting the change over time.

In figure 4.4 Timeline shows the trend in the metric ‘Coupling between Objects’ for three versions of a model. The horizontal axis represents time, the vertical axis the value of the metric or quality attribute. The color scale indicates that in the second version the value of metric approached the upper threshold.

4.1.6 Lange Diagram

A Lange diagram is best described as a meta diagram. It consists of the different views on the software model and their relations. The views are grouped in different layers on abstraction level and range from requirements at the top to implementation class diagrams at the bottom. The layers that are implemented in MetricView Evolution are the use cases, scenarios (or message sequence diagrams), design classes and state diagrams.

The main purpose of the Lange diagram in MetricView Evolution is navigation. It is a replacement of the model explorer in the original MetricView (the treelist widget on the left side of the window). It should provide the user that needs some specific information from the model quick and easy access to that information. Additionally, users that are not looking for specific information but merely try to understand the model should be presented with a overview of elements that are present in the model and the relations between these elements.

The relations that are currently implemented in Metricview Evolution are:
4.1. PROTOTYPED IDEAS

- **Use Case - Scenario.** A Scenario, depicted in a message sequence diagram describes the interactions in the system that take place to execute a Use Case. In UML the link between these two consists of the owner of a message sequence diagram being a Use Case.

- **Scenario - Classes.** The interactions depicted in a message sequence diagram that describes a scenario take place between instances of classes. In UML various links between Scenarios and classes can be found. The one that is currently being used is the type of an object (ClassifierRole) in a message sequence diagram being a class. Another possibility would be to use the relation between the messages in the message sequence diagram and the methods in a class.

- **Classes - State Diagrams.** The internal behavior of a class can be modelled using a state diagram. In UML the link between classes and state diagrams consists of the owner of the state diagram being a class.

The default layout of the diagrams in the Lange-Diagram is similar to the layout of text that is centered in a text editor. Each type of diagram is put on a new 'line'. When a line gets too long and it doesn’t fit in the window any more it is wrapped.

Figure 4.5 shows an example of a Lange Diagram with inter-diagram relations. The ellipses in the upper left part of the figure represent use case diagrams. The use cases in these diagrams are connected to the sequence diagrams that describe these use cases. In these sequence diagrams there are objects of a specific type. If this type matches a classifier in the class diagram a relation is drawn. The fourth type of diagram, the state diagram, that can be used to describe the behaviour of a class, is not connected in this case. This is due to the fact that the creator of this model has not linked the diagram to the class.

4.1.7 Context Diagram

After the identification of metrics with a suspicious value another step is needed to determine the cause. If the model is large and there are many diagrams it can be a lot of work to find the information that contributes to the metric. To help in this process the notion of a context diagram has been developed. A context diagram contains only those parts of the model that contribute to the value of a specific metric related to a model element, so it acts like a kind of filter. A diagram containing these parts of the model is automatically generated. This diagram should help to shorten the time the user needs to determine the cause of a specific problem indicated by a metric.

Figure 4.6 shows the added value a context diagram can offer. In this case the metric ‘Number of Children’ was selected. The (normal) diagram on the right shows only part of the children making it look like there is nothing out of the ordinary here. The context diagram on the left however reveals that the class on which the focus lies (S.467) has no less than 28 children, which is quite a lot. It would be a tedious job to trace in which diagrams this class occurs and finding all the child classes by hand. A context diagram makes this kind of analysis much easier. The children found in the normal diagram are
highlighted in both diagrams, this is done by hand to show where they are in the context diagram.

4.1.8 Search and Highlight

Finding a specific element in a large model with a huge number of diagrams can be tedious job when done manually. To help with this, search functionality is included in a MetricView Evolution prototype. Searching is done by typing in a keyword in a text box. The tool then highlights all diagram elements that are somehow related to this keyword. The user can then quickly zoom in on the elements or use the 'previous' and 'next' functionality to automatically navigate through all matches.

The Search and Highlight function has led to some unintended usage. When highlighting all elements that are related to a specific keyword the tool also implicitly shows the relations between these elements themselves. An example of this is found in the TrafficLight Control System case (see 8.1) where searching for 'TrafficLight' reveals a class, several instances of this class in sequence diagrams, some use cases that are described by these sequence diagrams and finally a state diagram describing the internal behavior of the class.
4.2. THE NOTION OF TASKS

4.1.9 Clustering

The default layouting of the Lange-Diagram does not add any extra value other than allowing the users to spot easily the different types of diagrams that are available. Based on the idea of explicitly showing the relations between diagrams and making use of the force-directed layout algorithm that was already present a new layouting approach is created for the Lange-Diagram. This approach uses the relations between diagrams as attracting forces in the force-directed layout algorithm. This results in clusters of diagrams that are in some way related. If a system description is complete it should only display one cluster that is connected. By drawing a convex hull around each of the clusters the user can easily see the different clusters that are present.

Figure 4.7 shows an example of a Lange Diagram with clustering enabled. One diagram is selected and the transitive closure of the fan-in and fan-out of the relations originating in that diagram are visualized as arcs.

4.2 The notion of tasks

The functionality of most of these ideas is nice in itself but for a useful tool it is better to structurally support tasks that are predefined. After brainstorming sessions and interviewing potential users the following task categories where it could be useful to
Figure 4.7: Lange Diagram showing clustering.
apply the tool were identified. For each of these phases a set of tasks related to it is described below.

- Program Understanding
- Changing a Model
- Evaluation

4.2.1 Program Understanding

These set of tasks are mainly performed by users who are unfamiliar with the system that is described by a model. Before embarking on tasks that require knowledge of the system they first desire to gain this knowledge. The following tasks that contribute to this can be supported by the tool.

- Browsing
- Identifying key classes
- Identifying complicated interactions between classes

Browsing is actively supported by the Lange Diagram. It helps users to create a mental map of the system by showing an overview of all diagrams and their relations. This allows a user to remember the position of a diagram instead of only a name and a type. Together with the search and highlight functionality this should reduce the time it takes to find a certain element.

When selecting a class in the Lange Diagram, it will be highlighted in all diagrams that it occurs in. If a class occurs in many diagrams, there is a high probability that the class is important. Key classes can also be identified using certain metrics that indicate their importance. An example could be the number of classes that depend on a certain class. If this number is high, it could be an indication that the class is important. This task can be partially automated by using outlier detection and visualization. There are two mechanisms that support this, filtering based on outliers and highlighting elements for which a metric is beyond a pre-defined threshold. The filtering mechanism puts the emphasis on outliers by making all the other elements almost completely transparent. If a threshold has been defined for a metric, elements for which the metric value is beyond the threshold can be marked (e.g. with red) making identification from the Lange Diagram an easy task.

4.2.2 Changing a Model

There are several reasons why it might be necessary to change a model after its creation, some of which are:

- Extension
• Bug Fixing
• Change Request

Two important questions related to such changes are what the impact of the change will be on the system and how the change affects the quality of the model. The ideas implemented in the tool can help in the process of answering these questions.

The inter-diagram relations that can be visualized in the Lange Diagram together with the context view form a good basis for impact prediction. Impact prediction is useful when changing a model because it provides an estimation of the effort that is associated with the change. Let’s say that a change needs to be made to class A. By selecting the class in the Lange Diagram it can be seen that class A occurs in a large number of class diagrams, is instantiated in several message sequence diagrams and has a complicated state diagram describing its internal behavior. This already gives some indication that changing this class will be costly in terms of effort. If, when applying a coupling metric and looking at the Context View, it shows that the class is coupled to a large number of other classes the confidence that the impact for changing this class is high, increases.

4.2.3 Evaluation

From a quality assurance perspective it is necessary to evaluate the model to get information about its current state and to be able to make predictions about its future state and development time. The three evaluation activities that are supported by ideas implemented in the tool are:

• Quality: Snapshot
• Quality: Trend over time
• Maturity

Evaluating the quality of a single version of a model is supported by the Quality Tree, the Context View and the metrics visualization. The Quality Tree enables users to apply their notion of quality to a UML model and trace the cause of certain quality attribute values back down to the level of metrics. Using the Context View this tracing can even continue to the level of the model elements that contribute to the value of the metric. In other words the quality of a single version of a model can be evaluated at different abstraction levels.

When data is available for multiple versions of a model the Timeline can be used to evaluate the quality at the higher abstraction levels. It can show trends in the average values of quality attributes and metrics. When a more low-level view is needed, the metric visualization that maps metrics onto the model visualization allows trends to be shown for individual metrics applied to individual model elements.

The third activity, evaluation of the maturity of a model, is supported by the Lange Diagram combined with metrics or design smells visualization. As stated before the
4.2. THE NOTION OF TASKS

Lange Diagram offers an overview of all diagrams and relations between these diagrams. This helps to assess how much diagrams of different types are present. Combined with metrics about the size of the model this can give an indication of the level of completeness.
Chapter 5

MetricView Evolution - Requirements

5.1 Introduction

5.1.1 Purpose

This chapter specifically states what the tool is supposed to do according to the client. It will be used as a guideline and reference for verification in later stages.

5.1.2 Scope

The tool to be made, named MetricView Evolution, will be based on an existing tool. As the name suggests, this base is MetricView. During this project not only specific functionality will be added to MetricView but also the extensibility will be increased to make future updates easier.

5.1.3 Overview

The rest of this chapter is structured as follows. The next section covers the environment in which the tool will have to function, the types of users and important general requirements and constraints. Section 5.3 contains specific requirements and constraints as well as the corresponding priorities of those requirements. To make explicit how the tool is going to be used, a number of use cases are defined in section 5.4. Finally, section 5.5 contains (references to) descriptions of external interfaces and internal models.

The requirements stated in this document were defined in collaboration with Michel Chaudron and Christian Lange who fulfilled the role of clients during this project.

5.2 General description

This section describes the environment of the tool and the influence of the environment on the tool. It does not state specific requirements, but explains and motivates the
requirements that are specified in the next section.

5.2.1 Product perspective

A software architecture is created during the software development process. It represents the architect’s vision of the solution to the problem stated in the requirements and is used to communicate this vision and its consequences for the final product to stakeholders. An architecture usually goes through several development iterations during which changes are made to improve the quality of the solution. These changes are a tradeoff between different quality attributes. Some of these quality attributes can be quantified using metrics. This makes it possible to give a quantification of the tradeoff.

An architecture changes during its development. By applying metrics on several stages of the evolution of an architecture the changes in the quality tradeoff can be measured. This measuring can be automated if a version control system is used to store the states of the architecture during development.

The last step of this evolution is the implementation. The implementation often deviates from the architecture for several reasons. By reverse engineering an implementation and including this reverse engineered model into the evolution, it is possible to trace the change of quality attributes into the implementation.

5.2.2 Mission statement

The purpose of this tool will be to visualize the quality attributes of the evolution of an architecture and corresponding implementation. This visualization will help the architect and other stakeholders by gaining insight in the quality level of the artifacts changing over time.

5.2.3 General capabilities

As stated in 5.2.2 the main task of the tool is to visualize the quality attributes of the evolution of an architecture. Because the importance of these attributes can vary between different projects the tool should provide a method of selecting the quality attributes that are deemed important by the architect. This basically comes down to creating a quality model. A quality model can be seen as a model that depicts the composite characteristics of quality and their relationships [Fen96]. The visualizations should adapt to this quality model to display the important aspects.

It must be possible to export images of these visualizations for use within external documents. Furthermore, it should be possible to generate reports based on the quality model. These reports can be used to communicate with other stakeholders. It should also be possible to extract metrics from a model within the tool itself, so that if no external tool for this is available, the tool will still be useful.

To visualize the final step in the evolution of the architecture, namely the implementation, the tool must be capable of importing UML models generated by a reverse engineering tool (e.g. COLUMBUS). These models will typically lack any information
5.2. GENERAL DESCRIPTION

about diagrams or their layout. The tool must be able to reconstruct this information based on the correspondence between the designed model and the implementation model. The layout must be generated in such a way that the architect who created the design can recognize his creation in the reverse engineered model.

5.2.4 General constraints

The language used in all the deliverables of this project including the tool itself will be English.

The tool will be developed on the Windows platform but it should be possible to port it to Linux and Mac OS without spending too much effort (e.g. only creating a new GUI component).

5.2.5 User characteristics

Architect

The main user of the tool is the software architect who created the architecture and wants to evaluate the evolution of his creation. This user has a mental map of his creation that must be preserved in the tool to ease navigation.

Client

A secondary type of user is the client who wants to verify statements about the quality and/or progress of the architectural development. This user has less insight in the ins and outs of the complete architecture. An aspect of the architecture that is important to the client is the set of use cases, which are often created in collaboration with the client.

5.2.6 Environment description

A global overview of the environment in which MetricView Evolution will function can be seen in figure 5.1.

XMI

Most of the tools in which UML modelling is done support the XML Metadata Interchange format. Therefore it is important that this tool offers the functionality to import and export UML models that are stored in this XMI format.

Related to this is the layout information inside the XMI. The tool must be able to import this information and add generated layout information to an XMI file.

With respect to this XMI format, the testing reference should be the XMI generated by the Unisys XMI plug-in for Rational Rose. Because Rational Rose is a widely used CASE tool this makes sense.
Figure 5.1: Tool environment
5.3. SPECIFIC REQUIREMENTS

Metrics

The tool must offer functionality to import metrics as outputted by the SAAT tool and relate these metrics to a model. This should be implemented in such a way that it is possible to add functionality to import information from other analysis tools.

Version control

The tool must offer functionality to retrieve the versions of a UML model from a version control system like Subversion or CVS and use the version and time-stamp information for the creation of the evolution visualization.

5.3 Specific Requirements

In this section all requirements and constraints of the tool that is to be developed are specifically stated. The tool will adhere to these requirements. Each of the requirements has a unique identifier (UID) so it can be traced throughout the project. Each of the requirements also has a priority varying from 1 (highest) to 4 (lowest). The meaning of each priority level is described in table 5.1.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The tool must fulfill this requirement in order to be of any use.</td>
</tr>
<tr>
<td>2</td>
<td>The tool should fulfill this requirement.</td>
</tr>
<tr>
<td>3</td>
<td>The tool could fulfill this requirement.</td>
</tr>
<tr>
<td>4</td>
<td>The requirement is nice to have. Requirements with this priority are very likely not to be met within this project.</td>
</tr>
</tbody>
</table>

Table 5.1: Priority Levels

5.3.1 Capability requirements

Input

The specific input requirements are stated in table 5.2 and 5.3.

Output

The specific output requirements are stated in table 5.4.

Functionality

The specific functionality requirements are stated in table 5.5, 5.6 and 5.7.
### Chapter 5. Metricview Evolution - Requirements

<table>
<thead>
<tr>
<th>UID</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN001.a</td>
<td>The tool must have the ability to import UML models in the XMI format created by the Unisys XMI plug-in version 1.3.6 for Rational Rose 2003.</td>
<td>1</td>
</tr>
<tr>
<td>IN001.b</td>
<td>The tool must be able to import UML models according to the XMI 1.2 - UML 1.4 format.</td>
<td>1</td>
</tr>
<tr>
<td>IN001.c</td>
<td>It would be nice if the tool had the ability import UML models according to the XMI 2.0 - UML 2.0 format.</td>
<td>4</td>
</tr>
<tr>
<td>IN002.a</td>
<td>The tool must have the ability to import source code facts in the form of UML models in the XMI format generated by the reverse engineering tool COLUMBUS/CAN 3.5</td>
<td>4</td>
</tr>
<tr>
<td>IN002.b</td>
<td>The tool should have the ability to import source code facts in the form of reverse engineered UML models according to the XMI 1.2 - UML 1.4 format.</td>
<td>4</td>
</tr>
<tr>
<td>IN002.c</td>
<td>It would be nice if the tool had the ability import source code facts in the form of reverse engineered UML models according to the XMI 2.0 - UML 2.0 format.</td>
<td>4</td>
</tr>
<tr>
<td>IN003</td>
<td>The tool could have the ability to import layout information for artifacts from other sources in the form of XMI 1.2.</td>
<td>3</td>
</tr>
<tr>
<td>IN004.a</td>
<td>The tool must have the ability to import multiple versions of a UML model and their revision history from a Version Control System</td>
<td>1</td>
</tr>
<tr>
<td>IN004.c</td>
<td>The tool should offer the functionality to add support for other Version Control Systems as a plug-in.</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.2: Artefact data

#### 5.3.2 Constraint requirements

The specific constraint requirements are stated in table 5.8 and 5.9.
5.4 USE CASES

5.4.1 Input

An overview of all input use cases is depicted in figure 5.2.

Import from XMI

The user opens a file opening dialog, navigates to a valid XMI file (see 5.3.1) and opens this file. The tool imports the information from the XMI file, both model information and layout.

Import metrics

The user has generated metrics for a specific model with SAAT and already has this model opened in the tool.

The user opens a file opening dialog, navigates to the output file from SAAT and opens this file. The tool imports the metrics information from the output file and relates the metrics to their model elements.

Import source code

The user has reverse engineered a UML model with COLUMBUS and stored this model in a XMI file.

The user opens a file opening dialog, navigates to the output file from COLUMBUS and opens this file. The tool imports the model information from the XMI.

Import from version control system

The user opens a browser dialog for the version control system and indicates from which XMI file the revision history is to be imported. The tool extracts the revision history.

<table>
<thead>
<tr>
<th>UID</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN005.a</td>
<td>The tool can import product attributes generated by the SAAT tool.</td>
<td>4</td>
</tr>
<tr>
<td>IN005.b</td>
<td>The tool can import product attributes generated by other analysis tools (e.g. SDMetrics).</td>
<td>4</td>
</tr>
<tr>
<td>IN006</td>
<td>The tool can calculate metrics given an artefact.</td>
<td>1</td>
</tr>
<tr>
<td>IN007</td>
<td>The tool can import quality metrics of several types using a data format that is to be specified and can include artefact evolution data, resource data, testing data. Information about mapping the analysis data to the artefact must be available.</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.3: Analysis data
CHAPTER 5. METRICVIEW EVOLUTION - REQUIREMENTS

<table>
<thead>
<tr>
<th>UID</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT001</td>
<td>New visualization types can simply be added to the tool using an extension mechanism.</td>
<td>3</td>
</tr>
<tr>
<td>OUT002</td>
<td>Visualized images can be exported in a graphics format that uses lossless compression or no compression at all.</td>
<td>3</td>
</tr>
<tr>
<td>OUT003</td>
<td>Product attributes that are created by the tool can be exported into a human- and machine readable format (e.g. CSV or XML).</td>
<td>1</td>
</tr>
<tr>
<td>OUT004.a</td>
<td>UML models with a changed layout can be exported into XMI</td>
<td>3</td>
</tr>
<tr>
<td>OUT004.b</td>
<td>The tool must be able to export UML models according to the XMI 1.2 - UML 1.4 format.</td>
<td>3</td>
</tr>
<tr>
<td>OUT004.c</td>
<td>It would be nice if the tool was able to export UML models according to the XMI 2.0 - UML 2.0 format.</td>
<td>4</td>
</tr>
<tr>
<td>OUT005.a</td>
<td>The tool must be able to generate a report based on a quality model that is applied to the evolution of an architecture</td>
<td>1</td>
</tr>
<tr>
<td>OUT005.b</td>
<td>The generated report must be in a human readable format that is easily transformed into HTML and LaTeX (e.g. XML).</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.4: Output requirements

from the version control system, loads all versions of the model.

**Import external analysis data**

The user has generated external analysis data that conforms to the defined format (see 5.5.1), and is related to the model that is opened in the tool.

The user opens a file opening dialog, navigates to the file containing the external analysis data and opens this file. The tool imports the analysis data and relates it to the open model.

**5.4.2 Output**

An overview of all output use cases is depicted in figure 5.3.

**Export to XMI**

The user has opened and edited the model in the tool that needs to be exported to XMI.

The user opens a file saving dialog, navigates to the folder the XMI needs to be exported to, chooses a filename and exports the model to XMI. The tool generates a valid XMI file (see 5.3.1) containing both the model and the layout information.
5.4. USE CASES

Export visualization to image

The user has setup the visualization that is to be exported.

The user opens a screen-shot saving dialog, navigates to the folder the visualization image is to be exported to, chooses a filename and exports the visualization. The tool generates a valid image (see 5.3.1) containing the visualization.

Export analysis data

The user has generated analysis data inside the tool for a specific model.

The user opens a file saving dialog, navigates to the folder the data needs to be exported to, chooses a filename and exports the data. The tool generates a file containing the data that conforms to the defined format (see 5.5.1), and is related to the model that is opened in the tool.

Create a report

The user has defined a quality model in the tool based on which he wants to generate a report.

The user opens a file saving dialog, navigates to the folders the reports needs to be generated in and chooses a filename. The tool generates a valid report (see 5.5.1).
containing the evaluation of the model that was opened, based on the defined quality model.

5.4.3 Functionality

An overview of all functionality use cases is depicted in figure 5.4.

Navigate

The user has opened a revision history of a model from a version control system. The user navigates to the oldest version of the model by constraining the visualized versions of the model.

View evolution quality attributes

The user has opened a revision history of a model from a version control system. The user activates the evolution view and selects several metrics. The user navigates to the metrics for a single element, a single diagram and the entire model.

Evaluate architecture

The user has opened a revision history of a model from a version control system and defined a quality model.

The user activates the evolution view and navigates to the evaluation for a single element, a single diagram and the entire system.
5.4. USE CASES

Figure 5.4: Use cases: functionality

**Match design and implementation**

The user has opened a revision history of a model from a version control system and imported a reverse engineered model from the implementation.

The user activates the matching.

**Generate metrics**

The user has opened a model.

The user starts the generation of metrics and afterwards exports them to a file.

**Edit UML model**

The user has opened a model.

The user changes the layout of the UML model by dragging around classifiers.
5.5 Software requirements

5.5.1 External interfaces

XMI document format

A specification of XMI 1.2 can be found in [Gro02], the specification of the 2.0 version can be found in [Gro03].

External analysis data document format

This format is to be defined in a later phase of this project.

Report format

This format is to be defined in a later phase of this project.

SAAT document format

A description of the SAAT output file format can be found in the SAN subversion repository subdirectory ‘/research/projects/empanada/sources/SAAT/docs/’.

Prototype of User Interface

Prototypes of the User Interface will be constructed during the design phases.

Specification Version Control System communication protocol

The three first contenders for the main version control system to be implemented are:

- CVS
- Subversion
- ClearCase

A specification of the subversion protocol does not seem to be available at the moment. A platform independent open source subversion library based on the wxWidgets library, which could be used instead of implementing the protocol during the project, is available however. A specification of the CVS protocol is also not available. A CVS library (libcvs) is available but not in C++. Little is known about ClearCase at the moment. Research will have to be conducted in the design phases to retrieve more information about all three contenders and make an informed choice with respect to which to implement first.
5.5. SOFTWARE REQUIREMENTS

5.5.2 Internal models

MetricView

MetricView currently offers the functionality to import UML models from XMI (version 1.0 and 1.2 for UML 1.4). It can display Class diagrams, Message sequence diagrams, State diagrams and use case diagrams. Metrics can be imported and displayed onto class diagrams. Visualizations are available in 2D and in 3D.

A global overview of the processing that takes place in MetricView is displayed in figure 5.5 (taken from the MetricView manual). The visualization subsystem contains a single canvas that coordinates the visualization of a model, see figure 5.6. To handle more than one diagram the canvas has a tree of windows (created by splitting the canvas horizontally or vertically). Each window has a viewport that contains the size of the window in screenspace (pixels) and a UMLModelVisualizer that performs the actual visualization. This visualizer uses a visitor pattern to access the data in the model as can be seen in figure 5.7. The actual drawing is done by the GraphicsGL class of which there is only one. The reason for this is that OpenGL is a procedural library.

The metrics parser is implemented as a method in a single class. The XMI parser on the other hand is implemented as a separate component. It relies on a XMIParserHandler to do the actual parsing. Currently two parser handler are available: a DOM parser and a SAX parser that is more memory efficient and can therefore handle larger models. Both use the Xerces library for parsing the XMI.

The inner structure of the model manager is displayed in figure 5.8. The UML-ResolveMap is the global collection UMLElements indexed by ID. The map is used throughout MetricView to find model elements and diagram elements based on a given ID. An important thing to notice is that a UMLDiagramElement contains a reference to a single UMLModelElement. The relation between UMLModelElement and UMLDiagramElement is one-to-many. These two facts combined mean that when you need the diagram elements related to a model element you need to do a full search in the resolve-map.

![Figure 5.5: Global processing MetricView](image-url)
CHAPTER 5. METRICVIEW EVOLUTION - REQUIREMENTS

Figure 5.6: Structure of the visualizer subsystem

Figure 5.7: Visualization uses visitor pattern

Mapping design - implementation model

The creation of a mapping between the design model and a reverse engineered model from the implementation will be performed with the help of the methods developed by Dennis van Opzeeland (e.g. see \[vO05b\]).

Mapping stages of UML model evolution

Related work on this topic can be found in \[OWK03\] and \[Roc00\].

Quality meta-model

A model for defining quality models will have to be defined in a later phase of the project.
Figure 5.8: Model manager structure
### Table 5.5: Functionality requirements

<table>
<thead>
<tr>
<th>UID</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUN001</td>
<td>The layout of UML models can be edited within the tool. For specific UML editing requirements see table 5.6.</td>
<td>2</td>
</tr>
<tr>
<td>FUN002.a</td>
<td>Given a (partial) layout and an artefact, the artefact can be layouted according to the existing reference layout. E.g. When a UML model and source code of a system is available, the tool must create a (partial) matching between source code elements and model elements and create a layout for the source code (as a UML diagram) based on the available layout information. (related to IN002, IN003)</td>
<td>2</td>
</tr>
<tr>
<td>FUN002.b</td>
<td>The generated layout should preserve the mental map the creator of the reference layout has.</td>
<td>2</td>
</tr>
<tr>
<td>FUN002.c</td>
<td>The tool should offer the functionality to add new layouters as a plug-in.</td>
<td>2</td>
</tr>
<tr>
<td>FUN003.a</td>
<td>The tool must be able to calculate product attributes (i.e. integration SAAT tool and MetricView). (related to IN001, OUT003).</td>
<td>1</td>
</tr>
<tr>
<td>FUN003.b</td>
<td>The tool should offer the functionality to add new analyzers as a plug-in.</td>
<td>2</td>
</tr>
<tr>
<td>FUN004.a</td>
<td>The tool must handle and visualize multiple (related) artifacts in one visualization.</td>
<td>1</td>
</tr>
<tr>
<td>FUN004.b</td>
<td>The tool must be able to show relations between multiple artifacts using the layout of the artifacts</td>
<td>1</td>
</tr>
<tr>
<td>FUN004.c</td>
<td>The tool must be able to show relations between multiple artifacts using other visual hints like color similarity or links</td>
<td>2</td>
</tr>
<tr>
<td>FUN005</td>
<td>The tool must offer an evolution view. (for specific requirements see table 5.7)</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 5.6: Editing functionality requirements

<table>
<thead>
<tr>
<th>UID</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUN006</td>
<td>The user must be able to change the position of UML classifiers by dragging them</td>
<td>1</td>
</tr>
</tbody>
</table>
### 5.5. SOFTWARE REQUIREMENTS

#### Table 5.7: Evolution view requirements

<table>
<thead>
<tr>
<th>UID</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUN007</td>
<td>The evolution view must visualize data from different versions of the developed system.</td>
<td>1</td>
</tr>
<tr>
<td>FUN008</td>
<td>The evolution view should be available for data with different levels of granularity. At least on system level, diagram level and element level.</td>
<td>1</td>
</tr>
<tr>
<td>FUN009</td>
<td>The evolution view should offer a view on the UML model, where the evolution of the model elements is shown.</td>
<td>2</td>
</tr>
</tbody>
</table>

#### Table 5.8: Platform requirements

<table>
<thead>
<tr>
<th>UID</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLA001a</td>
<td>The tool must run under the Microsoft Windows XP operating system with Service Pack 2 installed.</td>
<td>1</td>
</tr>
<tr>
<td>PLA001b</td>
<td>It would be nice if the tool could run under Linux/X11.</td>
<td>4</td>
</tr>
<tr>
<td>PLA001c</td>
<td>It would be nice if the tool could run under Mac OS X.</td>
<td>4</td>
</tr>
<tr>
<td>PLA002</td>
<td>The tool must run responsively on a computer with:</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>- At least 512Mb main memory,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- 1.8 GHz Intel Pentium 4 performance-wise equivalent processor or better.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- A graphics card with at least 64MB of RAM that supports hardware accelerated Transformation &amp; Lighting and has a stencil buffer. (e.g. A graphics card with DirectX 8.1 level hardware acceleration).</td>
<td></td>
</tr>
<tr>
<td>PLA003</td>
<td>The language of the user-interface must be US English.</td>
<td>1</td>
</tr>
<tr>
<td>UID</td>
<td>Description</td>
<td>Priority</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>QUA001</td>
<td>Navigating the visualizations should be possible in real-time (i.e. display updates must happen within 15 ms).</td>
<td>1</td>
</tr>
<tr>
<td>QUA002</td>
<td>Editing the layout of a UML model should be possible in real-time (i.e. display updates must happen within 15 ms), with the relaxation that certain visualization options can be disabled temporarily by the tool.</td>
<td>2</td>
</tr>
<tr>
<td>QUA003</td>
<td>A progress bar must be shown to inform the user of the progress of an operation if performing this operation takes longer than 1.0 second.</td>
<td>1</td>
</tr>
<tr>
<td>QUA004.a</td>
<td>Generating a layout from a reference layout should take &lt;0.5 seconds for small models (&lt;20 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA004.b</td>
<td>Generating a layout from a reference layout should take &lt;1.0 seconds for medium sized models (&lt;100 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA004.c</td>
<td>Generating a layout from a reference layout should take &lt;5.0 seconds for large models (&lt;500 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA004.d</td>
<td>Generating a layout from a reference layout should take &lt;1 minute for huge models (&lt;2500 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA005.a</td>
<td>Generating a report based on a custom quality model should take &lt;1.0 seconds for small systems (&lt;200 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA005.b</td>
<td>Generating a report based on a custom quality model should take &lt;2.0 seconds for medium sized systems (&lt;1000 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA005.c</td>
<td>Generating a report based on a custom quality model should take &lt;10.0 seconds for large systems (&lt;5000 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA005.d</td>
<td>Generating a report based on a custom quality model should take &lt;2.0 minutes for huge systems (&lt;25000 entities).</td>
<td>2</td>
</tr>
<tr>
<td>QUA006</td>
<td>The user interface should be responsive at all times when the tool is operational.</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.9: Quality requirements
Chapter 6

MetricView Evolution - Design

6.1 System Overview

Figure 6.1 shows the tool in its immediate context. It will be a single user application that takes a UML model, a quality model and metric definitions as input and creates a visualization or report.

6.2 Design decisions and preliminary information

6.2.1 MetricView extension

The design presented in this chapter is not created from scratch but based upon the design of the original MetricView tool. The reason behind this important design decision is twofold. Part of the reason is the development process that consists of iterative prototyping using the code of MetricView as baseline. Another part of the reason is that the original design was created with extensibility and reuse in mind. It was estimated to be more efficient to adapt this design than create a new one. The parts of the design that are highly similar to the original design are:

- The visualization package. Although the overall structure has been expanded to support the Lange-Diagram and multiple versions, a large part of the original classes are still present and interact in the same way.

- The GUI package.

6.2.2 GUI Library

MetricView uses the wxWidgets GUI library for its interaction with the user and the operating system. During a short evaluation this library turned out to provide sufficient functionality. Another GUI library that was tested was Qt (it is used in the QualityTreeEditor see the Appendix B). Although it was in some ways better and easier to use, the effort to convert the existing implementation to this library was found to be
Figure 6.1: A global analysis model of MetricView Evolution and its context
6.2. DESIGN DECISIONS AND PRELIMINARY INFORMATION

too high. An additional benefit of using a GUI library is that it should not be too hard to port the application to another platform like Linux or OS X. Although not critical it might help in expanding the userbase.

6.2.3 XML

XML is used in quite a few places throughout the application. On the input side, of course, there is the UML model defined in XMI and the quality model that is defined in XML as well. On the output side, XML is used to store generated reports on the calculated metrics. To parse the XML two different methods are used. The XMI files are parsed using a set of XSL transformations external to the application. The main reason for this approach is that it is easier to adapt the parser to different flavors of XMI of which there exist many. The original MetricView tool used an internal XML parser that had to be changed often to be able to parse XMI generated by a new tool. The downside of this is that parsing is a little slower. The QualityTree that contains the quality model is parsed using an internal XML parser. Because the XML format is defined by the author of the tool itself changeability is not really an issue. The reports on the metrics are generated by a class in the application itself. The XML that is outputted by this class is later formatted and transformed into HTML by a XSL transformation.

6.2.4 MySQL

The design decision has been made to base the storage inside the tool on a database. This is different from the way MetricView handled storage. In MetricView the XMI parser generated an object model of the UML model that was later used throughout the rest of the application. The benefit of this was that it was fast and easy to use. The downside was that it was relatively hard to calculate metrics over the model in an uniform way. One of the other tools developed in the EmpAnaDA project, SAAT, used an external database to store the model. The metrics that this tool calculates are defined as SQL queries augmented with some Perl to support recursive definitions. The idea for the MetricView Evolution project was to integrate these two approaches, thereby combining the benefits of both. As stated above, the storage in this new tool is based on a database. Instead of using queries throughout the rest of the application to access this database, there is an abstraction layer that generates an object model from the database. Retrieval and manipulation actions are defined on this object model, that translates them to database queries and provides caching to keep the performance at a reasonable level. Instead of using an external database, the decision was made to use an embedded database. The benefit of this approach is mainly easier installation. The downside is that database administration is a little bit harder because the default tooling can not be used to make changes to the database. The database that was chosen is MySQL. This open source database had already proven itself in SAAT and the author already had some experience using it. Other alternatives that were considered are Firebird and SQLite.
6.2.5 Python

Just as with SAAT metrics are defined using SQL. The problem with SQL is that it does not allow recursion. In SAAT this problem was overcome by partly implementing the metrics in Perl, as the rest of the program was already defined in this language. A similar approach could be taken for MetricView Evolution, partly implementing the recursive metrics in C++. The problem with this would lie in the extensibility of this approach. Each new recursive metric that is added would require a recompile. This would be very user unfriendly and defy the main goal of the project of encouraging the use of metrics. A second solution would be to use stored procedures. Stored procedures offer a basic programming language in which procedure can be defined that are stored in the database. When needed these procedures can be executed. This has two benefits, first it increases the expressiveness and second it decreases the number of transactions that are needed to perform the same manipulation on the database because everything is executed on the database side. This feature was new to the latest version (5) of MySQL at the time of writing. Unfortunately, the embedded version of the database does not support it. Attempts to enable it in the source-code failed to give any results. At this point the switch to a different database could have been made, but this turned out to be too costly. The third, and final solution, was to integrate an interpreter for a scripting language into MetricView Evolution. The scripts contain the metric definition and are allowed direct access to the database. Both Perl and Python integration have been tested and Python turned out to have a much nicer embedding interface. Although this makes it possible to define recursive metrics in a quite straightforward way, it does burden maintainers of the tool with another language.

6.3 System Design

6.3.1 Design Method

The design presented in this chapter has been developed in an iterative manner in between prototyping cycles. As always there was a tension between the design and the implementation and the risk of the design quickly becoming outdated.

6.3.2 Decomposition Description

The tool consists of 7 packages each having their own specific functionality: Control, DataStorage, FileIO, GUI, Manipulation, Visualization and Instrumentation. An overview of dependencies between the packages as designed can be found in figure 6.2. Each of these packages is described shortly below, for a more detailed description see section 6.3.

GUI

The GUI package contains the classes that together make up the User Interface of MetricView Evolution or help in the construction of it.
Figure 6.2: Decomposition
CHAPTER 6. METRICVIEW EVOLUTION - DESIGN

Control

The Control package is a collection of classes that facilitate various ways for other components to communicate. They comprise, so to say, the plumbing and wiring of MetricView Evolution. The most important parts of this package are an implementation of the Model-View-Presenter pattern (see the appendix for more details [B.1.3]), an implementation of the Publish-Subscribe pattern and a manager class that implements selection.

DataStorage

The DataStorage package consists of a collection of classes that facilitates the storage of UML models, metrics and a quality model. The storage of UML models and metrics is based on an embedded database from which an object model is created that enables easy data retrieval and navigation throughout the rest of the application.

Manipulation

The Manipulation package is a collection of classes that manipulate data in or extract data from the DataStorage package. At the moment the available manipulation types are diagram layout, quality tree layout, metrics calculation and the extraction of inter-diagram relations.

Visualization

The Visualization package consists of everything needed to visualize the data that is stored by the DataStorage package. It ranges from a main drawing canvas, split into windows that support multiple layers of diagram visualizations to the visualizers that perform the actual rendering using an OpenGL wrapper, providing an easy interface to a lot of standard drawing routines.

FileIO

The FileIO package is responsible for the input and output of the data stored by the DataStorage package. A large part of this, the XMI parser, is implemented using the XSLT library XALAN and collections of xsl scripts that translate the XMI into SQL queries. These queries are used to insert the data in the database that is inside the DataStorage package. Because of the use of XSLT for parsing, most of the logic resides in stylesheets external to the application. The XML (SAX) parser used to load QualityTrees is internal to the application, and uses the XERCES library.

Instrumentation

The final package, Instrumentation, contains functionality to instrument the usage of the Tool. This is useful when one wants to investigate how the tool is used to improve
on it. Currently, the only thing that makes use of this instrumentation interface is a logging class.

6.4 Package Description

6.4.1 GUI

Purpose
The GUI package contains a collection of classes that make up the Graphical User Interface of MetricView Evolution or directly support it in some way.

Subordinates
MainFrame, TimelinePanel, SQLDebugConsoleDialog, GlobalOptionsDialog, CheckListCtrl, LayoutPreview, ProgressUpdater and ArtProvider

Dependencies
The original design for the communication between the GUI package and the rest of the application was to use the Model-View-Presenter pattern, thereby having a loose coupling. In its current form the coupling between GUI and other parts of the application is much more direct. Therefore this package currently depends on almost all other packages. Again, this is due to the prototype kind of development process.

Interface
The package does not feature an external interface. The reason for this is that GUI is not depended on directly by other packages.

Main Classes

MainFrame This class provides the main application window. It builds up this window and fills it with the other widgets. Most functionality of this class has to do with either construction of the main window or event handling of the main GUI elements.

TimelinePanel The Timeline is a rectangular widget at the bottom of the window. Its main goal is to visualize the versions of a model that are loaded into the tool and which of these versions is selected. Additionally it provides a quick overview of the variations of one quality-attribute or metric measured in these different versions of the model.

The Timeline widget is visually subdivided into years, month and even days if there is enough space on the screen. Internally a procedural approach is taken to this subdivision. There is a procedure to draw years, months and one to draw days. The alternative would have been to create different classes for years, months and days that draw themselves.
The decision to use procedures was taken based on the assumption that this would allow faster redrawing the widget without too much extra complexity.

**Miscellaneous** Other classes in the GUI package are:

- **SQLDebugDialog.** Provides direct access to the underlying relational database from the GUI by means of SQL queries. The main reason this dialog was integrated is that it was thought to be very helpful during debugging. Not only can relevant information be extracted, but it is also possible to make modifications when necessary.

- **GlobalOptionsDialog.** Functions as a wrapper around the global Configuration structure, thereby enabling users to tweak most settings. It automatically generates a tree from the configuration parameters that are available.

- **CheckListCtrl.** A custom ListControl that is used to represent the metrics that are available in the model. In this function it is assisted by the Quality Tree.

- **LayoutPreview.** A fully custom widget that shows the layout and coloring of the metrics as they are mapped onto the model elements.

- **ProgressUpdater.** A wrapper around the status bar of the tool. The reason for this wrapper is that it makes it easier to replace the actual implementation of progress indication when necessary.

- **ArtProvider.** Provides access to the art (icons, images, etc.) used throughout the user-interface. This art is loaded from file on request and stored inside the class during a run. This is an improvement over the old situation in the original MetricView where the art was hard-coded in a source file using the XPM format.

**Things that should be improved**

- **MainFrame. Because MainFrame fulfills three tasks it is a little complex. This complexity could be reduced if the instantiation, wiring and large parts of the event handling would be delegated to some other class(es).**

- **TimelinePanel.** The signatures of the drawing methods have too many parameters at the moment.

- **Event handling.** Currently the event handlers of widgets directly call the backend of the tool. To improve the extendibility the coupling should be loosened by using a Model View Presenter structure. This means that the widgets only report to the Presenter what gesture the user made and that the conversion from user-interface gestures to actual operations is done centrally in the presenter. An example implementation is present for TimelinePanel, which uses Presenter to notify Storage of changes in the selected version of the model. Other parts of the system like WindowCompositor have already been adapted to the Model View Presenter structure.
6.4. PACKAGE DESCRIPTION

6.4.2 Control

Purpose

The Control package facilitates the communication between (parts of) other packages by providing a collection of interaction patterns. The first example is the Model-View-Presenter pattern. It is used to decouple models from views on these models. Part of the responsibility of the Presenter of this pattern also lies in maintaining selections in the model, this functionality is also present in the Control package. The interaction second pattern, assisting the first in several ways, is the Publish-Subscribe pattern. It is used to notify interested parties in events that occur throughout the application.

Subordinates

Controller, Presenter, SelectionManager and SpotManager

Dependencies

Control has a large number of incoming dependencies. This is because a large part of the communication between other packages happens through some mechanism that is implement in Control.

Interface

Because control provides the plumbing and wiring of MetricView Evolution and this application was developed in a prototyping manner it consists of a relatively large set of interfaces. The interfaces can roughly be divided into 3 categories:

- Model-View-Presenter related. The Presenter implements a number of interfaces: AbstractPresenter to accept gestures, AbstractInvoker to store and execute commands and AbstractGestureVisitor to differentiate between different types of gestures. The AbstractGesture interface form the complementary part in this Visitor pattern. For AbstractInvoker, the complementary interface is AbstractCommand. Commands that are stored and/or executed need to implement this interface, which consists of nothing more than a execute method. Everything else, like parameters, is passed in the construction of concrete commands. The events that travel between Model and View implement the CustomEvent interface that is related to the IEventVisitor interface, again to implement a Visitor pattern.

- Publish-Subscribe related. The two main interfaces used for the interaction in the Publish-Subscribe pattern are IPublishSubscribeManager and ISubscriber. The former is used to (un)register classes that are interested in receiving certain events at the Controller. The latter is the interface such class needs to implement to be able to receive notifications.

- Selection related. The selection manager keeps track of elements that implement the ISelectableElement interface and are selected. The default behaviour is that
When the `setSelected` method is called in this interface, the element (un)registers itself at the selection manager depending on the value of the first parameter. To break the cyclic dependency between elements and the selection manager the `ISelectionManager` interface is introduced.

**Main Classes**

An overview of the internal structure of the Control package showing the main classes can be found in figure 6.3

**Controller** The `IPublishSubscribeManager` interface described above is implemented by the Controller class. It stores a mapping from event types to collections of subscribers to keep track of which class needs to receive what notifications.

**Presenter** Responsible for the translation of gestures into commands that manipulate the model, or directly into events. The Presenter class is therefore split into two parts: `GestureToCommand` and `GestureToNotification`. The first part does translate gestures into commands, the second part translates gestures into notifications. The latter is used to communicate requests to change the view to each view that is affected. Presenter, `GestureToCommand` and `GestureToNotification` all use the visitor pattern to distinguish between gesture types.
6.4. PACKAGE DESCRIPTION

**PublishSubscribeManager**  This manager class maintains a mapping from event types to lists of subscribers that are interested in those events. When an event of a given type is published it notifies all subscribers in the list.

**SelectionManager**  The ISelectionManager interface is implemented by this class. It keeps track of the collection of elements that is selected at any given time during the execution of the program. In general, when an element is selected, the setSelect method of the element is called with the SelectionManager as argument. The element will then register or unregister itself at this manager depending on whether selection or unselection took place. Selection in the QualityTree is a little more involved. To visualize the dependencies of a metric or quality attribute the selection is cascaded. This is done using an observer pattern in which the QualityTree functions as an observer and the vertices in the tree function as observables. When a vertex is selected it notifies the tree that cascading needs to be performed. To prevent infinite loops in this process the tree temporarily enters a cascade mode in which it ignores further cascade events. It calculates the upstream and downstream dependencies of the edge or node that was selected and propagates the selection to all the dependencies.

**SpotManager**  This class facilitates the wiring between the search and highlight functionality in the user interface and the underlying implementation in DataStorage. It also maintains a list containing the results of the current search.

**Pitfalls**

In a strict implementation of the Model-View-Presenter pattern, the Presenter should determine and store the selection. At the moment this is not the case and the selection is determined in the visualization package. The reason for this is that drawing functionality is used to make selection fast enough even with large models. The selection is communicated through gestures, which is not optimal but works. An example interaction related to the implementation of the Model-View-Presenter is figure 6.4.

SelectionManager does not differentiate between different types of selectable elements and maintains only a single selection set. This was a design decision to keep selection management simple.

The interactions that take place to handle the selection of a diagram element are somewhat intricate. First there is WindowCompositor using Canvas to get selection events published, instead of directly communicating with the Selection Manager. Canvas propagates the selection events to the controller, which notifies SelectionManager. The second intricacy is that selection is defined on model element level instead of diagram elements. However, the selection is first performed on diagram elements that propagate it to the underlying model element. The reason for this is that if selection events would contain a reference to the model element instead of the diagram element, information about which visual representation was selected would not reach the Presenter. This information is vital for focussing (automated zooming) on a specific diagram element. The Sequence diagram of figure 6.5 describes these interactions.
Figure 6.4: Focus interaction scenario using Model-View-Presenter pattern

Figure 6.5: Selection of a Diagram Element/Model Element
Things that should be improved

6.4.3 DataStorage

Purpose

DataStorage fulfills the role of Model in the Model-View-Presenter pattern. It is the place where all domain related data is stored. To meet this goal DataStorage includes a relational database and a series of layers that convert the data from the database into object models. These object models are faster and easier to use throughout the rest of the application than queries would be.

Subordinates

DataBaseWrapper, DataBaseWrapper, ModelStorage, DiagramStorage, MetricStorage and QualityTreeStorage.

Dependencies

DataStorage only depends on the Control package. This dependency is needed for publishing events when something changes in the model.

Interface

DataStorage is accessible through 2 main interfaces, one for read-only access and one for read/write access. The reason behind this is that restricting the read/write access only to classes that actually need it should improve maintainability.

IStorageRetrieval This interface is used to retrieve data from the storage package. It is divided into several sub-interfaces by means of aggregation. The reason for this is that it increases maintainability, because changes that are made have a more local effect and for understanding the complete interface one only needs to understand the sub-interfaces which can be inspected independently. The sub-interfaces of which IStorageRetrieval consists are:

- IModelStorageRetrieval. Used to retrieve model elements from the storage package.
- IDiagramStorageRetrieval. Used to retrieve diagrams and diagram elements from the storage package.
- IMetricStorageRetrieval. Used to retrieve metrics information from the storage package.
- IQualityTreeStorageRetrieval. Used to retrieve (parts of) the quality tree from the storage package.
• IRelationStorageRetrieval. Used to retrieve inter-diagram relations from the storage package.

In general it holds that single elements are retrieved using an identifier, version pair as a key and a pointer to the element will be returned. There is only one instance of a model element for each key. Multiple elements can be retrieved by passing a reference to a list that will be filled with the results.

IStorageManipulation This interface is used to directly manipulate data in the storage package. The current design tries to constrain its usage to the Manipulation and FileIO components. It implements IStorageRetrieval so that it is not necessary to pass two references to components that need both retrieval and manipulation. Just as IStorageRetrieval and for the same reasons it is divided into several sub-interfaces by means of aggregation:

• IModelStorageManipulation. Used to manipulate model elements in the storage package. This interface is not actively used in the current version of MetricView Evolution because editing of models is not supported.

• IDiagramStorageManipulation. Used to manipulate diagrams and diagram elements in the storage package. This interface supports creating new diagrams, setting a position, assigning a diagram as a context diagram and removing all diagram information of a specific version.

• IMetricStorageManipulation. Used to manipulate metrics in the storage package. Currently only supports inserting new metrics.

• IQualityTreeStorageManipulation. Used to manipulate the Quality Tree. An important method in this interface is updateMetrics. The Quality Tree is used as a function. The parameters are given as metrics and the updateMetrics method is used to evaluate the function and insert the result into the Quality Tree. There is only a single instance of the Quality Tree in the storage package so it is vital to update it each time a different version of the model becomes active. However, instead of doing this directly using this interface, the IMetricManipulation interface should be used.

• IRelationStorageManipulation. Used to manipulate the inter-diagram relations in the storage package. Currently only supports adding new relations, marking a set of diagrams as a cluster, and removing all relation information for a specific version of a model.

Main Classes

Figure 6.6 shows an overview of the internal structure of the DataStorage component.

At the core of the DataStorage package is the DataBaseWrapper class. It provides query functionality on a relational database and converts the results into tuples of
Figure 6.6: Internal structure of DataStorage
strings. Around this database core are 3 classes that convert between relational representation and object representation: ModelStorage, DiagramStorage and MetricStorage. They do this respectively for UML Models, Diagrams and Metrics. ModelStorage and DiagramStorage both use a caching scheme to minimize database access and thereby improve performance. The final two subordinates do not have a model representation in the relational database. QualityTreeStorage is responsible for storing a representation of the Quality Tree, in RelationStorage all inter-diagram relations are collected.

The DataStorage class itself acts as a facade for its subordinates. ModelStorage, DiagramStorage, MetricStorage, QualityTreeStorage and RelationStorage implement the sub-interfaces of IStorageRetrieval and IStorageManipulation. In this way changes to the interfaces have only local impact while it is still possible to grant full access to all storage functionality by passing only a single reference.

### Pitfalls

- DataStorage, and DatabaseWrapper in particular do not support concurrent access.

- The DatabaseWrapper implementation does not support typing of the data that is communicated with the database.

### Things that should be improved

The main improvements that DataStorage would benefit most from both have to do with the manipulation part. Currently manipulation is very limited, it is not needed because the tool does not support editing operations. To be able to provide editing support the manipulation interfaces should be extended to match the level of the retrieval interfaces. Doing this would also require to update the way caching is implemented. Changes in cached data should be made persistent in the underlying relational database. Another improvement would be to add support for concurrent access. This would involve adding some kind of locking strategy for the attributes in the various storage classes and the wrapper around the database itself. Also all methods would need to be reentrant. DataStorage would also benefit from exception handling. This would not only improve robustness but also make it easier to debug. Another thing that would improve robustness and debugging capabilities would be to add typing support to DatabaseWrapper implementations.

### 6.4.4 Manipulation

### Purpose

This package consists of classes that perform manipulations on DataStorage. These have been grouped into a package to make it easier to identify them. Together with the classes in the FileIO package they should be the only ones to make use of read/write access to DataStorage.
6.4. PACKAGE DESCRIPTION

Subordinates
LangeDiagramLayout{FD/Sort/Wheel}, ContextExtractor, RelationsExtractor, MetricsExtractor, QualityTreeLayout, PythonInterpreter

Dependencies
Manipulation depends on DataStorage for retrieval and storage of manipulated data.

Interface
The Manipulation interface is used to perform manipulations on data in the storage package. Instead of scattering these manipulations throughout the application they are grouped in the Manipulation package. Manipulation is also responsible for storing the results of these manipulations into DataStorage. Just as the DataStorage interfaces, the Manipulation interface is divided into sub-interfaces that each offer specific functionality:

- IModelManipulation. This interface is not used and currently empty because editing of models is not supported in the current version of the tool.
- IDiagramManipulation. Offers functionality to layout the Lange-Diagram.
- IMetricManipulation. Can be used to update the values in the Quality Tree, preferable to using IQualityTreeStorageManipulation directly.
- IQualityTreeManipulation. This interface is used to layout the Quality Tree.
- IRelationManipulation. This interface is used to extract all inter-diagram relations of a specific version of the model and store them.

Main Classes
- Manipulation, acts as a facade, implementing the interfaces described below by calling the subordinates that provide the needed functionality.
- LangeDiagramLayout{FD/Sort/Wheel}. These three classes provide different layouts for the Lange-Diagram. The default layout of the diagrams in the Lange-Diagram (Sort) is similar to the layout of text that is centered in a text editor. Each type of diagrams has its own 'paragraph' called a bucket. Inside this bucket, diagrams are sorted based on their height. After this the 'lines' of the paragraph are 'written'. Finally the lines are centered horizontally and vertical spacing is applied.
- ContextExtractor, is responsible for calculating the context of a model element with respect to a metric. Because context calculation is now implemented in the external (Python) script calculating the metrics, ContextExtractor only retrieves the needed information from the database and generates a diagram from it.
• RelationsExtractor, extracts inter-diagram relations from the model. It does so by inspecting relations between model elements that cross the boundary of a single diagram. An example is the identity relation, whereby a single element exists on multiple diagrams.

• MetricsExtractor, is used to calculate metrics over the model. It uses Python-Interpreter to reach this goal, because metrics are defined using a combination of Python scripts and SQL. PythonInterpreter is not called directly but through MetricStorage, which caches the results.

• QualityTreeLayout, creates a (fixed) layout for the Quality Tree.

• PythonInterpreter, enables execution of Python scripts that are used for metrics calculation and the evaluation of quality functions in the Quality Tree. The connection between the main application and the python scripts is bi-directional. From the tool functions inside a Python script can be called and these functions can make calls to the application. The latter communication type is used to access the database from the scripts and to redirect debug output to the log file of the main application. In the case of quality function evaluation the communication from the tool to the script needs a translation. It involves converting metric data from C++ objects to Python objects. The metric data can include tagged values such as the weight of a value. To store these tagged values a wrapper object is used in Python. This wrapper object encapsulates a basic data type like boolean, int or float. During the translation from C++ objects to Python objects the tagged values are stored as attributes of the wrapper object. In Python it is possible to do this adding dynamically. In theory, because basic data types are also objects, it should be possible to just add the attributes to the translated metric values themselves. In practice however, certain optimizations, block this functionality. The downside of using wrapper objects is that every variable used in the quality function scripts must be encapsulated by such an object or it will not be possible to apply operations on that variable and a certain metric value. An overview of the classes related to quality functions can be seen in figure 6.7, the evaluation of a quality function of an attribute is described by the Sequence diagram in figure 6.8.

Things that should be improved

MetricsExtractor relies on MetricStorage to perform the actual extraction using the integrated Python Interpreter. This goes against the design and might make it harder to understand. MetricExtractor should use PythonInterpreter directly instead and insert the results into DataStorage.
6.4. PACKAGE DESCRIPTION

Figure 6.7: Classes related to QualityFunction
6.4.5 Visualization

Purpose

The purpose of the Visualization package is to offer views on the models stored in DataStorage. Besides functionality to generate views it also contains the low level part of the implementation for user interaction with these views. An example of this is selection, that uses a rendering functionality very similar to the one used for visualization.

Subordinates

Canvas, WindowCompositor, Window, LayerCompositor, Layer, LangeDiagram, UMLModelVisualizer, QualityTreeVisualizer, GraphicsGL

Dependencies

Visualization depends on DataStorage for retrieving model data, indirectly on GUI for the output to the screen, and on Control for user interaction.

Interface

The Visualization package currently does not offer a set of well defined interfaces for other packages to use. The reason for this that the prototyping inflicted many changes
to the classes inside the package. Below is a description of the interfaces of the most important classes in the Visualization package:

- GraphicsGL. At the bottom of the Visualization package we find the GraphicsGL class. It provides a collection of drawing routines that ease the use of drawing primitives and text using the OpenGL library.

- UMLModelVisualizer. The collection of UMLModelVisualizers offer functionality to render models. They use the Visitor pattern to distinguish between the different elements that can be on a diagram, hence the many Visit methods.

- LangeDiagram. This class is responsible for visualizing a Lange Diagram. It does so by making use of a UMLModelVisualizer to render each of the diagrams. Additionally it renders the lines that represent the inter-diagram relations and groups of connected diagrams that make up clusters. The methods that are important to make use of LangeDiagram are addDiagram to fill it with diagrams and drawContent to initiate the rendering. Finally, it is important to call setPublishSubscribeManager to have LangeDiagram register itself for the events it needs to be notified about.

- Layer. A Layer represents a single version of the model. It has a drawContent method to render the Lange-Diagram it contains. In theory each layer could have its own UMLModelVisualizer but currently only one visualizer is used.

- LayerCompositor. This class manages the rendering of all layers that are within the model, this can be initiated by calling drawContent. Just as with LangeDiagram it needs to register to event notifications; for this reason setPublishSubscribeManager needs to be called.

- Window. A window represents a view on the model. In the current version of MetricView there are windows for displaying the UML Model, the Quality Tree and for showing the context of an element. The interface is very small, only two rendering methods and getters to receive the size of the window.

- WindowCompositor. In early prototypes of the tool windows could be stacked on top of each other to overlay different views. The class that was responsible for this is called WindowCompositor. Currently it is responsible for event handling within a window, world-space to screen-space transformations and focussing (automated zooming). To influence the transformations panning, zooming and rotation parameters can be set in addition to a viewport.

- Canvas. At the top of the visualization hierarchy is the Canvas class. It supports splitting the drawing canvas into several windows and is also responsible for routing events to the correct WindowCompositor.

**Main Classes**

Figure 6.9 shows an overview of the internal structure of the Visualization component.
Figure 6.9: Internal structure of Visualization
6.4. PACKAGE DESCRIPTION

**LangeDiagram**  A Lange diagram is best described as a meta diagram. It consists of the different views on the software model and their relations. The views are grouped in different layers on abstraction level and range from requirements at the top to implementation class diagrams at the bottom. The layers that are implemented in MetricView Evolution are the use cases, scenarios (or message sequence diagrams), design classes and state diagrams.

The main purpose of the Lange diagram in MetricView Evolution is navigation. It is a replacement of the model explorer in the original MetricView (the treelist widget on the left side of the window). It should provide the user that needs some specific information from the model quick and easy access to that information. Additionally, users that are not looking for specific information but merely try to understand the model should be presented with an overview of elements that are present in the model and the relations between these elements.

The relations that are currently implemented in MetricView Evolution are:

- **Use Case - Scenario.** A Scenario, depicted in a message sequence diagram describes the interactions in the system that take place to execute a Use Case. In UML the link between these two consists of the owner of a message sequence diagram being a Use Case.

- **Scenario - Classes.** The interactions depicted in a message sequence diagram that describes a scenario take place between instances of classes. In UML various links between Scenarios and classes can be found. The one that is currently being used is the type of an object (ClassifierRole) in a message sequence diagram being a class. Another possibility would be to use the relation between the messages in the message sequence diagram and the methods in a class.

- **Classes - State Diagrams.** The internal behavior of a class can be modelled using a state diagram. In UML the link between classes and state diagrams consists of the owner of the state diagram being a class.

LangeDiagram is responsible for the visualization of the Lange-Diagram. It retrieves its information out of the diagrams that are added to it (using the addDiagram method). Rendering the Lange-Diagram currently is a two step process. Visualizing each of the diagrams that are part of it and rendering the relations between the elements inside these diagrams and the diagrams itself. The first part, visualizing the diagrams, is delegated to a UMLModelVisualizer, and buffering is used to speed up this process in the case that nothing has changed. Drawing the relations is done inside LangeDiagram and is not buffered because it depends on the filtering that is active (currently selection of elements).

A part of the interactions that take place when the canvas is to be refreshed is modelled in the sequence diagram in figure 6.10.
Pitfalls

Things that should be improved

- Support for other diagram types should be added (e.g., Activity Diagrams).
- Filtering should be implemented such that certain diagram types can be hidden at the users request.

6.4.6 FileIO

Purpose

FileIO facilitates the input and output of data in the DataStorage package from or to files.

Subordinates

QualityTreeParser

Dependencies

This package depends on DataStorage to store and retrieve its data.

Interface

FileIO The FileIO package can be accessed through the IFileIO interface. In general these interfaces are only used to initiate data input or output. The actual communica-
6.4. PACKAGE DESCRIPTION

tion with DataStorage is handled by FileIO internally. IFileIO is sub-divided into the following interfaces by means of inheritance:

- IFileIOXMI. Currently only has one method, to load a model from an XMI file.
- IFileIOQualityTree. Used to load a Quality Tree from an XML file.
- IFileIOMetrics. Deprecated, because metrics are calculated internally.

Main Classes

- FileIO. This main class acts as a facade for other classes that actually implement the functionality.
- QualityTreeParser, sets up a parser using XERCES and register QualityParser-Handler to receive the necessary notifications.
- QualityTreeParserHandler, provides the actual implementation of the QualityTree parser. Consists of a collection of methods for each type of node. When such a node is encountered by XERCES the corresponding method is called.

As stated before, a large part of the FileIO functionality is actually outside the application. The XMI parser is implemented as a set of XSL transformations. For each supported type of model element a stylesheet is defined that converts the XML definition into a SQL query that can be used to load the element into the database. There is also a ‘version’ stylesheet that attempts to extract version information from XMI, if this fails the application falls back on the creation date of the file. Most of the other stylesheets need this version information because it is part of the primary key of most database tables. The information extracted by the ‘version’ stylesheet or the fallback mechanism is integrated with the queries from the other stylesheets just before the queries are fed to the database.

Things that should be improved

FileIO currently only supports input. Output capabilities for metrics and reports should be added. And implementation of metric output to XML is available in the command line version of the tool.

6.4.7 Instrumentation

Purpose

The purpose of the Instrumentation package is to provide a mechanism to extract information about the run-time behavior of the tool. This information can be used to help debugging, or analyze usage patterns.
Dependencies

Instrumentation does not depend on other packages. At the moment all packages that have functionality that needs to be instrumented depend on the Instrumentation package.

Interface

- InstrumentationKind. This is the interface through which classes that are instrumented report what events occurred and subscribers can be registered to receive (some of) these events.

- InstrumentationSubscriber. This interface, that has only one method (accept), has to be implemented by all classes that need to be notified when some instrumentation event occurs.

- AbstractInstrumentationData. This abstract class defines the interface for data that can be send together with an instrumentation event. At the moment it is limited to a single method that converts the data to a string, which is used for logging purposes.

Main Classes

Figure 6.11 shows an overview of the internal structure of the Instrumentation component, its main classes are:

- InstrumentationKind. This class basically works as a Publish-Subscribe manager. It handles the registration of subscribers that are interested in specific events and when it receives an event it routes it to subscribed clients.

- Logging. Implementation of the InstrumentationSubscriber interface that provides some basic logging functionality.

- InstrumentationDataGeneric. A data class that can be used to communicate a string through an instrumentation events.

Pitfalls

Because of all the incoming dependencies, changes to the Interfaces have a large impact on the rest of the system.

Things that should be improved

- The number of incoming dependencies should be reduced. This could be done by extending the use of the Model-View-Presenter throughout the application. Instrumentation could then be hooked into the Presenter to capture interactions.
Figure 6.11: Internal structure of Instrumentation
Chapter 7

MetricView Evolution - Implementation Notes

In this short chapter the motivation for some parts of the implementation is discussed.

7.1 DataStorage

Other embedded databases that were tested, however, all required substantial changes to part of the application that generates the object model. In hindsight, the prototyping approach failed in this case because too much of the implementation already depended on MySQL.

7.2 Visualization

To increase performance LangeDiagram uses a rendering buffer. This buffer stores all commands that are needed to visualize a diagram in an OpenGL display list. As long as the visualization doesn’t change this display list can be executed, which is faster that executing each of the commands in the list separately. The downside of this approach is that there is no control afterwards on the order in which the elements on a diagram are drawn. This is a problem when transparency is used. For proper rendering the primitives should be rendered from back to front based on their z-coordinate. This order changes when the diagram is for instance rotated. The solution with the best visual result would be to sort transparent elements each time before they are drawn. The main problem with this approach is that it would involve a large change in the way the rendering is performed or it would result in a major performance hit. The choice was made to live with the glitches it causes and just document the problem.

A similar problem occurs at diagram level. Because each of the diagrams is rendered separately the order in which the individual elements can be drawn is rather restricted. To minimize problems with rendering of transparent elements without loosing too much performance and adhering to existing design of the rendering package the choice was
made to sort the diagrams on their z-coordinate and render them from back to front.
Chapter 8

Validation

The goal of this validation chapter is to find out whether the tool is really usable to monitor the quality of software architectures. Not only will the functionality be tested but also the visualization techniques that are used and the way the user can interact with the tool through the graphical user interface. This test consists of a small experiment held among coworkers. Another section is dedicated to the integration that was performed at an industrial site and the results of this integration.

8.1 Case Studies

One part of the validation consists of applying MetricView to several different cases. This is mainly done to test the functional working of the tool. It also functions as a first usability test. The observations that were made in each of the cases are described below. General information about the size of the cases is presented in table 8.1 and 8.2.

8.1.1 TrafficLight Control System

One of the first cases that is used to test MetricView Evolution is a small model developed by OOTI students at the Stan Ackermans Institute. As the name suggests the model describes a control system for traffic lights.

<table>
<thead>
<tr>
<th>Diagram type</th>
<th>TLCS</th>
<th>Case B</th>
<th>Case C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case</td>
<td>3</td>
<td>35</td>
<td>4</td>
</tr>
<tr>
<td>Message Sequence</td>
<td>4</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Class</td>
<td>5</td>
<td>152</td>
<td>22</td>
</tr>
<tr>
<td>State</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Collaboration</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Module</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>21</strong></td>
<td><strong>188</strong></td>
<td><strong>47</strong></td>
</tr>
</tbody>
</table>

Table 8.1: Basic View size for various cases
CHAPTER 8. VALIDATION

<table>
<thead>
<tr>
<th>Element type</th>
<th>TLCS</th>
<th>Case B</th>
<th>Case C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packages</td>
<td>14</td>
<td>184</td>
<td>15</td>
</tr>
<tr>
<td>Actors</td>
<td>6</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>Use Cases</td>
<td>11</td>
<td>141</td>
<td>11</td>
</tr>
<tr>
<td>Scenarios</td>
<td>4</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Objects</td>
<td>40</td>
<td>0</td>
<td>103</td>
</tr>
<tr>
<td>Messages</td>
<td>27</td>
<td>0</td>
<td>210</td>
</tr>
<tr>
<td>Classifiers</td>
<td>36</td>
<td>467</td>
<td>75</td>
</tr>
<tr>
<td>States</td>
<td>19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Transitions</td>
<td>39</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>196</td>
<td>817</td>
<td>438</td>
</tr>
</tbody>
</table>

Table 8.2: Basic Model size for various cases

One of the things that were noticed using the tool on this case is the added value of the search functionality. When one searches on the keyword 'TrafficLight' the tool automatically highlights a use case ('WaitAtTrafficLight'), a class ('TrafficLight') and a state diagram ('TrafficLight'). Also two relations are shown: one from an object to the class 'TrafficLight' and one from the highlighted use case to a package 'CrossRoad'. The benefit of this is that one can immediately spot all information that is related to traffic lights. When a user is exploring the model to gain a better understanding of the system this feature shows relations that are not explicitly modelled. An example of this is the state diagram 'TrafficLight' that describes the behavior of the class 'TrafficLight'. This relation is only implicit (through naming).

8.1.2 Case B

The first large case used to test MetricView Evolution is denoted as ‘Case B’. The size of this case revealed some scalability issues, mainly in the visualization part of the tool. Because the drawing code originally designed for single diagrams is now used to render a Lange-Diagram potentially consisting of hundreds of diagrams this performance issues were very likely to arise. But performance was not the only problem, also the visualization of the Lange-Diagram itself was less scalable than expected. For large cases with many diagrams there is little space available for each individual diagram. Space efficient layouting helps to some extend to reduce the effects of this problem. Related to this is the problem that for large cases the values of the individual metrics are hard to judge when viewing the complete Lange-Diagram. This activity is performed when looking for outliers. Instead of trying to increase the visibility of the individual metric values, active support for the task of outlier detection was added to the tool. It also turned out to be hard to find information about a specific model element if only (part of) the name was known but not the diagrams it occurs in. To assist with this task the search and highlight functionality proved to be very helpful.
8.1.3 Case C

This case is only used in the usability experiment. One of the reasons for this is that it has a decent size and features relations that can be traced from use case level down to class level. It is large enough to have outliers for some metrics but small enough to comprehend fairly quickly.

8.1.4 Observations when using the tool during case analysis

One of the tasks that the tool supports and can really save user’s time compared to performing the same task without this tool or with other tooling is outlier detection and analysis. This point has already been stressed some times in this document and the following example will show that this claim holds. One of the tasks that have been performed regularly within the EmpAnADa project is creating an analysis report for a case. Such report consists, among other things, of quite some tables showing outliers for all kinds of metrics applied to the model. The general way to create these tables is to run a set of analysis tools, combine the results in an spreadsheet, calculate statistics, perform outlier detection by sorting the data according to each of the metrics and finally extracting the outliers. MetricView can automatically perform these steps and output the results to an XML document. If necessary, this document can be transformed into HTML, or some other format that is easily human readable. Not only does this save a lot of time, it also reduces the probability of mistakes introduced by human error.

Once this report is generated, MetricView can be used to inspect the cause of the outliers. This can of course also be done using an ordinary CASE tool but this takes a lot more time. Typically the steps involved in the inspection process using a normal CASE tool are: Select some metric, find the outliers in the corresponding table in the report, find the corresponding model elements in the CASE tool and look up all diagrams in which they occur, create a mental representation of the information that is combined in all those diagrams and filter out anything that is not relevant with respect to this metric and finally make an analysis as to why this causes an outlier. Using MetricView Evolution one only has to select the metric, focus on a outlier marked in the Lange Diagram and request a context diagram that contains all the relevant data before one can embark on the analysis itself. It is not difficult to see that this saves much time, especially when large models need to be analysed for many metrics.

8.2 Usability Tests

8.2.1 Setup

To evaluate the usability of the tool a small experiment has been performed. During this experiment 13 coworkers have evaluated the tool by performing specific tasks. These tasks were put in the form of answering the following 6 questions:

1. For what classes does the value of the metric "cbo1" (Coupling between Objects) exceed the threshold of 5? The tool has already been configured to have a threshold
of 5 for this metric.

2. How many descendants (so, children and children of children and so on) does the Dispatcher class have? (NOD: Number of Descendants) Please, also state (abbreviated) names of these classes.

3. Which class plays the major role in the implementation of the "Initialization" Use Case?

4. For which Use Cases is the Request class of importance?

5. What trend can be observed in the quality of the 3 versions of this model. What is the cause for this trend? Please state the name of quality attribute(s) and metric(s)

6. What diagram(s) has/have been changed between the three versions?

The first 4 questions had to be answered about medium sized model described above as Case C, the last 2 questions were about 3 versions of the small TLCS model also described above. The idea behind these questions is to let the user explore the various features the tool has to offer without forcing him to follow a specified path. This should allow for some creativity and therefore interesting observations.

At the start of the user-experiment the background of each participant is noted using a short questionnaire. After this questionnaire the tool is explained to the participants by means of an example model that is different from the two models that are used during the experiment itself. After the explanation the users have time to answer the questions. While they are busy doing this they are encouraged to answer questions and make remarks about what they are trying to do and how they feel about the tool. Finally they have to fill in a second questionnaire that is used to evaluate how useful the user thinks different features of the tool are for doing different categories of tasks. The experiment was performed in 4 sessions. The first session partly functioned as a pilot-study to fine-tune some of the questions.

8.2.2 Observations

One of the most important observations made during this experiment is that users find very creative ways of using features that the designer had not thought of. Sometimes this works as expected but more often than not it doesn’t work and results in confusion. This confusion will lead to a lower perceived quality of the tool. An example of this is the context view. Some users tried to request the context of something inside the context diagram. The tool did initially not allow this but did not notify the user that it is not possible. Because of this the users assumed they had done something wrong and kept trying in different ways. Another observation is that once the user has found something that works well it will try to apply the same feature to solve other problems. Again the context view is a good example of this. The quickest way to answer the first two questions was to use the context view. Most users, either unconsciously or
not, assumed that they could also use it to solve the third question and tried to find a suitable metric that would yield a Context View that would provide the information to answer the question. A third observation very much related to the last is that users will try to use gestures they know from other applications. One user tried to select an element by dragging a square around it because this is the way to select something in many applications. Almost all users tried to double-click on elements to focus on them instead of using the alt+click combination. Even after telling the users that alt+click was the correct gesture a few minutes later they automatically tried to use double-click again.

8.2.3 Problems and Suggestions for improvement

From the observations stated above and the comments that the users made during the experiment two lists have been formed. The first list contained problem reports, bugs that made it impossible or very difficult for a user to complete a task. The other list contained change requests. Items on both lists were given a severity and impact level. The first to indicate how severe the problem is, the second is an estimation of the amount of effort that is required to fix the problem or make the change. Combining those two attributes results in a priority for each of the items. This priority is used to decide what items will be fixed and what items will be left as future work. The most noticeable suggestions we got from users are:

- The Timeline shows great potential and should be the starting point from which to explore the quality of a model. For this to work better it should be possible to show trends of multiple metrics or quality attributes together in different colors. Also showing the actual values in the Timeline together with the graph is a suggestion we got from multiple users.

- The task of finding differences between two or more versions of a model is complicated by the fact that for each of the Lange Diagrams a separate layout is created. It would be much more useful if one unified layout was created that made sure that each diagram is on a fixed position in the plane. This would make it much easier to spot changes within diagrams. The reason for this is that it is hard to see changes in something that moves, this is what is perceived when the user quickly changes between versions.

8.2.4 Results

The analysis of the result is split up into three parts. First some user background will be presented. After that the users performance with respect to answering the questions with the tool is analyzed. The last part is about the results of the second questionnaire that was used to get feedback about the expected usefulness of the tool to perform different kind of real-world tasks.
User Background

Most of the participants in the experiment have a Master degree in computer science. The others are either working on their Masters project or have a different background like electrical engineering. Their knowledge of UML diagram types is good, most have applied it in either academic context or a project with external clients. The most well-known diagram type is the class-diagram closely followed by the sequence diagram. The participants’ knowledge of software development is also sufficient for this experiment. The level of software visualization knowledge is a little lower although there are some noticeable exceptions. Something similar can be said about knowledge of software metrics.

8.2.5 Tasks

The first question is answered correctly by all participants. The same holds for the second question. The third question was answered incorrectly by one participant. In answering the fourth question some participants during the first session were confused by the way the inter-diagram relations are drawn. The lines that represent these relations were not completely connected to some use cases, but a little apart from the edge. This could cause confusion about to which use-case such a relation was connected when viewing it from the Lange-Diagram overview. The fifth question was very much an open question stated in such a way to encourage users to form their own opinion about the quality of the model using the QualityTree and Timeline features of the tool. Therefore there is not really one correct way to answer the question. The sixth and final question is answered correctly by all users although some go through great lengths to explain exactly what has changed while others simply state the name of the diagrams. During the experiment some timing information was collected. From this information we can conclude that most time was spent on the last two questions, in some cases more than twice as much as on other questions. The timing information also gives a hint about why one user failed to answer the third question correctly. He used the least time to answer all the questions, investigation showed that he was in a hurry to get back to work.

8.2.6 Evaluation

The highest rated features with respect to usefulness for correctly performing tasks are: Lange Diagram, Search and Highlight and Context View. With respect to usefulness for fast and efficiently performing tasks the highest rated features are: Search and Highlight, Lange Diagram and Design Smells. For this last feature the result is remarkable because it was not been actively used during the experiment. Another view on the results is offered in figure 8.1 and 8.2. These histograms show for each task type what percentage of the users found a particular feature useful.

The evaluation of the understandability of the tasks shows that the Quality Tree was the easiest to understand. Of the 4 main views, the Context View was hardest to understand. Observations made during the experiment reveal that most users had
8.2. USABILITY TESTS

Figure 8.1: Usefulness for correctly doing a task

Figure 8.2: Usefulness for efficiently doing a task
problems activating the Context View, because it involves actions in multiple windows that have to be performed in the right order. These observations also made clear that once the Context View was activated participants had little problems using it. Overall Design smells scores the lowest, as stated above this feature was not actively used during the experiment, which might be a reason for this low rating.

The aggregated results of the evaluation are shown in appendix C (tables C.1, C.4 and C.5). With respect to usefulness the participants were asked to mark the feature they expected to be useful for a given task. Understandability and how intuitive a feature is, was rated on a scale from 1 to 5. For the features Search and Highlight and Context View, figures 8.3 and 8.4 show a histogram of the ratings. It can clearly be seen that search functionality was much easier to use for most participants than the Context View. Searching being a common feature in many tools, this result is not strange.

8.2.7 Conclusions

The results in this section show that in general the tool was rated quite good. Some features scored noticeably higher, like the search functionality and the Lange Diagram. Others scored not as good as expected. The main example of this is the Context View that turned out to be too hard to activate for most users. Observations during the experiment have shown that this was a specific usability issue regarding activation, once the Context View was activated users felt comfortable using it. The experiment resulted in many useful suggestions for improvement. Some of these have been completed in the mean time, others require significant changes to the tool and are documented and will be left for future work. Looking at the task types that the tool could support, we see
8.3 Integration

To ease the task of retrieving model evolution data a special version of MetricView Evolution has been created and tested. This version lacks the graphical user interface and just transforms the UML model, quality model and metric definitions into a metrics report. To test it and collect the first batch of model evolution data the stripped version of the tool has been deployed at one industrial site. The details of this process are described below.

8.3.1 Environment

The models are stored in the ClearCase version control system, part of the IBM Rational tool set. The automated build process is managed with two different versions of CruiseControl. The original version is used to manage J2EE (java) projects while CruiseControl.NET (a port for the .NET platform) is used to manage .NET projects. For the integration of MetricView Evolution the fact that there are two different versions is not that relevant, because the integration is at model level, not at source-code level.

What is highly relevant, is the dataformat that is used to store the models in ClearCase. At the moment the Rational XDE is the default application used to create

![Figure 8.4: Context View usability rating](image)

that most participants considered the tool useful for program understanding, quality evaluation and maturity/completeness evaluation. Testing received a much lower rating, which was to be expected because the tool was not developed specifically with this type of task in mind.
and maintain the models. XDE is able to export a model to XMI-1.x that is supported by MetricView. The only catch is that Diagram information is not exported. This severely hampers the use of MetricView to evaluate these models. In the near future the development of models will be done in Rational Software Architect (RSA). The impact of this on the integration of MetricView Evolution is expected to be severe. RSA uses the XMI/UML-2.0 format to store models. This format and version of the UML MetaModel is not yet supported by MetricView. Writing a new parser for the XMI-2.0 format should not be too much of a problem, but integrating the UML2.0 MetaModel is a huge operation having a impact on parts of the system that are tightly coupled.

8.3.2 Adaptation

Because the XMI exported by Rational XDE lacks diagram information and the native format used by XDE (MDX) is also xml based, the decision was made to add a new parser to MetricView. Because such a parser consists of a set of XSL files external to the application the change impact is low. The main problem is the difference in the way geometric information is stored in XMI versus MDX. This is solved by transforming the geometry during the parsing using XSL. The simplest and most effective way to integrate MetricView Evolution into the automated build process is as a command-line tool. The input of this tool is a collection of XML files that represent a model. A CruiseControl script retrieves a snapshot of these files out of ClearCase and runs a merge script to convert the XDE project into a single XML file that is easier to parse. After merging MetricView is called. MetricView calculates its metrics over the model and outputs the results into an XML-file. This output can later be stored in a database as a history of the quality of the model. The output can also be used by CruiseControl to show the results of the analysis on the dashboard (a web-application used to monitor the quality of the builds). As part of the validation of the MetricView project the output is also directed to a e-mail address at the university where the data is stored and analyzed.

8.3.3 Results

Directly after the integration was finished some test runs were performed. Merging, metric calculation and output formatting all worked fine. The metrics report in CruiseControl was updated and mail was sent to the specified receiver. However, shortly afterwards the build server on which everything was implemented went down for unknown reasons. After the server was back on-line the metric analysis build step kept failing. These problems unfortunately could not be solved in time. Additionally the project that was chosen to be monitored was terminated also preventing us from collecting any substantial evolution data.
Chapter 9

Evaluation and Conclusions

9.1 Evaluation

9.1.1 Activities

The project had four central activities:

- Visualization research. During the prototyping phase many ideas were tried out by creating and adapting small prototypes. Depending on the kind of idea that was to be implemented, a single day, a few days or a week was allocated for prototyping. The results of this were used as inspiration for the next prototyping cycles. This was certainly one of the most fun parts of the project.

- Tool development. The extension and partially redesign of MetricView into MetricView Evolution. Roughly half of the time of this project has been spent on development.

- Case studies. Both with and without MetricView Evolution. The first case studies were performed to get a feel for the size and nature of typical cases and what data was desired as output of such an evaluation. In later case studies, the tool was applied and fine-tuned.

- Validation experiment. At the end of the project a small experiment has been performed to validate the tool for usability. Although it was quite some work to organize it, looking back it was definitely worth it. It resulted in some really useful suggestions for improvement and confirmed several ideas that we tried out in the tool.

9.1.2 Lessons learned

Some important lessons that were learned during the aforementioned activities are:

- A simple usability test held among some coworkers can result in a lot of problem reports and change requests. One should be cautious about making assumptions
about the way a person will use a tool. Users are more creative than one might think and this ‘abuse’ can lead to both detection of well-hidden bugs and interesting suggestions for improvement. In this case the test was performed at the very end of the project. It would have been better to do tests multiple times starting at a much earlier phase of the project. This would have required some time but in my opinion this time would be well spent.

- The importance of unit tests and related problems when refactoring software. Because of the nature of the development process a lot of refactoring has been taking place. Normally unit-testing can provide a great level of confidence when making changes to software. The problem with this in relation to refactoring occurs when interfaces are changed. The unit-tests that are using these interfaces have to be changed themselves and thereby become part of the problem instead of the solution. The effects of this can of course be minimized by keeping the interfaces stable, but sometimes changes are unavoidable.

- Some of the features in the tool were developed with the idea that the UML models to be analyzed would be more or less complete. This was an unrealistic expectation. Due to the purpose of the UML models, time available and guidelines within a project most UML models were to some degree incomplete. This is not meant to criticize these models or their creators because the reasons for incompleteness are understandable. The only problem is that the usefulness of some features in the tool is reduced when completeness is low. An example of this is the visualization of inter-diagram relations, which can in theory be used for tracing from the use-case level down to the level of state-diagrams. In practice often either some of the relations are implicit or some diagram type is missing in between.

9.1.3 Contributions

During this project the following contributions have been made to the field of software engineering:

- The implementation of an extensible metrics visualization and extraction tool for monitoring UML models.

- The first implementation of the Lange Diagram in a tool. We believe that this will lead to a better understanding of software designs/architectures. The benefits are fast navigation allowing the user to make a natural mental map of the system, thereby making it easier to find specific information. Together with the GIS approach to metric visualization it also makes it easier to spot potential problems in a model. The small usability experiment indicated that these beliefs are justified.

- The implementation of a quality model in a UML analysis tool. This enables the evaluation of the quality of a model at multiple abstraction levels. Although implementations of Quality Models exist (see [KWS9]) this is the first time we
know of it has been applied at a software design level instead of implementation level.

- Several case studies have been performed, in which defects have been found and practitioners in the field have been made aware of how they could improve the quality of their models using metrics and metrics tooling.

- The tooling has been integrated into a production environment and can be used to automatically retrieve evolution data about the characteristics of software architectures. This will help in future research to obtain the needed data sets.

9.2 Future Work

Some suggestions for future work include:

- The QualityTree has been tested with several small quality models. A reference quality model for UML models would not only give a better view on the real usability of the QualityTree but also be very helpful for users of MetricView Evolution. It would give them a base model to tailor into a quality model for their specific needs.

- The current mechanism for outlier detection together with the context view enables users to effectively detect potential problems in their models and the causes for these problems. As already stated in chapter 8 an extension to this mechanism would be to suggest fixes for these problems.

- Some metrics are defined in terms of other metrics. These derived metrics are currently not supported in the QualityTree and are believed to be a useful extension.

- Another point of interest for future work is the layouting of the Lange Diagram. Different layouts could be created that are targeted at emphasizing specific properties of the Lange Diagram.

- The original Lange Diagram ranges from requirements down to implementation. Requirements traditionally have priorities and are associated with use cases. If the use cases in the tool could be marked with these priorities the inter-diagram relations could be used to derive priorities for other model elements. This could then in turn be used to help in ranking potential defects that are found.

- The support for multiple versions of a tool could be exploited to a higher level by adding prediction capabilities.

- At the moment multiple versions of a model have to be loaded into the tool by hand. Although some automation is available (for loading all files in a directory structure) it would be much nicer if integration with version control systems would be added. This was already stated in the requirements of the tool but not implemented due to time constraints and priorities.
• A final recommendation for future work is using the tool in experiments to further validate its usefulness and obtain ideas for extension and improvement. This could also help in establishing the quality model for UML that was mentioned.

9.3 Conclusions

In the introduction chapter of this document the main goal of this project was defined as the development of a tool that supports the use of metrics to monitor the quality of evolving software architectures. An existing tool called MetricView has been extended with this goal in mind and the resulting application has been applied to a number of cases and demonstrated to several practitioners in the field. These practitioners offered useful comments and suggestions to improve the tooling to suit their needs. During the analysis of the cases the tool has been shown to provide added value by automating tedious tasks to a large extent and offering visualization options that were not possible before. The experiment conducted to validate the usability of the tool showed that participants found the tool useful for program understanding, quality evaluation and maturity/completeness evaluation. Although there is room for improvement with respect to usability, this experiment confirms our beliefs that MetricView Evolution is a useful contribution. These beliefs were further confirmed during demonstrations of the tool, which resulted in enthusiastic responses about its capabilities and potential. We received several requests to make it publicly available. By doing this we hope more people will use metrics to systematically improve the quality of their UML models.
Bibliography


[KLS95] John Krogstie, Odd Ivar Lindland, and Guttorm Sindre. Towards a deeper
understanding of quality in requirements engineering. In CAiSE ’95: Proceed-
ings of the 7th International Conference on Advanced Information Systems

[KW89] Barbara A. Kitchenham and John G. Walker. A quantitative approach to

[Lan03] Christian Lange. Empirical investigations in software architecture complete-

[LC05] Christian Lange and Michel Chaudron. Combining metrics data and the
structure of uml models using gis visualization approaches. In ITCC ’05: Pro-
cedings of the International Conference on Information Technology:
Coding and Computing (ITCC’05) - Volume II, pages 322–326, Wash-

1969.

[LN00] Jukka Paakki A. Inkeri Verkamo Lilli Nenonen, Juha Gustafsson. Measuring
object-oriented software architectures from uml diagrams. In Proceedings
of the 4th International ECOOP Workshop on Quantitative Approaches in
Object-Oriented Software Engineering, pages 87–100, June 2000.

Metrics and laws of software evolution - the nineties view. In METRICS
’97: Proceedings of the 4th International Symposium on Software Metrics,

2nd international conference on Software engineering, page 407, Los Alamitos,


2002.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions
of uml diagrams. In ESEC/FSE-11: Proceedings of the 9th European soft-
ware engineering conference held jointly with 11th ACM SIGSOFT inter-
national symposium on Foundations of software engineering, pages 227–236,


Appendix A

User Manual

A.1 Introduction

To get started with MetricView Evolution you first need to install Python 2.4. This library can be obtained from www.python.org. After it is installed you can start the tool by double-clicking on ‘MetricView Evolution.exe’. The first time the embedded database will have to be initialized, which can take some time.

A.2 Screen Layout

The Menubar is located at the top of the screen as usual, just below it is the Toolbar. Below the toolbar we find the Canvas on the left-hand side and the Metric Visualization Properties panel on the right. Even further down the Timeline is positioned. Below the Timeline there is a Statusbar.

A.3 Menubar

The Menubar consists of three menus. The File menu, the Debug menu and the Help menu.

A.3.1 File

The File menu contains the following entries

- Open UML Model. Loads a model from file. After loading, metrics will be calculated. Watch the Statusbar to see when this task is finished.

- Import Directory. Loads all supported files from a directory tree. N.B. This feature is not actively supported at the moment.

- Close Model. Removes the currently active model from memory. To see which model is active use the Timeline.
Generate metrics report. Generate and save XML file containing the metrics data for the version of the model that is active.

Open settings. Loads preferences from a file.

Save settings as. Saves current preferences to a file.

Export Canvas. Save a screenshot of the Canvas to a .png file.

Exit. Shuts down the application

A.3.2 Debug

The Debug menu contains only one item: SQL Debug Dialog. This opens a dialog that provides direct access to the embedded database in which the models and metrics are stored. Use with care!

A.3.3 Help

The Help menu contains only one item: About. This shows a dialog with some credits.

A.4 Toolbar

The Toolbar is divided into 7 sections: File, Filter, Zoom, Windows, Preferences, Search, Clustering

A.4.1 File

The File section contains two buttons: one to open a file and one to close the currently active model.

A.4.2 Filter

The Filter section contains a combobox to select the type of filter and a text field to provide a filter parameter. This parameter can e.g. be used with the ‘name’ filter to filter on a specific name.

A.4.3 Zoom

The Zoom region contains only one button. This button zooms out the view in the active window to fit all contents.

A.4.4 Windows

The Windows region contains three buttons. From left to right they are used to: split the active window horizontally, split the active window vertically and remove the active window.
A.4.5 Preferences
The Preferences section contains only one button that can be used to open the preferences dialog. This dialog shows a tree with all available global options. The change an option, select it in the tree and make modifications in the panel right of the tree.

A.4.6 Search
The Search section contains a text field and three buttons. The text field can be used to search in the Lange-Diagram. Searching will be performed when typing and the results will be highlighted as if they were selected. The two green buttons next to the search field are used to navigate to the search results by zooming in, previous and next result respectively. The red button is used to clear the search field and undo the navigation.

A.4.7 Clustering
The clustering section contains three sliders that can be used to manipulate the layout of the Lange-Diagram. Changing the leftmost slider will activate a force-directed layout and clustering view on the lange diagram. The ‘clustering’ slider controls the attractive force between the diagrams, the ‘cooling’ slider is used to restrict the freedom of movement of the diagrams, the ‘iterations’ slider is used to control the number of iterations is used during the layout calculation. On slower computers this should not be too many.

A.5 Canvas
The Canvas is shown as a grey rectangle when MetricView Evolution is started. This is to indicate that no visualization is active. Visualizations can be activated inside windows. Initially the Canvas contains one window. To create more windows use the buttons in the Windows section of the toolbar. Windows can be resized by dragging their borders. Only one window is active at a given time (indicated by a red border), to active a window click on it once. To enable a visualization inside a Window use on of the following keys:

- F2. Activates the Model Visualization showing an overview of UML diagrams in a Lange-Diagram.
- F3. Activates the QualityTree Visualization, showing the quality model that is used to evaluate the quality of the UML model.
- F4. Activates the Context Visualization, showing the context of a model element with respect to a metric.

In the lower righthand corner of each window a single character indicates the visualization that is active (M, C, Q). Alternatively the combobox in the Visualization Properties panel can be used to select a visualization.
A.5.1 Model

The model visualization show an overview of 4 types of UML diagrams in a Lange-Diagram. If the normal layout is active (that is, the ‘clustering’ slider in the toolbar is in its default position) the diagrams are grouped by type:

- UseCase Diagrams. Surrounded by an ellipse.
- Message Sequence Diagrams. Surrounded by a rectangle with one triangular side.
- Class Diagrams. Surrounded by a rectangle with one triangular side an black ‘ears’.
- State Diagrams. Surrounded by a rounded rectangle.

The Lange-Diagram allows visualization of inter-diagram relations. To enable drawing of these relations select an element in one of the diagrams. If relations to elements in other diagrams are available they will be shown by black splines. When force-directed layout is active, these relations are grouped by diagram to show the attracting forces.

To navigate use the mouse. Panning is done by dragging with the middle mouse button; rotating by holding SHIFT while dragging with the middle mouse button. To zoom, use the scroll wheel. Notice that scrolling with the mouse cursor over some element will cause the view to ‘lock’ onto this element. For quick navigation in the Lange-Diagram focussing can be used. To focus on something, hold down the ALT button while clicking on an element. This will trigger zooming and panning and places the focus on that element. To unfocus and return to the overview, press the ESC button. When you have loaded multiple versions of a model, navigating between these versions can be done by holding the ALT button while rotating the scroll wheel.

QualityTree

The QualityTree is a quality model. It consists of 2 types of nodes and edges. The first node type is the metric. A metric is visualized by a blue rectangle. Metrics can usually be found on the right-hand side of the QualityTree. The value of a Metric is directly calculated from the UML model. The other node type is the Quality Attribute. Quality Attributes are abstractions from Metrics or other Quality Attributes. The value of a Quality Attribute is calculated by applying a Quality Function on the nodes connected to the incoming edges. Nodes in the QualityTree can be selected. This will show upstream and downstream dependencies.

A.5.2 Context

The Context visualization shows a context diagram. A context diagram contains all relevant information from the model with respect to a metric and one model element. For this to work you need to place the focus on some model element (see A.5.1) and enable a metric in the Metric Visualization Properties panel (see A.6).
A.6 Metric Visualization Properties

This panel is used to configure the visualization properties of the metrics in the Model Visualization. The panel is divided into 3 parts. The upper part controls the layout, the middle part which metrics are active and the lower part controls the specific visualizer options for each metric. When a metric is activated this enables a metric visualization for that metric. Currently a bug prevents the Canvas from correctly redrawing afterwards, to see the metric visualization click somewhere in the Model Visualization.

A.7 Timeline

The Timeline fulfills two functions. First it shows which versions of a model are loaded and which is active. Each version is denoted by a vertical line at the position related to the date of version. The active version is marked by a yellow color. When a Metric or Quality Attribute is selected in the QualityTree the Timeline visualizes the value of the selected node. This gets interesting when multiple versions of a model are loaded because then Timeline makes it possible to spot trends in quality changes.

A.8 Statusbar

The Statusbar is mainly used to indicate the progress of loading a file and calculating metrics.

A.9 Tutorial

Open the files ‘TLCS - older.xml’, ‘TLCS - high coupling.xml’ and ‘TLCS - original.xml’ that were supplied with the Tool by using the File menu or the Toolbar. Wait for each loading action to finish before starting the next. If everything went correctly you should see two black lines in the Timeline and one yellow one. Click on the (grey) canvas and press F2. If necessary, resize the canvas by dragging around its borders. You should now see a Lange-Diagram of the TrafficLight Control System.

Hover over the diagrams and elements until you find a class with the name ‘Crossing’ and click on it. This should render the class yellow and trigger 5 relations to become active. Focus on one of the Sequence Diagrams by holding down ALT and clicking on the diagram. The incoming relation from the class diagram should point to an object of type ‘Crossing’. When finished, press ESC to return to the overview.

The Metric Visualization Properties panel on the right should contain a set of metrics. Click on the white toggle box next to the metric ‘cbo1’. This should trigger Metric Visualization to become active. (Due to a bug this might not be the case, click somewhere in the Lange-Diagram to force it to become active).

Click on the ‘Split Window Vertically’ button in the Toolbar and click once on the window that appears. Press F4 to select the QualityTree. Click on the ‘Split Window Horizontally’ afterwards and activate this window by clicking on it. Press F3 to select
the Context Visualization. Now activate the Model Window again and focus on the ‘Crossing’ class by holding down ALT and clicking on it. This should trigger the Context Visualization to show a context diagram. Resize the windows to your liking.

Activate the Model Window and press ESC to return to the overview. Activate the QualityTree Window and select the metric ‘Coupling between objects’ (second one from above, on the right). Resize the Timeline panel by dragging the splitter. This should cause a graph to appear showing the changes in coupling between the different versions of the model.
Appendix B

Additional Tooling

B.1 Quality Tree Editor

B.1.1 Motivation
The MetricView Evolution QualityTree is defined in XML. This makes it possible to define such quality models using nothing more than a text editor. For small models this is perfectly possible but for larger models something less time consuming is required. The Quality Tree Editor helps making definition of QualityTrees fast and painless.

B.1.2 Requirements

B.1.3 Design
QualityTreeEditor is designed around the Model-View-Presenter design pattern. This pattern is an adaptation of the Model-View-Controller pattern. This section will explain how this pattern works, why it was chosen and how it used in the design of QualityTreeEditor.

Structure
Because the structure of the MVP pattern is based on the MVC pattern let’s first recall the structure of this classic pattern.

As the name suggest the MVC pattern consists of three main classes. The Model that is responsible for the data storage, the view that visualizes this data and the controller that manages the interactions from the view to the model. The triangle is closed by an observer pattern between de model and the view that enables the model to notify the view of changes.

The MVP pattern is an evolution of the MVC pattern in a sense that the main structure hasn’t changed. The thing that has changed is the way the view communicates with the controller and the way the controller communicates with the model. This communication has been generalized by defining three more classes that encapsulate the communication principles:
Figure B.1: The Model View Controller pattern

Figure B.2: The Model View Presenter pattern
B.1. QUALITY TREE EDITOR

- Interaction
- Selection
- Command

The first class, Interaction, is used for communication from the View to the Presenter. It defines a translation from events that happen in the user interface to the type of changes in the data of the Model.

The last two classes are used for communication from the Presenter to the Model and translate the type of changes into actual operations that need to be performed.

Another difference between the MVC and the MVP pattern is that the Controller class is replaced by the Presenter class. The Presenter class handles the communication from the View class to the Model class. Because of the central position of this class it is the ideal candidate to perform tasks like keeping track of the current selection, keeping track of performed actions for undo management, and logging communication between the View class and Model class.

Parameters

Following the advice of [Pot96] the MVP pattern is best used by first answering 6 questions about the nature of the application. These questions are:

1. What is my data? (Model)
2. How do I specify my data? (Selection)
3. How do I change my data? (Command)
4. How do I display my data? (View)
5. How do events map onto changes in my data? (Interaction)
6. How do I put it all together? (Presenter)

What is my data? The first question is related to the Model class of the pattern. In the case of QualityTreeEditor the data consists of the Quality Tree. Unlike the name suggests, this Quality Tree is not really a tree but rather a directed acyclic graph. The graph has two types of nodes: metrics and quality attributes. Both type of nodes have a name. The metrics only have outgoing connections and contain a reference to the actual metric that they represent. The quality attributes can have both incoming and outgoing connections and contain a reference to a function that converts incoming data to outgoing data. For clarity of presentation the Quality Tree is split into one or more levels. These levels are used to group similar nodes and can have a name. Each of the nodes is mapped onto exactly one level, a level can have many nodes.
How do I specify and select my data? The second question is related to the Selection class of the pattern. In answering the first question we encountered several entities:

- Level
- Metric Node
- Quality Attribute Node
- Connection
- Metric Reference
- Function Reference

Levels, Nodes and References have a name attribute by which they could be selected. The problem is that these names can be changed by the user and might therefor not be unique. The common solution to this problem is to introduce unique identifiers. Element selection is done using these identifiers.

How do I change my data? The third question is related to the Command class of the pattern. Answering it will reveal the different Command subclasses that are needed. The basic operations that can be performed on the data are:

- Add
- Change attribute (e.g. name, reference, position)
- Remove

How do I display my data? This question is related to the View class of the pattern. The visualization of the Quality Tree is visible in figure 13.3. Nodes are displayed as boxes with a different color depending on the type. Connections between nodes are depicted by arrows. The levels are displayed as large rectangles on which the nodes can be placed. The names of the levels and nodes are printed on top of these elements.

How do events map onto changes in my data? This question is related to the Interaction class of the pattern. The purpose of this class is to translate gestures in the user interface into meaningful operations on the data. To answer this question the first thing that needs to be determined is the set of available gestures in the user interface and their semantics.

1. Drag and drop a stencil on the Canvas or Level. This is the gesture for adding the element that the stencil represents (Node or Level) to the Quality Tree.
Figure B.3: A screenshot of a prototype of QualityTreeEditor
2. Drag and drop an element from one position to another. This is the gesture for changing the position of the element within the Quality Tree. If a node is moved from one Level to another it also changes the mapping from the node to the level.

3. Drag and drop to the trashcan. This is the gesture for removing elements. If a level is removed it also results in the removal of all elements on that level.

4. Double Click. This is the gesture for changing the name of an element (Node or Level).

5. Shift+Double Click. This is the gesture for changing the function reference of a node.

6. Alt+Double Click. This is the gesture for changing the metric reference of a metric node.

7. Shift+Drag and drop. This is the gesture for adding a connection between two nodes.

8. Ctrl+Drag and drop. This is the gesture for removing a connection between two nodes.

**How do I put it all together?** The final question is related to the Presenter class of the pattern. This class translates interactions into commands in selections; a translation that needs to be defined. Because of the selections the presenter can be stateful.

**Advantages of the MVP pattern** Some noticeable advantages of using the MVP pattern are.

- Replaceable views. Because there is a clear distinction between the model and views on this model it takes less effort to replace a view with a better one at a later point in time.

- Automated user interface testing. The fact that the translation of user interaction to actions is performed at one central point, the presenter, makes it possible to define unit tests for the user interface.

- Commands can be queued, easy undo integration. The command pattern used in MVP enables persistence of commands. This can be used to implement command queueing, executing commands remotely (e.g. view and model separated over network) and eases the implementation of undo management. For this last part it is necessary that the commands can undo their actions.

**B.1.4 Implementation Notes**

Below are a few notes about parts of the implementation that can be tricky to understand and are in general good candidates for refactoring. These notes are also available in the code by searching for ‘HACK’.
B.1. QUALITY TREE EDITOR

**ViewLevel**

The private attribute \_insertIndex is used to store the point at which the last node that was dropped on the level. The is done because the execution flow of the drop event runs through the presenter and the model before it finally arrives at the ViewLevel class again. It would be possible to store and pass this index through all calls that are made during the handling of the event. Because of the fact that this data is used only locally in ViewLevel the decision was made to use a private attribute.

**QualityTreeView**

The same trick with a private attribute \_insertIndex is used as with ViewLevel.

**QualityTreePresenter**

The method flushCommandQueue is needed because some commands do not allow immediate execution. An example of this is dragging and dropping a node to the trashcan. If the command were to be executed immediately, the element from which the event originates would be deleted during the handling of the event. To get around this problem flushCommandQueue is called when the mouse is moved over the trashcan without holding down a mouse-button.
Appendix C

Validation Results

C.1 Usefulness

Table C.1 shows the summed usefulness results. Table C.2 shows the percentage of users that marked a particular feature as useful for performing a specific task type correctly. Table C.3 shows the percentage of users that marked a particular feature as useful for performing a specific task type efficiently. The different task types are:

1. Program understanding
2. Development
3. Testing
4. Maintenance
5. Quality evaluation
6. Maturity/Completeness evaluation

C.2 Understandability

Table C.4 shows the percentage of users that gave a specific feature a particular understandability rating. Table C.5 shows the percentage of users that gave a specific feature a particular rating for how intuitive the feature is.
## APPENDIX C. VALIDATION RESULTS

<table>
<thead>
<tr>
<th>Feature</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lange Diagram</td>
<td>32</td>
<td>29</td>
</tr>
<tr>
<td>Quality Tree</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Context View</td>
<td>28</td>
<td>25</td>
</tr>
<tr>
<td>Timeline</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>Metric View</td>
<td>22</td>
<td>21</td>
</tr>
<tr>
<td>Search and Highlight</td>
<td>33</td>
<td>35</td>
</tr>
<tr>
<td>Evolution of a Metric</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>Design Smells</td>
<td>25</td>
<td>27</td>
</tr>
<tr>
<td>Filtering</td>
<td>21</td>
<td>17</td>
</tr>
<tr>
<td>Clustering</td>
<td>16</td>
<td>19</td>
</tr>
</tbody>
</table>

Table C.1: Summed usefulness results. C: correctly doing a task, F: doing a task fast

<table>
<thead>
<tr>
<th>Feature</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lange Diagram</td>
<td>83.33%</td>
<td>58.33%</td>
<td>25.00%</td>
<td>41.67%</td>
<td>25.00%</td>
<td>33.33%</td>
</tr>
<tr>
<td>Quality Tree</td>
<td>16.67%</td>
<td>16.67%</td>
<td>16.67%</td>
<td>33.33%</td>
<td>83.33%</td>
<td>41.67%</td>
</tr>
<tr>
<td>Context View</td>
<td>75.00%</td>
<td>58.33%</td>
<td>25.00%</td>
<td>33.33%</td>
<td>25.00%</td>
<td>16.67%</td>
</tr>
<tr>
<td>Timeline</td>
<td>25.00%</td>
<td>25.00%</td>
<td>16.67%</td>
<td>50.00%</td>
<td>58.33%</td>
<td>41.67%</td>
</tr>
<tr>
<td>Metric View</td>
<td>41.67%</td>
<td>16.67%</td>
<td>0.00%</td>
<td>25.00%</td>
<td>58.33%</td>
<td>41.67%</td>
</tr>
<tr>
<td>Search and Highlight</td>
<td>66.67%</td>
<td>58.33%</td>
<td>16.67%</td>
<td>50.00%</td>
<td>41.67%</td>
<td>41.67%</td>
</tr>
<tr>
<td>Evolution of a Metric</td>
<td>16.67%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>8.33%</td>
<td>66.67%</td>
<td>33.33%</td>
</tr>
<tr>
<td>Design Smells</td>
<td>33.33%</td>
<td>50.00%</td>
<td>16.67%</td>
<td>25.00%</td>
<td>50.00%</td>
<td>33.33%</td>
</tr>
<tr>
<td>Filtering</td>
<td>50.00%</td>
<td>16.67%</td>
<td>16.67%</td>
<td>41.67%</td>
<td>25.00%</td>
<td>25.00%</td>
</tr>
<tr>
<td>Clustering</td>
<td>66.67%</td>
<td>8.33%</td>
<td>25.00%</td>
<td>33.33%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Table C.2: Usefulness for correctly doing a type of task

<table>
<thead>
<tr>
<th>Feature</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lange Diagram</td>
<td>83.33%</td>
<td>41.67%</td>
<td>16.67%</td>
<td>50.00%</td>
<td>16.67%</td>
<td>33.33%</td>
</tr>
<tr>
<td>Quality Tree</td>
<td>8.33%</td>
<td>16.67%</td>
<td>16.67%</td>
<td>33.33%</td>
<td>83.33%</td>
<td>50.00%</td>
</tr>
<tr>
<td>Context View</td>
<td>66.67%</td>
<td>50.00%</td>
<td>8.33%</td>
<td>33.33%</td>
<td>25.00%</td>
<td>25.00%</td>
</tr>
<tr>
<td>Timeline</td>
<td>25.00%</td>
<td>8.33%</td>
<td>16.67%</td>
<td>50.00%</td>
<td>66.67%</td>
<td>41.67%</td>
</tr>
<tr>
<td>Metric View</td>
<td>41.67%</td>
<td>25.00%</td>
<td>8.33%</td>
<td>25.00%</td>
<td>41.67%</td>
<td>33.33%</td>
</tr>
<tr>
<td>Search and Highlight</td>
<td>83.33%</td>
<td>50.00%</td>
<td>16.67%</td>
<td>50.00%</td>
<td>41.67%</td>
<td>50.00%</td>
</tr>
<tr>
<td>Evolution of a Metric</td>
<td>25.00%</td>
<td>8.33%</td>
<td>0.00%</td>
<td>8.33%</td>
<td>66.67%</td>
<td>41.67%</td>
</tr>
<tr>
<td>Design Smells</td>
<td>16.67%</td>
<td>41.67%</td>
<td>25.00%</td>
<td>33.33%</td>
<td>66.67%</td>
<td>41.67%</td>
</tr>
<tr>
<td>Filtering</td>
<td>50.00%</td>
<td>0.00%</td>
<td>25.00%</td>
<td>16.67%</td>
<td>25.00%</td>
<td>25.00%</td>
</tr>
<tr>
<td>Clustering</td>
<td>66.67%</td>
<td>8.33%</td>
<td>16.67%</td>
<td>25.00%</td>
<td>25.00%</td>
<td>16.67%</td>
</tr>
</tbody>
</table>

Table C.3: Usefulness for efficiently doing a type of task
### C.2. UNDERSTANDABILITY

<table>
<thead>
<tr>
<th>Feature</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lange Diagram</td>
<td>0,00%</td>
<td>0,00%</td>
<td>30,77%</td>
<td>23,08%</td>
<td>46,15%</td>
</tr>
<tr>
<td>Quality Tree</td>
<td>0,00%</td>
<td>0,00%</td>
<td>25,00%</td>
<td>16,67%</td>
<td>58,33%</td>
</tr>
<tr>
<td>Context View</td>
<td>0,00%</td>
<td>23,08%</td>
<td>38,46%</td>
<td>23,08%</td>
<td>15,38%</td>
</tr>
<tr>
<td>Timeline</td>
<td>0,00%</td>
<td>0,00%</td>
<td>7,69%</td>
<td>53,85%</td>
<td>38,46%</td>
</tr>
<tr>
<td>MetricView</td>
<td>0,00%</td>
<td>0,00%</td>
<td>23,08%</td>
<td>53,85%</td>
<td>23,08%</td>
</tr>
<tr>
<td>Search and Highlight</td>
<td>0,00%</td>
<td>0,00%</td>
<td>23,08%</td>
<td>23,08%</td>
<td>53,85%</td>
</tr>
<tr>
<td>Evolution of a Metric</td>
<td>0,00%</td>
<td>0,00%</td>
<td>25,00%</td>
<td>25,00%</td>
<td>50,00%</td>
</tr>
<tr>
<td>Design Smells</td>
<td>0,00%</td>
<td>46,15%</td>
<td>15,38%</td>
<td>30,77%</td>
<td>7,69%</td>
</tr>
<tr>
<td>Filtering</td>
<td>8,33%</td>
<td>16,67%</td>
<td>33,33%</td>
<td>25,00%</td>
<td>16,67%</td>
</tr>
<tr>
<td>Clustering</td>
<td>16,67%</td>
<td>0,00%</td>
<td>41,67%</td>
<td>25,00%</td>
<td>16,67%</td>
</tr>
</tbody>
</table>

Table C.4: Understandability ratings.

<table>
<thead>
<tr>
<th>Feature</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lange Diagram</td>
<td>0,00%</td>
<td>0,00%</td>
<td>23,08%</td>
<td>46,15%</td>
<td>30,77%</td>
</tr>
<tr>
<td>Quality Tree</td>
<td>0,00%</td>
<td>0,00%</td>
<td>0,00%</td>
<td>41,67%</td>
<td>58,33%</td>
</tr>
<tr>
<td>Context View</td>
<td>0,00%</td>
<td>30,77%</td>
<td>53,85%</td>
<td>7,69%</td>
<td>7,69%</td>
</tr>
<tr>
<td>Timeline</td>
<td>0,00%</td>
<td>7,69%</td>
<td>0,00%</td>
<td>46,15%</td>
<td>46,15%</td>
</tr>
<tr>
<td>MetricView</td>
<td>0,00%</td>
<td>0,00%</td>
<td>23,08%</td>
<td>30,77%</td>
<td>46,15%</td>
</tr>
<tr>
<td>Search and Highlight</td>
<td>0,00%</td>
<td>0,00%</td>
<td>0,00%</td>
<td>38,46%</td>
<td>61,54%</td>
</tr>
<tr>
<td>Evolution of a Metric</td>
<td>0,00%</td>
<td>0,00%</td>
<td>16,67%</td>
<td>41,67%</td>
<td>41,67%</td>
</tr>
<tr>
<td>Design Smells</td>
<td>0,00%</td>
<td>15,38%</td>
<td>46,15%</td>
<td>23,08%</td>
<td>15,38%</td>
</tr>
<tr>
<td>Filtering</td>
<td>0,00%</td>
<td>16,67%</td>
<td>33,33%</td>
<td>33,33%</td>
<td>16,67%</td>
</tr>
<tr>
<td>Clustering</td>
<td>0,00%</td>
<td>8,33%</td>
<td>50,00%</td>
<td>33,33%</td>
<td>8,33%</td>
</tr>
</tbody>
</table>

Table C.5: Intuitivity rating