Integration of a Standard-Based Quality Assessment into the VizzAnalyzer

David Ruiz de Azua
Bachelor Thesis

Integration of a Standard-Based Quality Assessment into the VizzAnalyzer

David Ruiz de Azua Nieto

31st of May 2006

Department of Computer Science
Växjö University

Supervisors:
Prof. Dr. Welf Löwe
Rüdiger Lincke, PhD Student
Abstract
More than half of the total costs in ownership of a software system are maintenance costs. Reverse engineering is becoming more important and complex for huge systems, and tools for reverse engineering are necessary for system evaluation.

The ISO/IEC 9126 standard defines software quality and The VizzAnalyzer Framework is a stand-alone tool for analyzing and visualizing large software systems’ structures.

In this thesis, we describe the design and implementation of plug-ins for the VizzAnalyzer Framework, a system for reverse engineering, extending their features under the standards of software quality. It has proven to be useful in applying the new features into the VizzAnalyzer Framework being the first tool that includes a software quality model.

Keywords: reverse engineering, software systems, framework, software quality
# Table of Contents

Abstract ....................................................................................................................... 5  
Table of Contents ........................................................................................................ 7  
List of Figures ............................................................................................................ 9  
List of Tables .......................................................................................................... 11  
Glossary .................................................................................................................. 13  
1 Introduction ........................................................................................................ 15  
  1.1 Context of the thesis ................................................................................... 15  
  1.2 Problem ...................................................................................................... 15  
  1.3 Goals and Criteria ...................................................................................... 16  
  1.4 Motivation .................................................................................................. 17  
  1.5 Outline ........................................................................................................ 17  
2 Background ........................................................................................................ 19  
3 State of the Art ................................................................................................... 21  
4 Requirements ..................................................................................................... 23  
  4.1 Technical aspects of the problem ............................................................... 23  
  4.2 Define the system ....................................................................................... 23  
  4.3 Users ........................................................................................................... 24  
  4.4 Features ...................................................................................................... 24  
  4.5 Use Cases ................................................................................................... 24  
    4.5.1 Use Case model .................................................................................. 24  
    4.5.2 Use Cases ........................................................................................... 25  
  4.6 Functional requirements ............................................................................. 29  
5 Architecture ........................................................................................................ 33  
  5.1 VizzAnalyzer architecture .......................................................................... 33  
  5.2 VizzAnalyzer plug-ins architecture ........................................................... 33  
  5.3 Overall architecture for the solution .......................................................... 34  
  5.4 The Quality Model XML file ..................................................................... 35  
6 Design and implementation ............................................................................... 39  
  6.1 Metrics component (script) ........................................................................ 39  
  6.2 Software Quality Model component (plug-in) ........................................... 40  
    6.2.1 Software Quality Model plug-in ........................................................ 40  
    6.2.2 Data Structure .................................................................................... 43  
  6.3 Statistics Plug-in ........................................................................................ 45  
7 Conclusion and Future Work ............................................................................. 49  
  7.1 Conclusions ................................................................................................ 49  
  7.2 Future work ................................................................................................ 50  
Appendix A User Manual ......................................................................................... 51  
  A.1 Installation .................................................................................................. 51  
  A.2 Evaluation of a Software Quality Model .................................................. 51  
    A.2.1 Fulfilling the preconditions: Analyzing a Software System .......... 51  
    A.2.2 Applying Metrics (Metrics script) ...................................................... 52  
    A.2.3 Evaluating the Software Quality Model (Quality Model Plug-in) ... 53  
    A.2.4 Applying Equivalence Classes (preparing statistics) ..................... 54  
    A.2.5 Calculating the statistics .................................................................. 56  
References ................................................................................................................. 59
List of Figures

Figure 2.1: Quality Software Matrix representation .................................................. 19
Figure 4.1: Use Case Diagram .................................................................................. 25
Figure 5.1: VizzAnalyzer architecture ...................................................................... 33
Figure 5.2: Analyzer Framework Design .................................................................. 34
Figure 5.3: Pipeline Architecture ............................................................................ 35
Figure 5.4: Quality Model DTD .............................................................................. 36
Figure 5.5: QMXL example .................................................................................... 38
Figure 6.1: Metrics component script (NOC, DIT, LOC) ......................................... 39
Figure 6.2: Class Diagram SQM plug-in component .............................................. 40
Figure 6.3: Sequence diagram Software Quality Model evaluation ....................... 42
Figure 6.4: Data Structure Design .......................................................................... 43
Figure 6.5: Sequence Diagram data structure creation and evaluation .................... 44
Figure 6.6: Data Structure example ........................................................................ 45
Figure 6.7: Class Diagram Statistics component plug-in ....................................... 46
Figure 6.8: Sequence diagram Statistics plug-in calculation .................................. 47
Figure A.1: VizzAnalyzer Program ........................................................................ 51
Figure A.2: Loading Graphs .................................................................................... 52
Figure A.3: Metrics Result ...................................................................................... 53
Figure A.4: Quality Model Plug-in Execution ....................................................... 53
Figure A.5: QM Selection QMXML file .................................................................. 54
Figure A.6: QM Result ........................................................................................... 54
Figure A.7: CrocoPat Configuration ....................................................................... 55
Figure A.8: CrocoPat Mark Results ........................................................................ 55
Figure A.9: CrocoPat script .................................................................................... 55
Figure A.10: EC Selection ...................................................................................... 56
Figure A.11: Running EC Script ............................................................................. 56
Figure A.12: Statistics Plug-in Execution .............................................................. 57
Figure A.13: Statistics Parameters ......................................................................... 57
List of Tables

Table 3.1: Software Quality Analyzer tools .......................................................... 22
Table 4.1: Features ............................................................................................... 24
Table 4.2: VizzAnalyzer Tool Use Cases ............................................................... 26
Table 4.3: Use Cases ............................................................................................. 29
Table 4.4: Requirements ....................................................................................... 31
## Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC</td>
<td>Equivalence Classes</td>
</tr>
<tr>
<td>FCM</td>
<td>Factor-Criteria-Metric</td>
</tr>
<tr>
<td>GRAIL</td>
<td>GRAPh ImpLementation</td>
</tr>
<tr>
<td>QM</td>
<td>Quality Model</td>
</tr>
<tr>
<td>QMXML</td>
<td>Quality Model XML</td>
</tr>
<tr>
<td>SQM</td>
<td>Software Quality Model</td>
</tr>
<tr>
<td>SQX</td>
<td>Software Quality Matrix</td>
</tr>
<tr>
<td>VAF</td>
<td>VizzAnalyzer Framework</td>
</tr>
</tbody>
</table>
1 Introduction

According to different studies, currently more than half of the total costs in ownership of a software system are maintenance costs [8], [9]. This makes it important to control software qualities like maintainability, usability, and portability directly during software development.

The ISO/IEC 9126 standard [10] defines software quality, which is described as using internal and external software qualities and their connection to attributes of software in a so-called Software Quality Model (SQM). The Software Quality Model defined in ISO 9126 follows the Factor-Criteria-Metric model [11] proposed by McCall in 1977. It defines six quality factors, which are refined into criteria. These criteria are in turn assessed by metrics measuring the design and development process and the software itself. As pointed out by Lincke and Löwe in [1], the measures are currently intended to be performed manually, which creates a series of drawbacks. Among others, they are error-prone, time consuming and might cause tensions in the organizations.

Lincke and Löwe are currently researching a Software Quality Model (SQM) being based on the ISO 9126 standard which makes up for these drawbacks. Since the standard defines internal software quality according to the Factor-Criteria-Metric (FCM) approach, yet does itself not define metrics, being suitable for the automated evaluation of internal software quality upon the static analysis of program code, Lincke and Löwe are currently conducting an experiment with the goal to evaluate well-discussed and validated metrics from literature in order to bring them in relation to the ISO quality model. This instantiation of a SQM shall be evaluated in research and industrial setting requiring a software implementation and tool support.

1.1 Context of the thesis

The VizzAnalyzer Framework is a stand-alone tool for analyzing and visualizing large software systems’ structures. It extracts information from a system’s source code, providing them as GRApIh ImpLementation (GRAIL). Graphs using the GRAIL performs further analyses and allows for the application of various metrics. Further over its plug-in architecture it is extensible and currently provides several visualization tools to display the analysis results and structure of the software systems.

The context of this bachelor thesis is to extend the VizzAnalyzer Framework in order to create the tool support needed for the experiment described in [1].

1.2 Problem

A complex, versatile and non-uniform Software Quality Model is developed and evaluated in a research experiment involving industrial partners. This model presents much information that must be organized, so that it be viewed in its whole and detailed in its single parts. Constraints in respect to flexibility of the content of the SQM and the definition of the application focus and visualization of the results need to be considered.

In general terms, it is necessary to understand:

- which kind of information must be organized
- which kind of structure will have our information
- and how to focus on this information

Creating a data model allowing for the application of the SQM onto a software system in an efficient manner, and thereby being flexible enough to adapt to changes in the SQM is not trivial. Additionally, the amount of data created can be expected to be huge, since the software systems used for evaluation are small to medium size (some
hundred classes and over 100k lines of code). It is necessary to allow for providing overview information in the form of statistical analysis.

VizzAnalyzer Framework is a composition system which supports the rapid construction of reverse engineering tools, allowing measuring quality aspects of software systems [4]. Its current instantiation in the VizzAnalyzer tool provides front-ends for the extraction of quantitative information about software systems provided in source code. Its internal data model can be used for the calculation of metrics. Several metrics have been already implemented. Yet an analysis providing the application of a SQM is not implemented. Statistical information on metrics is only calculated per metric, it is not flexible in its application.

To address these shortcomings and to support the research of Lincke and Löwe, a bachelor thesis with the “Integration of a standard-based Software Quality Assessment into the VizzAnalyzer” is designed to improve on these shortcomings.

Thus, the problem addressed by this thesis is:

“The extension of the VizzAnalyzer tool with suitable analysis allowing the automated evaluation of a flexible Software Quality Model, the automated calculation of the required metrics and the statistical analysis of the results.”

1.3 Goals and Criteria

This section describes the goals pursued by this thesis in order to solve the problem and the criteria used for validating the goals.

- The first goal is the execution of metrics on a Graph representing a software system. The precondition is that this graph contains all relevant information for the calculation of these metrics. The goal is met when, after applying the metrics to the graph, the individual elements have the metric values as properties attached.

- The second goal is to find a suitable format for describing the SQM in a machine and human readable form, so that it can be input to an analysis. This goal is met when all parts of the SQM, including definitions for factors, criteria, metrics, their relations and the weights for them are defined, and it is easily possible to extend the model content with new factors, criteria’s, metrics, relations or weights by editing the definition with a normal text editor.

- The third goal is to evaluate the SQM on a graph and store the resulting information to each node of the graph. The criterion for meeting this goal is that all relevant elements (nodes) in a graph on which the SQM has been applied to and has the resulting information attached in the form of properties. This means that the calculated factor and criterion values are available for further analysis and visualization.

- The last goal is to calculate statistics over the evaluated SQM data to allow for easy comprehension and detection of outliers. This goal is reached when it is possible to apply different statistics methods, including maximum, minimum and average, to all elements of a graph or groups of them.

Other goals besides these are required.

- First, creating a good software design making the plug-ins extendable with new statistical methods, changes in the SQM, and new metrics.

- Secondly, performance is a concern. It is met when the calculation of the metrics, statistics and the SQM is performed in feasible time. This means seconds or minutes to run on a graph rather than hours or days.
The expected outcome of this thesis is the implementation of plug-ins extending the VizzAnalyzer tool with the features needed for the evaluation of a SQM under the presented constraints, reaching the stated goals.

1.4 Motivation
For assessing the internal quality of a software system according to a software quality model, and in particular for conduction the experiment described by Lincke and Löwe in [1], appropriate tool support is mandatory. Without it, it is not possible to test different configurations of a SQM and the impact of different metrics on the measurement of the overall quality in a software system. Repeatability, automation and flexibility are crucial. Since the VizzAnalyzer framework provides all means for analysis software systems including an extendable plug-in architecture, it is natural to extend this tool with the needed features, instead of implementing a complete solution on its own.

1.5 Outline
The structure of this paper thesis is as follows.
Chapter 2 describes the present background and all that it is necessary to understand the context of this thesis.
Chapter 3 describes the most advanced tools for software analysis and calculation of metrics.
Chapter 4 describes the features, use cases, the functional requirements resulting from the goals and criteria stated in Chapter 1.3.
Chapter 5 covers the architecture from VAF and the plug-ins implemented for this thesis.
Chapter 6 describes the design and implementation of the Quality Model Plug-in and the Statistics Plug-in, parts of the system architecture introduced in Chapter 5. Further, it covers the Data Structure designed during the develop process.
Chapter 7 concludes the thesis and describes future work.
Appendix A is a user manual; it contains information for the user using VAF and the plug-ins developed, as well as a demonstration of the current available features.
2 Background

Lincke and Löwe are currently building a knowledge base mapping standard qualities to well-established software metrics and vice versa. Their effort is documented in “The Compendium of Software Quality and Metrics” [12]. They have the goal of providing an information resource which precisely defines their interpretation of the software quality standards, the software metrics and their variants. Furthermore, they propose connections between them.

Currently, their compendium describes
- 37 software quality properties (factors and criteria),
- 14 software quality metrics.

The information contained in the compendium can be summarized in a matrix where the factors, criteria, metrics and their relations representing an instance of the Software Quality Model are displayed. For this thesis it will be referred as Software Quality Matrix (SQX). This is shown in Figure 2.1: Quality Software Matrix representation. The influence between the criteria and metrics is represented with a weight depending on how related the criteria and metrics are.

Metrics are related with criteria, and depending on the metric and criteria they can be highly directly, directly, highly inversely or inversely related. The relationship can be demonstrated by using a weight variable. Highly directly relationship indicates a high quality and the weight value is 1. Directly related is indicated with ½ weight value. Highly inversely indicates a negative influence in the quality having a -1 weight value. Inversely related indicates less negative influence and is represented with a – ½ weight value. If the metric is not related or not evaluated with criteria the value taken is 0.

For further information refer to the “How to use the Compendium” [5].
### 3 State of the Art

This section describes some of the current tools available for the analysis of software and the calculation of metrics. Each tool has a name, vendor, website and a brief description of the tool. The evaluation of these tools is the foundation for deciding whether to create a new system or reuse/extend an existing one.

The criteria for the evaluation are:

- The tool should be non-commercial, and should have the ability for developers to extend its features.
- Documentation about how to extend new plug-ins should be available.
- Support for the project is considered an important factor because any extension made must have the necessary agreements and support from the supervisors of the project during the development process.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU/Logiscope</td>
<td>Logiscope supports the following programming languages: Ada, C, C++ and Java. It is addresses two kind of problems: Static Analysis for metrics of the source code. Dynamic Analysis: for test coverage on instrumented program. The tool help does not provide any rationale about the selection of the subsets of metrics associated to the criteria.</td>
</tr>
<tr>
<td>McCabe QA</td>
<td>The vendor provides a toolset to cover different aspects of the quality and complexity of SW systems. The vendor implements and automates the industry adopted testing methodology as described in the National Institute of Standards and Technology Special Publication 500-235 [7].</td>
</tr>
<tr>
<td>MAISA</td>
<td>MAISA is a research and development project aiming at developing methods for the measurement of software quality at the design level. The metrics are computed from the system's architectural description, predicting the quality attributes of the system derived from it. Most notably, size and performance metrics are addressed. The performance analysis is refined by analysis at code level. The work in the MAISA project started on the first of September 1999 and ended at the end of December 2001.</td>
</tr>
<tr>
<td>VizzAnalyzer</td>
<td>VizzAnalyzer is a framework for reverse engineering allowing</td>
</tr>
</tbody>
</table>
the integration and interaction of different analysis and visualization tools. The framework uses a dynamic type system to guarantee the proper exchange of data between the tools and a set of wrapper classes to handle their communication. This allows for an easy and secure integration of tools that have originally not been designed to work together. In this sense, existing tools can be (re-)used and integrated.

Table 3.1: Software Quality Analyzer tools

According to the aforementioned criteria, the VizzAnalyzer tool was selected for this thesis in order to implement the solution of the problem as plug-ins. The following reasons were taken into account:

- Reverse engineering and analysis framework delivering the software model on which the SQM can be applied
- Easy extendable with plug-ins
- Well-documented framework
- Total good support from the developers
- Non-commercial tool
4 Requirements
This section describes the technical aspects of the problem and outlines the main idea for the solution. Further, the users of the system, features of the system, and Use Cases for its application are described in order to elicit the functional and non-functional requirements on the solution. The software system developed as part of this thesis has to meet these requirements in the described extent and to fulfill the goals and criteria stated in the introduction.

4.1 Technical aspects of the problem
The source code of a software system is analyzed to create a representation of its internal structure including the semantic relations of the different elements. It is stored in the form of a graph containing nodes representing relevant elements like classes, interfaces, methods and attributes. Edges between these elements represent relationships between the elements (e.g., a class contains methods and attributes), and the semantic relationships (e.g., a method calls another method, and a class uses another class). Further, each element (node or edge) has properties attached to store information such as the name or type of a certain element. New properties can be attached any time to store other values (for instance analysis results for later use).

A Software Quality Model follows the Factor-Criteria-Metrics model and it is necessary to combine input values according to the model to calculate the output values. The inputs deliver the metrics and are read from the properties attached to the different elements in the graph. The outputs are the criteria and the factors whose calculated values are attached to the respective elements in the graph. The values of different metrics have to be combined with a value for a criterion according to the SQM considering the weights.

Example: For instance, a class c1 has metrics M1 to M10 attached as properties. For calculating criteria C1 metrics M1 to M3 are relevant. M1 is highly directly related having a value of 10, M2 is highly inversely related having a value of 15, and M3 is directly related having a value of 8. The weights are w1 = 1, w2 = -1 and w3 = 0.5. Therefore results a value for C1 for class c1 of: C1 = M1*w1 + M2*w2 + M3*w3 = 10*1+15*-1+8+0.5 = -1 indicating a quality below average. A new property C1 is created and attached to c1.

A SQM requires that on all elements in a graph the input is applied to the necessary metrics, which are calculated and attached in the form of properties. This needs to be guaranteed, making a systematic evaluation of all relevant metrics for a SQM essential.

Since a graph can contain thousands of elements, each with dozens of metrics, criteria and factors attached, the information provided by an evaluated SQM is not comprehensible; therefore, a statistical overview is needed to condense the information and to present it in form of averages or outliers, possibly grouped in packages or other organization structures which make sense.

The VizzAnalyzer Framework is an application that supports a fast way to produce a reverse engineering tool and allows analysis of software systems [3][4]. The modularity of VizzAnalyzer allows adding plug-ins and new features to its framework that improve the system for the measure of the software’s quality.

4.2 Define the system
After analyzing state-of-the-art techniques, it has become clear that the solution of choice is to extend the VizzAnalyzer tool with plug-ins and features necessary to perform the computations. The main idea for the solution of the problem is therefore to
create a flexible evaluation engine for the VizzAnalyzer consisting of two plug-ins. It will require GRAIL graphs formed by using the VizzAnalyzer tool and quality models instantiations as input to calculate the metric, criteria and factor values according to the quality model. The result will be available for further analysis or visualization as a GRAIL graph.

In detail: The GRAIL graphs contain the static structure of a software system. The elements of this model represent entities in the software system. The SQM relevant metrics will be calculated and the result is attached to the graph elements in the form of properties. The elements are put into equivalence classes on which statistics will be calculated. This combination or evaluation of the single properties is specified in a XML file as well. The result will be provided as a new graph representing the evaluated quality model. It can be visualized using the capabilities of the existing framework.

The VizzAnalyzer is implemented in Java; its plug-ins will be implemented in Java as well. Working with a framework makes it necessary to follow its plug-in architecture in order to extend it in the intended way and to take advantage of the framework features.

4.3 Users
There is only one user of the system. It is a simple user, utilizing the VizzAnalyzer tool to assist in analysis tasks. The user analyzes a software system, applies metrics, SQM’s and other analysis, and creates visualizations out of the results. The user also creates the SQM definitions (XML file) according to the compendium and adjusts it to the specific needs of a software system.

4.4 Features
This subchapter summarizes and formalizes the features of the software system which are derived from the problem analysis. From these features the requirements of the software system will be derived.

<table>
<thead>
<tr>
<th>Feature 01</th>
<th>Capability to calculate a Quality Model on Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The developed software system has the capability to calculate a Software Quality Model on graphs. It reads the metrics values from the relevant elements, combines them according to the SQM and stores the resulting values for the factors and criteria in the relevant elements. The calculations are repeatable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature 02</th>
<th>Capability to calculate Statistics on Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>The software system has the capability to calculate statistics methods over each node on the Graph, storing the results in the graph for further calculations.</td>
</tr>
</tbody>
</table>

Table 4.1: Features

4.5 Use Cases
To better understand the requirements resulting from the interaction between the intended system and the user, it is analyzed with the help of Use Cases. The next section shows an overview about the relevant Use Cases in a Use Case diagram. These Use Cases will be described in detail in the section after.

4.5.1 Use Case model
The gray ellipses are Use Cases already implemented by the VizzAnalyzer tool which are relevant for the new uses of the new features of the system. They are included in the
model to show the interface between the existing features of the VizzAnalyzer and the newly implemented ones.

The already available Use Cases for the VizzAnalyzer tool are used to analyze an existing software system, thereby generating the needed GRAIL graphs. Further, it is possible to apply single metrics on a specific graph, to define equivalence classes and to visualize a graph. The new Use Cases allow the application of a SQM, the generation of statistics, and the application of all metrics needed for a SQM.

![Use Case Diagram](image)

**Figure 4.1: Use Case Diagram**

The new Use Cases extend from the current VizzAnalyzer tool and include the performance of equivalent classes’ definition and single metric. The new Use Cases for applying a quality model and generating statistics include the visualization of the results as is shown in Figure 4.1: Use Case Diagram. Further, it is possible to apply more than one metric belonging to a VizzAnalyzer tool at the same time on a specific graph.

### 4.5.2 Use Cases

This subchapter describes the Use Cases in detail, defining a goal-oriented set of interactions between external actors and the current system. The Use Cases illustrate the features which make it easier to transform them into requirements.

There are two different kinds of Use Cases. The first is where the Use Case, which is already implemented by VizzAnalyzer tool, is explained by a title and description. The second type is where each Use Case, implemented for the context of this thesis, has a particular task and is described by a title, a goal, a pre-condition, post-condition, actors evolved in the use case, trigger event that execute the use case, description of the use case, extensions and alternatives events in the use case.
## Use Cases

<table>
<thead>
<tr>
<th>Use Case UC1</th>
<th>Analyse Software System</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Extract quantitative information about software systems provided in source code. This information is stored in internal data used for instance, for the calculation of metrics or getting data for other purposes extending the features of the system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use Case UC2</th>
<th>Define Equivalence Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Grouping the elements in a graph to extract information is required. This use case provides different ways to group information, for instance, equivalence classes by packages and by classes. Equivalence classes by classes are used for the context of this thesis getting a group containing all classes in the system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use Case UC3</th>
<th>Apply Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Metrics help to measure different aspects of maintainability as well as other qualities of a system. VizzAnalyzer tool provide different metrics as Depth of Inheritance Tree (DIT), Number of Children (NOC), Data Abstraction Coupling (DAC) and others metrics defined in “The Compendium of Software Quality and Metrics” [12]. The user apply a metric one by one.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use Case UC4</th>
<th>Visualization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>VizzAnalyzer tool provide a Visualization Use Case. The user visualizes the results of different operations, for instance, metrics, equivalence classes and a quality model within the graph.</td>
</tr>
</tbody>
</table>

*Table 4.2: VizzAnalyzer Tool Use Cases*
<table>
<thead>
<tr>
<th>Use Case UC5</th>
<th>Apply Quality Model.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Evaluation of a SQM on a selected graph, thereby attaching to each node in the graph the values calculated for its factors and criteria.</td>
</tr>
<tr>
<td><strong>Pre-condition</strong></td>
<td>Graphs of an analyzed software system are available within the VizzAnalyzer; a QMXML file (see Chapter 4.6 requirement R01) defining a SQM is available; EC are defined; the metrics needed by the SQM are calculated for the graph used.</td>
</tr>
<tr>
<td><strong>Post-condition</strong></td>
<td>Each node in the graph contains the result for its SQM.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>User</td>
</tr>
<tr>
<td><strong>Triggering event</strong></td>
<td>User wants to use the Quality Model Plug-in to attach information to the graph. He starts the plug-in from within the VizzAnalyzer tool.</td>
</tr>
</tbody>
</table>
| **Description** | User selects a graph and specifies which EC wants to used (see UC2).  
The user applies metrics in the graph (see UC3).  
The user executes the Quality Model Plug-in having already all the information necessary.  
The user visualizes the result within the graph for all the nodes (see UC4). |
| **Extensions** | Extend from UC1. |
| **Alternatives** | User introduces a wrong XML file. The file includes a DTD file that checks the consistent of the XML file. The Quality Model Plug-in checks that the file has one Quality Model in the QMXML file. If there are more than one QM the system shows that the file is inconsistent.  
User doesn’t execute an EC for the Quality Model. The system shows an error.  
User doesn’t apply any metric on the Quality Model. The system shows an error.  
User doesn’t apply all metrics defined in the QMXML file on the Graph. The Quality Node can’t apply the result for each node on the Graph the new properties FCM. The system shows error. |
<table>
<thead>
<tr>
<th>Use Case UC6</th>
<th>Calculate Statistics.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>Apply different statistics methods on a Graph.</td>
</tr>
<tr>
<td><strong>Pre-condition</strong></td>
<td>Graphs of an analyzed software system are available within the VizzAnalyzer; a SQM has been applied, equivalence classes are defined.</td>
</tr>
<tr>
<td><strong>Post-condition</strong></td>
<td>Different statistics methods are applied for each EC, the results are stored as new node, representing the equivalence class.</td>
</tr>
<tr>
<td><strong>Actors</strong></td>
<td>User</td>
</tr>
<tr>
<td><strong>Triggering event</strong></td>
<td>User wants to apply the statistics onto a Graph, to get a better overview.</td>
</tr>
</tbody>
</table>
| **Description** | User selects a Graph and specifies which EC wants to used using CrocoPat (see UC2).  
The user applies Metrics in the Graph (see UC3).  
The user executes the Statistics Plug-in specifying type of EC, Metric to apply the statistics and statistic method to apply.  
The user visualizes the result within the graph for all the nodes (see UC4). |
| **Extensions** | Extend from UC1. |
| **Alternatives** | User introduces a wrong XML file. The file includes a DTD file that checks the consistent of the XML file. The Quality Model Plug-in checks that the file has one Quality Model in the QMXML file. If there are more than one QM the system shows that the file is inconsistent.  
User doesn’t execute an EC for the Quality Model. The system shows an error.  
User doesn’t apply any metric on the Quality Model. The system shows an error. |
Use Case UC7: Apply Metrics

Goal: Apply all the metrics required by a SQM on a graph to guarantee, that the metrics values needed as input are present and up-to-date.

Pre-condition: Graphs of an analyzed software system are available within the VizzAnalyzer. All metrics can be applied on the graph and are available as plug-in in the VizzAnalyzer.

Post-condition: The graph has applied the metrics for all the nodes in it.

Actors: User

Triggering event: User wants to prepare the application of a SQM onto a specific graph by calculating all required metrics.

Description: User selects a graph the metrics shall be applied on. The User runs the script applying all metrics onto the selected graph.

Extensions: -

Alternatives: -

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Event/use case#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement R01 Defining a Software Quality Model file</td>
<td>The System shall provide a way to define a Software Quality Model by the user.</td>
<td>UC5</td>
</tr>
<tr>
<td>Rationale:</td>
<td>It is necessary to define the correct structure for the Software Quality Model file and perform the analysis.</td>
<td></td>
</tr>
<tr>
<td>Fit Criterion:</td>
<td>The user is able to define a Software Quality Model describing the required information.</td>
<td></td>
</tr>
</tbody>
</table>

Requirement R02 Load Graphs | Event/use case#: UC1,2,3,4,5,6,7
Description: | The System shall provide a way to load different graphs on the system. |
Rationale: | It is necessary for the User to load different graphs in the system that will be use to attach new information. |
Fit Criterion: | The user is able to load different graphs in the system and apply different features on them. |

Requirement R03 Select Graphs | Event/use case#: UC1,2,3,4,5,6,7
Description: | The System shall provide a way to select a graph from the graph list loaded in the system. |
Rationale: | It is necessary to select a graph to apply different features on it. |
Fit Criterion: | The user is able to select a graph. |

4.6 Functional requirements
This chapter describes the functional requirements capturing the intended behavior of the system. This behavior may be expressed as services, tasks or functions which the system is required to perform. Each requirement results from the features and Use Cases described above. Each requirement has a unique identifier, a name and a reference to the Use Case it belongs to, a description, rationale and a fit criterion.
### Functional requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Event/use case#</th>
</tr>
</thead>
<tbody>
<tr>
<td>R04 Select Equivalence Classes</td>
<td>The System shall provide a way to select different equivalence classes in a Graph.</td>
<td>UC2,5,6</td>
</tr>
<tr>
<td>Rationale:</td>
<td>It is necessary to select different equivalence classes grouping the information of the graph.</td>
<td></td>
</tr>
<tr>
<td>Fit Criterion:</td>
<td>The user is able to select different equivalence classes for the graphs.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Event/use case#</th>
</tr>
</thead>
<tbody>
<tr>
<td>R05 Apply Equivalence Classes</td>
<td>The System shall apply the equivalence classes by the user grouping the information in the system.</td>
<td>UC2,5,6</td>
</tr>
<tr>
<td>Rationale:</td>
<td>It is necessary to apply equivalence classes in a graph grouping the information.</td>
<td></td>
</tr>
<tr>
<td>Fit Criterion:</td>
<td>The user is able to apply different equivalence classes.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Event/use case#</th>
</tr>
</thead>
<tbody>
<tr>
<td>R06 Select Metric/s</td>
<td>The System shall provide a way to select different metrics in a Graph.</td>
<td>UC3,5,6,7</td>
</tr>
<tr>
<td>Rationale:</td>
<td>It is necessary to select different metrics in specific graph extending the information of the graph.</td>
<td></td>
</tr>
<tr>
<td>Fit Criterion:</td>
<td>The user is able to select different metrics for the graphs.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Event/use case#</th>
</tr>
</thead>
<tbody>
<tr>
<td>R07 Apply Metric/s on Graphs</td>
<td>The System shall apply metrics by the user in different graphs, extending the information of the system.</td>
<td>UC3,5,6,7</td>
</tr>
<tr>
<td>Rationale:</td>
<td>It is necessary to apply equivalence classes in a graph grouping the information.</td>
<td></td>
</tr>
<tr>
<td>Fit Criterion:</td>
<td>The user is able to apply different equivalence classes.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Event/use case#</th>
</tr>
</thead>
<tbody>
<tr>
<td>R08 Select Software Quality Model file</td>
<td>The System shall provide a way to select a Software Quality Model file.</td>
<td>UC5</td>
</tr>
<tr>
<td>Rationale:</td>
<td>It is necessary to select a Software Quality Model to perform a Quality Model in different graphs of the system.</td>
<td></td>
</tr>
<tr>
<td>Fit Criterion:</td>
<td>The user is able to select a Software Quality Model extending the information for each node from a graph.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Event/use case#</th>
</tr>
</thead>
<tbody>
<tr>
<td>R09 Select Statistic method</td>
<td>The System shall provide a way to select different statistics methods in a Graph.</td>
<td>UC6</td>
</tr>
<tr>
<td>Rationale:</td>
<td>It is necessary to select different statistics methods in specific graph extending the information of the graph.</td>
<td></td>
</tr>
<tr>
<td>Fit Criterion:</td>
<td>The user is able to select different statistics methods for different graphs.</td>
<td></td>
</tr>
<tr>
<td>Requirement</td>
<td>Event/use case#</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>R10 Apply Statistics on Graphs</td>
<td>UC6</td>
<td>The System shall apply statistics methods by the user in different graphs, extending the information of the system.</td>
</tr>
<tr>
<td>R11 Apply Software Quality Model on Graphs</td>
<td>UC5</td>
<td>The System shall apply a Software Quality Model specify for a Software Quality Model file by the user. The result is applied in a graph extending the information of the system.</td>
</tr>
<tr>
<td>R12 Display Results</td>
<td>UC1,2,3,4,5,6,7</td>
<td>The System shall provide a result of: applying a QM, calculating Statistics.</td>
</tr>
<tr>
<td>R13 Error Handling</td>
<td>UC1,2,3,4,5,6,7</td>
<td>The System shall provide a way to control the exceptions that the plug-ins could throw.</td>
</tr>
</tbody>
</table>

Table 4.4: Requirements
5 Architecture

This chapter describes the architecture of the solution and provides background about the reused parts of the VizzAnalyzer framework. It explains how the existing architecture of the VizzAnalyzer is reused and extended; how the plug-ins, once created, interact and integrate to provide a solution for the problem; and it introduces the structure of the QXML file used for describing a SQM in a human and machine readable form.

5.1 VizzAnalyzer architecture

In general, the VizzAnalyzer Framework is a system for reverse engineering, supporting the rapid composition of individual software reverse engineering tools by reusing arbitrary reverse engineering components.

Figure 5.1: VizzAnalyzer architecture shows the architecture of the VizzAnalyzer Framework incorporating a domain-specific, tool-based reverse engineering process. Each sub-process is defined by reverse engineering components that are plugged via so-called wrappers into respective variation points.

![Figure 5.1: VizzAnalyzer architecture](image)

The framework consists of concrete and abstract classes. The concrete classes implement the tool-based reverse engineering process and define the framework core. The abstract classes, also referred to as hot spots, enable the instantiation of the framework. The composition is defined by wrappers, which handle the information exchange between the framework and a component [3].

Different wrappers are available for retrieval, analysis and visualization components. Doing possible add new features to the VizzAnalyzer Framework.

What is interesting for this thesis is the reuse of the framework with its already existing retrieval components and the shared model, further extending it with appropriate plug-ins at the variation point for analysis. How the plug-ins integrate into the existing architecture is described in the following section.

5.2 VizzAnalyzer plug-ins architecture

The VizzAnalyzer framework (VAF) provides a complex API, yet the part necessary for developing the plug-ins is rather comprehensible. Figure 5.2: Analyzer Framework Design depicts the classes dedicated to the plug-in mechanism. The VAF allows adding several new plug-in types into the framework. It provides abstract classes for creating
new plug-ins based on the Mergers, Aggregator, Filters, HLANalyses and Metrics classes. For creating a new plug-in, only the appropriate class needs to be extended, and the implementing file stored in the appropriate package.

The gray area of the framework is the relevant part that has been used to develop the plug-ins. The HLANalysis class gets the information from graphs available; when getting the graphs, it is possible to extend these graphs to new plug-ins. The MetricsAnalysis class is used to apply metrics on the graph and it is used before the plug-ins to assure the correct perform of the plug-ins developed. The HLANalysisAbstract abstract class defines the methods that are necessary to implement by new plug-ins performing high level analysis on graphs stored in the VizzAnalyzer.

![Analyzer Framework Design](image)

### 5.3 Overall architecture for the solution

The requirement analysis resulted in the identification of three main parts for the solution, which will be integrated in VizzAnalyzer. These parts can work independently since they provide for themselves useful features, but once they are combined in the right way (evaluation engine) they will perform the calculations needed in order to apply a SQM to an existing graph, calculating the factors and criteria from the metrics, and providing a statistical overview about them. The architecture connecting these independent components is depicted in Figure 5.3: Pipeline Architecture.

The input creating the data model is provided by the retrieval components in the VizzAnalyzer. The data model contains an analyzed software system represented as a graph. Its static program structure contains the elements on which the SQM will be applied.

The Metric component, on which the SQM is based, makes sure that all elements in the graph have the metrics calculated. This calculation is needed so that the SQM component can perform meaningful work, but can also serve independently for the evaluation of metrics as input for other analysis or visualization.
The SQM component uses the data model, with metric information enriched, to perform the evaluation of the SQM. The SQM is defined in a SQM definition file stating how the metrics are combined to criteria and how the criteria are combined to factors and in the end to an overall quality value. The results of the calculations are added to the data model.

Statistical information about the metric, factor and criteria values present in the data model are calculated with the statistics components, which can also work on other types of numerical data contained in the data model.

After all three components are applied to the data model, it contains the SQM related information which can be visualized or serve as input to other analyses, as it is connected to the framework over the variation points.

**Figure 5.3: Pipeline Architecture**

### 5.4 The Quality Model XML file

This section describes how the Software Quality Model defined in the compendium is captured in a human and machine readable form. The SQM is stored in a Quality Model XML file (QMXML) representing the structure for the SQX. It stores the relations between the metrics, the criteria and factors, and their respective weights.

The *weight* specifies how important a metric for a concrete criterion is, and how much influence a certain criterion has for a specific factor. The possible weights are defined in the SQX and have been discussed in section 2.

Any SQM based on the FCM model can be described using this file structure. It will be instantiated as a XML file following the structure defined in the DTD accompanying the SQM plug-in.

The QMXML has the following structure matching the XML parser implemented as part of the SQM plug-in. The mapping from the SQX to the QMXML definition is as follows:

- **Metrics on the System** is represented by the tag `<metricsOnTheSystem>` and specifies the metrics used to calculate the QM for one Graph. The user needs to specify the metrics that they want to apply in the QM. This helps the user to have an overview about all the metrics that they are using, and giving usability. Another advantage to using this tag is that the user does not need to write all
metrics in each criterion with having irrelevant ones having weight 0 as it is explained in chapter 2 and having as reference Figure 2.1: Quality Software Matrix representation where it is possible to see the metrics not related, not evaluated and thus having a weight 0 for them.

- **Quality Model** is represented by the `<quality>` tag. It is just possible to have one QM; it means that it is not possible to apply more QM at the same time. This combines all factors in the quality model into one value representing the quality of a software system or part of it, according to the SQM. It is a combination of the different factors.

- **Factor** is represented by the `<factor>` tag. Specifying a single factor representing in the SQM. A factor is a combination of different criteria.

- **Criteria** is represented by the `<criteria>` tag. Specifying which metrics are relevant for the calculation of this aspect of the SQM. A criterion is a combination of different metrics.

- **Metric** is represented by the `<metric>` tag. Specifies the name of the metric property containing the value being input for the calculation of the respective criterion.

- This structure is defined in the QualityModel.dtd, which can be seen in Figure 5.4: Quality Model DTD.

```xml
<!ELEMENT root (metrics+,quality+)>
<!ELEMENT metrics (metricOnTheSystem+)>
<!ELEMENT quality (factor+)>
<!ELEMENT factor (criteria+)>
<!ELEMENT criteria (metric+)>
<!ELEMENT metric (#PCDATA)>
<!ELEMENT metricOnTheSystem (#PCDATA)>

<!ATTLIST quality  name CDATA #REQUIRED weight CDATA #REQUIRED>
<!ATTLIST factor name CDATA #REQUIRED weight CDATA #REQUIRED>
<!ATTLIST criteria name CDATA #REQUIRED weight CDATA #REQUIRED>
<!ATTLIST metric name CDATA #REQUIRED weight CDATA #REQUIRED>
<!ATTLIST metricOnTheSystem name CDATA #REQUIRED>
```

Figure 5.4: Quality Model DTD
A brief algorithm is executed for metric, criteria and factor tag and using the results get it for the parent factor, criteria and metric in each sub/level:

For each name found in the QMXML file

\[ \text{Factor/Criteria/Metric value} \times \text{weight value} = X_n. \]

\[ \text{Result} = \text{result} + X_n \]

End for

The algorithm is a recursive algorithm using as values for Criteria the result gets it for its Metrics in that level and for Factors the result gets it for its Criteria.

Figure 5.5: QMXL example is a sample QM XML definition file, defining a fictive Software Quality Model combining three metrics into five criteria and two factors. In detail we can see that the user is applying the metrics DIT, NOC and LOC. For the current sample the DIT value is 1, the NOC value is 0.5 and LOC value is -0.5. These values are provided by the system before to apply them in the Software Quality Model as is described in UC3 and UC7 chapter 4.5.2.

One Quality Model is defined having a qualityTree name with weight value 1 being highly directly related with the factors re-usability and portability. Both factors have a weight value informing how related they are with their criteria.

Summarizing the process and getting the values for the metrics above described, the results for each metric, criteria and factor for the current sample is:

1. Criteria understability = \(-0.5 \rightarrow [ (1 \times \text{DIT}) \times -1 (\text{weight}) ] + (-0.5 \times \text{LOC} \times -1 (\text{weight})) \]
2. Criteria operability = 0.75 \rightarrow [ (0.5 \times 1) + (-0.5 \times -0.5) ]
3. Criteria attractiveness = 1 \rightarrow [ (1 \times 1) + (0.5 \times 1) + (-0.5 \times 1) ]
4. Factor re-usability = 1.25 \rightarrow [ \text{number 1} + 2 \times 3] \rightarrow [-0.5+0.75+1]
5. Criteria adaptability = -1.25 \rightarrow [ (1 \times -1) + (0.5 \times -0.5) ]
6. Criteria replaceability = -0.75 \rightarrow [ (1 \times -1) + (-0.5 \times -0.5) ]
7. Factor portability = -2 \rightarrow [ \text{number 5} + 6] \rightarrow [-1.25+(-0.75)]
8. Quality Model qualityTree = -0.75 \rightarrow [ \text{number 4} + 7] \rightarrow [1.25+(-2)]

The result for this Software Quality Model is -0.75.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root SYSTEM "QualityModel.dtd">
<root>
  <metrics>
    <metricOnTheSystem name="DIT"/>
    <metricOnTheSystem name="NOC"/>
    <metricOnTheSystem name="LOC (Class)"/>
  </metrics>
  <quality weight="1" name="qualityTree">
    <factor name="re-usability" weight="1">
      <criteria name="understability" weight="1">
        <metric name="DIT" weight="-1"/>
        <metric name="LOC (Class)" weight="-1"/>
      </criteria>
      <criteria name="operability" weight="1">
        <metric name="NOC" weight="1"/>
        <metric name="LOC (Class)" weight="-0.5"/>
      </criteria>
      <criteria name="attractiveness" weight="1">
        <metric name="DIT" weight="1"/>
        <metric name="NOC" weight="1"/>
        <metric name="LOC (Class)" weight="1"/>
      </criteria>
    </factor>
  </quality>
</root>
```
<factor name="portability" weight="1">
    <criteria name="adaptability" weight="1">
        <metric name="DIT" weight="-1"/>
        <metric name="NOC" weight="-0.5"/>
    </criteria>
    <criteria name="replaceability" weight="1">
        <metric name="DIT" weight="-1"/>
        <metric name="LOC (Class)" weight="-0.5"/>
    </criteria>
</factor>

Figure 5.5: QMXL example
6 Design and implementation

This section describes how the architecture discussed in the previous section has been transformed into a design and finally has been implemented. Since the developed software itself is not complex enough to justify a distinct design and implementation model, design and implementation have the same level of abstraction. The following sections explain in detail how the significant parts of the architecture – Metrics component, SQM component and Statistics component– have been implemented as plug-ins into the VizzAnalyzer.

6.1 Metrics component (script)

The purpose of this component is to calculate all metrics required by a SQM on a specified graph. This could have been implemented as plug-in for the VizzAnalyzer, retrieving the list of metrics from a definition file, and then requesting the available metrics from the VizzAnalyzer framework, in order to apply them one by one on the specified graph. Yet since the VizzAnalyzer framework provides a scripting feature, the required functionality can be implemented without any development effort by recording a script executing the metrics needed by a specific SQM. Since this was the most efficient solution, a simple script has been recorded and is provided below:

```
// get selected graph from VizzAnalyzer
Object[] graphs0 = new Object[] { pm.getSelectionForName("Class Hierarchy Graph", false) };
// instantiate NOC metric
hlanalysis.metrics.NumberOfChildren hlaEngine0 = new hlanalysis.metrics.NumberOfChildren();
// run analysis
hlaEngine0.accept((grail.interfaces.GraphInterface) graphs0[0]);
hlaEngine0.setExtendCurrentGraph(true);
hlaEngine0.startAnalysis();
// save result
pm.setGraph(hlaEngine0.getResultGraph(), false);

// get selected graph from VizzAnalyzer
Object[] graphs1 = new Object[] { pm.getSelectionForName("Class Hierarchy Graph", false) };
// instantiate DIT metric
hlanalysis.metrics.DepthOfInheritanceTree hlaEngine1 = new hlanalysis.metrics.DepthOfInheritanceTree();
// run analysis
hlaEngine1.accept((grail.interfaces.GraphInterface) graphs1[0]);
hlaEngine1.setExtendCurrentGraph(true);
hlaEngine1.startAnalysis();
// save result
pm.setGraph(hlaEngine1.getResultGraph(), false);

// get selected graph from VizzAnalyzer
Object[] graphs2 = new Object[] { pm.getSelectionForName("Class Hierarchy Graph", false) };
// instantiate LOC metric
hlanalysis.metrics.Size hlaEngine2 = new hlanalysis.metrics.Size();
// run analysis
hlaEngine2.accept((grail.interfaces.GraphInterface) graphs2[0]);
hlaEngine2.setExtendCurrentGraph(true);
hlaEngine2.startAnalysis();
// save result
pm.setGraph(hlaEngine2.getResultGraph(), false);
```

Figure 6.1: Metrics component script (NOC, DIT, LOC)
6.2 Software Quality Model component (plug-in)

This section shows how the architecture described above has been implemented for the Software Quality Model component in form of a plug-in. It explains the design and implementation of the plug-in itself in section [6.2.1] and of the data structure in section [6.2.2].

6.2.1 Software Quality Model plug-in

The classes and interfaces implemented in the Software Quality Model Plug-in are depicted in Figure 6.2: Class Diagram SQM plug-in component together with their important associations. The white elements in the class diagram have been newly implemented to extend the VizzAnalyzer framework (gray).

The QualityModelPlugin class extends from the gray HLAnalysis interface, which is provided by VAF and the connection to it. The interface defines two methods which need to be implemented by any class realizing the HLAnalysis interface. The accept() method is called by the framework, after an analysis is applied to a specific graph. If the analysis is compatible with the graph and it returns true, it stores a reference to this graph for later access by its analysis algorithms. When the Graphs can be accepted, the startAnalysis() method is executed to run a final analysis on the graph.

The FCMReader class extends from QMReader interface. The interface is defined to create new readers for different Software Quality Models defined in the future. For the context of this thesis a relevant quality model is based on the SQX and instantiated in the QMXML file. The appropriate reader is implemented in the class FCMReader.

The QualityModel class reads a QMXML file using the appropriate instantiation of the QMReader, thereby creating a data structure stored as objects of the class QualityNode. The data model represented by the QualityNode and the MetricsValue class is explained in detail in the next section.

The QualityModel class contains Input and Output HashMap. Input HashMap contains all the metrics specified in the metricsOnTheSystem tag from the QMXML file. The metrics of each criterion are instantiated with its name from the Input HashMap and sharing the value of that metric. Subchapter 6.2.2 described this process detailed. Output HashMap contains the results of the quality model, factor and criteria for each node of the graph being attached as new properties in the node afterwards.

Figure 6.2: Class Diagram SQM plug-in component
Figure 6.3: Sequence diagram Software Quality Model evaluation shows the interaction between the user, the VizzAnalyzer framework and the classes of the plug-in in form of message sequences.

1. The user runs the VAF.
2. The user load graphs on the system and selects one of them for a Software Quality Model evaluation.
3. The user applies equivalence classes to grouping the elements of a graph.
4. The user selects a QMXML file describing the quality model that he wants to apply in the graph selected.
5. The user executes the Quality Model Plug-in.
6. It is created an object *QualityModelPlug-in* storing the information necessary during the process.
7. The plug-in accept the graphs on the System where it is possible to apply a Software Quality Model.
8. After accept the graphs the *StartAnalysis()* method is executed.
9. During the *StartAnalysis()* method is created a Quality Model object that.
10. Create an instance to read a FCMReader containing a QM structure stored in FCM.
11. The information provided by the QMXML file is stored.
12. A graph contains nodes and for each node is going to execute a *compute()* method that
13. Read metric values for a node.
15. Evaluating the SQM storing the values for a FCM.
16. Reading the SQM for
17. Storing the result in the graph.
**5. Finish**, the user visualizes the new properties for the nodes.
Figure 6.3: Sequence diagram Software Quality Model evaluation
6.2.2 Data Structure

The classes used to store and calculate the factors, criteria and metrics of a SQM are depicted in Figure 6.4: Data Structure Design. The factors and criteria are stored in `QualityNode` object, having a root `QualityNode` representing the quality model itself. Below this are `QualityNode` objects for each factor in the model, each of them having `QualityNode` objects as children representing the criteria. This is in principle the tree structure of the QMXML file. For storing the metric values, a `MetricValue` object is added for each metric belonging to a criterion to the `QualityNode` representing this criterion.

A `QualityNode` class has getter and setter methods defined for accessing the different attributes during creation and evaluation. The `name` attribute is the name of the factor or criterion and the weight the factor of how much its value is relevant for the combination with the next higher level in the QM. The value attribute stores the current value calculated for the factor or criterion, and children stores the children of the node. New children are added during the instantiation of the data structure using the `addChild` method. Depending on setting a metric value or a double value the according `setValue` method is used, and this process is shown in Figure 6.5: Sequence Diagram data structure creation and evaluation detailed. The `getValue()` method is a recursive method. It performs the actual evaluation of the SQM after it has been instantiated. To call this method on a node causes the node to call that method on all of its children, thus returning their actual value. The child value is multiplied with its weight and all child values are added up to calculate the value of the node.

![Data Structure Class Diagram](image)

**Figure 6.4: Data Structure Design**

Figure 6.5: Sequence Diagram data structure creation and evaluation depicts a sequence diagram describing in detail what happens how the data structure is build up while parsing the QMXML file.

Metrics occur several times in a SQM, being possibly associated with more than one factor at a time. For a more efficient implementation, the metric value is stored only once, and the factors share a single instance of it. Therefore a criterion has its own weight for each metric.
Figure 5.5: QMXL example summarizes the calculation showing how to get the results for each node and level. The same information is used to represent Figure 6.6: Data Structure example, which describes the data structure generated for a Software Quality Model.

In the legend it is possible to see and sphere that represents a Quality Node, a square which represents a metric object instantiated for different Quality Nodes. All objects created are Quality Nodes and they have their own weight. The Quality Model is represented at the top for a QualityTree quality node. Re-usability and portability represent factors. Understandability, operability, attractiveness, adaptability and replaceability represent criterias. An output HashMap is used to store the information of the Quality Node, which is a unique indicator.

Metrics occur several times, as the same metric value is used for different factors/criterions. An input HashMap stores the values of these metrics sharing them for different factor/criteria.

This data structure is attached as properties in each node of the graph extend the information of each one with its own FCM values.
6.3 Statistics Plug-in

This section describes how the architecture for the Statistics component has been implemented.

Figure 6.7: Class Diagram Statistics component plug-in depicts a class diagram representing the relation between the classes of the Statistics plug-in. As for the Quality Model plug-in, the \texttt{StatisticsPlugin} class extends from the gray \texttt{HLAnalysis} interface, which is provided by VAF and the connection to it. It handles the user interaction, and controls which statistics are calculated on which graphs, and how the result is stored. The \texttt{Statistics} interface is defined for easy extension with new statistic methods for this plug-in and VAF. It is necessary to implement two methods for a new statistic method, and \texttt{getName()} gives the name of the statistic method implemented. The \texttt{getResult(arrayList)} method returns a \texttt{Double} for the statistic method applied. The \texttt{ArrayList} contains all the values to evaluate the statistic method. The \texttt{getName()} and \texttt{getResult(arrayList)} are defined in \texttt{Statistics interface} because they are parameters suitable for the most of the statistical methods.

Currently, the plug-in includes three statistical methods for calculating the average, maximum and minimum of a series of values and they are implemented in the: \texttt{Average}, \texttt{Minimum} and \texttt{Maximum} classes respectively. The result is stored in the graph creating a new node per each equivalence classes, adding the result in it.
1) The user runs the VAF.
2) The user load graphs on the system and selects one of them for a Software Quality Model evaluation.
3) The user applies equivalence classes to grouping the elements of a graph.
4) The user executes the Statistics Plug-in.
5) The system asks for the parameters EC, metric and statistic method to execute.
6) The Statistics Plug-in is executed by the user.
7) It is created an object StatisticsPlug-in storing the information necessary during the process.
8) The accept() method is executed checking valid graphs for the system.
9) After accept the graphs the StartAnalysis() method is executed.
10) During the StartAnalysis() method is created a the Statistic object that
11) Match with the statistic method that the user wants to apply for each node in the Graph
12) During this loop, all the properties from a Graph are stored in a HashMap and
13) Using the HashMap created in 12) then the system apply the statistic method storing the results in a new node for each equivalence class in the graph as a new properties.
6) Finish, the user visualizes the new properties for the nodes in the graph.
Figure 6.8: Sequence diagram Statistics plug-in calculation
7 Conclusion and Future Work

This last chapter reflects on the results of the thesis. It points out the extent to which the problem described in the introduction has been solved and if all the goals have been met and the criteria fulfilled. It further discusses possibilities and ideas as to how the outcome of the thesis could be improved and directions of future developments.

7.1 Conclusions

A great deal of effort has been spent on describing the quality of a software system in a quantifiable way using models. Not all of these models can be automatically evaluated using measured directly derived from the source code of the software system. The ISO 9126 standard is one model which is well-defined and widely distributed. Further measures of software entities, e.g. Lines of Code, Depth of Inheritance Tree, and Number of Children, have been developed and known about for decades, but they had not shown their usefulness for the assessment of software quality. Attempting to bring the qualitative aspects of a software quality model in relation to the quantitative aspects of software metrics is a newer idea.

Appropriate tool support is necessary to support the research and experiments of Lincke and Löwe 0. The tool developed in this thesis expands on an existing software analysis tool, the VizzAnalyzer. It allows the automated evaluation of a Software Quality Model to perform based on the ISO 9126 standard and well known software metrics. It supports the evaluation of the required metrics as precondition, and supports the comprehension of the results by providing statistical calculations.

To repeat, the problem addressed by this thesis was:

“The extension of the VizzAnalyzer tool with suitable analysis allowing the automated evaluation of a flexible Software Quality Model, the automated calculation of the required metrics and the statistical analysis of the results.”

This problem was concretized in Chapter 1.3 through goals and criteria. The goals were specified by requirements in Chapter 4. The requirements have been analyzed and under consideration within the context of the background, available tools, and architecture, and were developed and described in Chapter 5. Chapter 6 describes the design and implementation of the architecture in form of plug-ins and scripts.

How have the goals and criteria formulated in Chapter 1.3 been addressed?

The first goal was to make sure that all the metric values a SQM relies on are actually calculated and available in the graph, on which the SQM shall be applied on. This goal is met by recording a script performing the evaluation of the needed metrics on a specified graph. The solution has been discussed in detail in Chapter 6.1. The preconditions for successfully executing all metrics is that the graph is suitable for all metrics which shall be applied on it, and that the VizzAnalyzer tool provides the requested metric analyses. Not re-implementing the metrics itself, but reusing existing functionality, it is assumed (and has been tested), that if the metrics analyses function properly the criterion will be met, and all individual and relevant elements in the graph will have their metric values attached as properties.

Finding a suitable format for describing the SQM in a machine and human readable form was the second goal. It was particularly important to maintain all relevant information of the SQM, including the definitions of factors, criteria, metrics, and their weighted relations which have been reached by defining a XML file (QMXML) structure preserving the tree-like containment structure of the FCM model. The file format is easily readable by humans and machines alike. Weight information is contained, and it can be edited with a normal text editor. Details about the solution for this goal are presented in Chapter 5.4.
The major goal was the evaluation of a SQM as defined in a QMXML file and storing the resulting information for the contained factors and criteria in the graph under evaluation. This goal has been reached by implementing a SQM plug-in (Chapter 6.2) as the core of the architecture described in Chapter 5.3. By designing the solution as different independent parts, which combined solve the problem, but in addition can be individually combined with other activities, thereby having an individual value, adds value to the VizzAnalyzer beyond reaching the goals stated.

The last goal of calculating statistics over the evaluated SQM data has been realized by implementing a flexible and extendable statistics plug-in, as described in Chapter 6.3. It fulfills the criteria of applying different statistics methods, including maximum, minimum and average to all elements of a graph or groups of them.

Apart from this, by using object oriented design and keeping extensibility in mind, it is easily possible to extend the set of statistical methods, by simply implementing the Statistics interface. Changes in the SQM are possible within the limit of the defining DTD, and new metrics can easily be added to an existing script, by either altering the script file or re-recording the script including the new metrics.

During the development and testing of the created components no performance issues were recognized. All operations seem to perform in feasible time, which is in agreement with the secondary concerns stated in Chapter 1.3.

The previous paragraphs show that all goals stated in the beginning of this thesis have been successfully addressed. Therefore it can be concluded that the problem underlying this thesis has been solved.

7.2 Future work

The implemented plug-ins serve their purpose and are suitable for solving the problem addressed by this thesis. Yet during the development certain improvements were noted. Currently the evaluation of the metrics needed for a SQM is defined in a separate component. The script creator has to take care that the metrics script suits the created QMXML file. This could be improved by implementing a plug-in, or extending the SQM component to execute the metrics directly on the graph, depending on the specification of the QMXML file, so that this component does not rely on the proper application of another component.

The VizzAnalyzer framework provides a consistent and flexible way to extend the framework with new plug-ins. Scripting and input of user parameters is supported, which allows a flexible use of the plug-ins and saves time by automating routine tasks. Yet user input is limited to several text fields by allowing only input as string. Presenting lists of choices or other user input is currently not supported. This limits, for example, the selection of statistics. In the future the VizzAnalyzer framework could gain value from allowing their plug-ins a more rich set of interaction widgets.

The SQM component relies currently on a Factor-Criteria-Metric model based structure of the QMXML definition file. In future software quality models having a different structure, for instance, three or more levels of indirection between the metrics measuring attributes in the software and the final quality value, or a structure different from the used tree structure, could be useful. A more flexible implementation could be advantageous. Also the algorithm for combining the values and weights of the different elements of the SQM could be exchangeable.

The comprehension of the results is eased by performing statistical calculations. Comprehending how the calculated quality of specific element or group of elements is determined by the factors and criteria could be supported by visualizing these using Kiviat diagrams.
Appendix A User Manual

This chapter describes a user manual explaining the installation of the plug-ins and scripts, and how to use them for evaluating a Software Quality Model on an analyzed software system.

A.1 Installation

This section contains the installation instructions for the plug-ins and the script. It assumes that the VizzAnalyzer tool has been installed and is ready for use.

For installing the SQM and the statistics component the file QualityModel.class and Statistics.class have to be copied in the following folder of the VizzAnalyzer installation:

<inst-dir>\hlanalysis\analyses

The Metrics component script should be copied in the working directory of the VizzAnalyzer tool (<inst-dir>).

A.2 Evaluation of a Software Quality Model

For the evaluation of a software quality model on a graph three steps need to be performed, and three preconditions need to be fulfilled. First, it is necessary that the software system has been analyzed by the VizzAnalyzer tool, and the created graphs are available; second, a Software Quality Model needs to be defined in a QMXML file; third, the metrics required by the SQM need to be provided by the VizzAnalyzer tool. The three steps needed are: running the metrics script on the graph, calculating all metrics, applying the SQM on the graph, calculating a statistical overview (these are four steps). These steps shall be explained in the following sections.

A.2.1 Fulfilling the preconditions: Analyzing a Software System

Prior to applying a SQM to a software system, the SQM needs to be loaded into the VizzAnalyzer tool by retrieving its program information using one of the front-ends. After starting the VizzAnalyzer the tool is presented as seen in Figure A.1: VizzAnalyzer Program.
Using one available front-end, for instance the Recoder front-end, is started by selecting Recoder in the Frontends menu. In the following configuration dialog select a project configuration by pressing the Browse... button in the Recoder Config File section. Select one of the available projects, e.g., generic_grail_examples.vap. This analyses the grail graph library. The analysis is started by pressing the Ok button, and the graphs selected in Recoder Default Configuration are created. Compare Figure A.2: Loading Graphs.

The created graphs representing the program structure of the grail library are visible as the Data graph in the left hand section of the VizzAnalyzer main window. The precondition is satisfied.

A.2.2 Applying Metrics (Metrics script)
The next step is to prepare the application of the Software Quality Model by calculating all the metrics needed. The metrics are calculated by running the provided metrics script file. For doing this, select Run Script in the Script menu. In the opening file open dialog, the metrics script file Metrics.bsh has to be selected, as it is the one with the QMXML file. In the current example three metrics have been calculated: NOC, DIT and LOC. Their values are visible after the calculation in the main window, refer to Figure A.3: Metrics Result where it is possible to see two rows with label DIT and LOC presenting the results of the metrics calculated (NOC is not visible, since outside of the view).
A.2.3 Evaluating the Software Quality Model (Quality Model Plug-in)

After the metrics have been calculated the Software Quality Model can be evaluated on the selected graph. It is started by selecting in the Analyser menu, the sub menu Analyzer Analyses, and there the entry Quality Model Plug-in as shown in Figure A.4: Quality Model Plug-in Execution.

In the opening dialog (Figure A.5: QM Selection QMXML file) the graph on which the SQM shall be applied (the same as the metrics have been applied) is selected. Further the QMXML file needs to be specified with the full path. The evaluation of the SQM starts by pressing the OK button.
A.2 Evaluation of a Software Quality Model

The results of the SQM evaluation are added to the graph as is shown in Figure A.6: QM Result. The columns for each node have attached the QM properties described in the QMXML file and its own values.

A.2.4 Applying Equivalence Classes (preparing statistics)

For applying the statistics in a meaningful way to the processed graphs, the elements in the graph need to be grouped into equivalence classes. This is performed by using the CrocoPat plug-in provided by the VizzAnalyzer tool. Using this plug-in first involves...
selecting the run script controlling its actions, and then starting it. The Configuration
dialog is available in the Analyser menu, in the sub-menu Crocopath. The user selects
_Configuration Dialog_ as shown in Figure A.7: CrocoPat Configuration.

![Figure A.7: CrocoPat Configuration](image)

For our purpose the results need to be marked in the current graph. Therefore _Mark
result in graph_ is selected as shown in Figure A.8: CrocoPat Mark Results.

![Figure A.8: CrocoPat Mark Results](image)

To execute a script grouping the elements in the graph into equivalence classes the
analysis script is selected choosing _Analyzer → CrocoPat → Open analysis script_ as
shown below.

![Figure A.9: CrocoPat script](image)

In the file open dialog Figure A.10: EC Selection select _EC_Classes.pat_ to create a
group containing all classes in the system. The last step executes the CrocoPat script
selected. Run the script as shown in Figure A.11: Running EC Script accessing to
_Analyser → CrocoPat → Run analysis_ from the VAF menu. After the execution of the
script the graph selected has the grouping applied, which is visible in Figure A.11:
Running EC Script in the column _EC_Classes.pat_.

![Figure A.10: EC Selection](image)

![Figure A.11: Running EC Script](image)
A.2 Evaluation of a Software Quality Model

A.2.5 Calculating the statistics

After the execution of the statistics plug-in has been prepared in the previous step by grouping the elements in the graph, it can be run. Select in the Analyser menu, sub-menu Analyzer Analyses, the Statistics Plug-in entry as shown in Figure A.12: Statistics Plug-in Execution.
It is necessary to specify parameters for the execution of the plug-in as is shown in Figure A.13: Statistics Parameters. In particular, it is important to identify the name of the column (element property) containing the group identification. The statistical method will be applied on all elements having the same value in the group. The Data Property contains the name of the property from which the values are taken for input to the statistical method. And finally the statistical method has to be selected by entering its name as string: *Average*, *Minimum* or *Maximum*. The plug-in is executed by pressing the OK button.

The results of the operation are written back into the graph, creating a new element representing the group, having the result of the calculation as properties.
References


