Fraunhofer Institut
Experimentelles
Software Engineering

Design and Implementation of a customizable Metrics Plug-in in Eclipse

Authors:
Ansgar Lamersdorf,
Jens Knodel

IESE-Report No. 091.06/E
Version 1.0
July 24, 2006

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Fraunhofer-Platz 1
67663 Kaiserslautern

Abstract

To analyze the quality of a software system metrics can be used that measure attributes of the software's internal structure. However, all these metrics are limited to analyzing a certain aspect of software from a certain viewpoint only. Many metric tools exist that collect and visualize a standard set of software metrics. But since the actual suitable set of metrics depends on the viewpoint and purpose of the measurement or the person measuring, a metric tool should be customizable to enable its users to concentrate on and select only the set and configuration of metrics that are needed.

This Bachelor thesis presents the design and implementation of a customizable metric framework which allows such customization by enabling the user to implement own metric and then to add them to the framework. It delivers all commonalities of metrics like selection of the analyzed software elements, storing of the results, or visualization of the results. So only the specifics of every metric like the algorithm calculating the results have to be implemented by the user. The calculation of metrics can be delegated to external metric tools.

The developed framework is:

- Extensible: new metrics can easily be added to the framework
- Flexible: many different types of metrics are supported
- Adaptable: The measurement can be adapted to specific needs by specifying the analyzed elements and metric-specific options
- Customizable: the visualization can be customized to the user's needs by selecting the type of visualization and the set of results visualized
- Repeatable: both a calculation's configuration and results are stored persistent, so the calculation can be repeated and a single calculation's results can repeatedly be visualized from different viewpoints.

The framework was built as a plug-in of the software development platform Eclipse and is integrated into SAVE, a tool for Software Architecture Visualization and Evaluation. SAVE can extract and analyze the architecture of an existing software system but did not support the measurement of metrics. The implemented framework adds a mechanism to SAVE that enables the measurement of metrics. It can calculate, store the results of and visualize all metrics registered to it. The results of a metric's calculation can be visualized within different kinds of diagrams or within an existing architecture view offered by SAVE.

Copyright © Fraunhofer IESE 2006

Finally, in order to validate the metrics plug-in a set of metrics was implemented covering all aspects of the framework's functionality. Some metrics were as well implemented by other people to validate the understandability of the framework.

Keywords: architecture, architecture metrics, code metris, metrics framework, metrics visualization

Projektname: ArQuE
IESE Produktname: SAVE
Technologie: Java, Eclipse, EMF, BIRT

Copyright © Fraunhofer IESE 2006

Table of Contents

1	**Introduction**	**1**
1.1	Software Metrics	2
1.2	Using metrics	4
1.3	Visualizing metrics	5
1.4	Overview of this bachelor thesis	6
2	**Related work**	**8**
2.1	Finding metrics	8
2.2	Calculating metrics	10
2.3	Presenting metrics	12
2.3.1	Within an architecture visualization	12
2.3.2	Within separate diagrams	14
3	**Background**	**18**
3.1	Eclipse, EMF, and BIRT	18
3.1.1	Eclipse Platform	18
3.1.2	EMF	19
3.1.3	BIRT	20
3.1.4	GEF	20
3.2	SAVE	20
3.2.1	Functionality of SAVE	21
3.2.2	Architecture of SAVE	21
4	**Designing a customizable Metric plug-in for SAVE**	**23**
4.1	A “customizable” metric framework	24
4.2	Classification of metrics	24
4.3	Registering metrics	27
4.4	Collecting metrics	28
4.5	Visualizing metrics	30
4.5.1	Visualization within SAVE	31
4.5.2	Visualization using diagrams	32
4.5.3	Single metric diagrams	32
4.5.4	Diagrams of a multiple metrics	35
5	**Architecture and Implementation of the Metric Plug-in**	**38**
5.1	Architectural views	38
5.1.1	Conceptual view	38
5.1.2	Structural view	40
5.1.3	Behavioral view	46

Copyright © Fraunhofer IESE 2006

5.2 Implementation 49
5.2.1 Metrics model 50
5.2.2 Wizards for calculation and visualization 50
5.2.3 Metric diagrams 53
5.2.4 Visualization within SAVE view 53
5.3 Extensibility 54
5.3.1 Adding a new diagram 55
5.3.2 Adding a new aggregation rule 55
5.3.3 Adding a new type of result 55

6 Implemented metrics 56
6.1 Equivalence classes 56
6.2 List of metrics 57
6.2.1 McCabe 57
6.2.2 LOC 58
6.2.3 Fan-Out 59
6.2.4 Ownership 59
6.2.5 Dependency cycles 60
6.2.6 Hierarchy depth 61
6.2.7 Best mach 61
6.2.8 Fan-Out to library 62

7 Conclusion 64
7.1 Summary 64
7.2 Future work 65

8 Literature 67

9 Appendix 70
9.1 How to add a new metric to the metrics plug-in 70
9.1.1 Calculator class (required) 70
9.1.2 Registration at extension point (required) 71
9.1.3 Wizard page (optional) 71
9.1.4 Results with list of components (optional) 72
9.1.5 Metric initialization (optional) 72
9.1.6 Setting optimum and tolerable values (optional) 72
9.1.7 Using an external metric (optional) 73

Copyright © Fraunhofer IESE 2006

1 Introduction

One important software engineering practice, like in all engineering disciplines, is the practice of *measurement*. On the one hand, processes and methods have to be measured to empirical validate their adequacy and effectiveness. On the other hand the outcome and intermediate artifacts of the process - the software itself - have to be measured, too. By measurement the quality of software can be determined in order to verify a given quality standard or to find parts of the software that should be refactored for improving the quality.

There exist many different kinds of quality attributes that can be measured at a given software component. In general they can be classified into two groups: External, dynamic attributes, like its reliability or changeability, and internal, static attributes, like its size or complexity. External attributes are more difficult to measure than internal ones since they are harder to define (the definition of reliability for example is much more complex than the definition of size) and they can only be measured after the implementation process. Internal attributes, on the other hand, can easily be refined into a set of metrics (see section 1.1) that assign a defined value to each analyzed part of the software. These metrics can already be measured during the implementation phase (if they are based on source code elements) or even during design (if they are based on the software architecture) and can easily be collected by automated tools.

However, in the end not the internal attributes but the external ones are important. The user of a software product does not care about its size or complexity as long it is reliable and easy to adapt. But internal attributes can be used as indirect measures to draw conclusions about the external quality. The complexity of a component for instance can predict its changeability – complex components are hard to understand for any developer who wants to implement changes, so their changeability is low. Thus, by analyzing the complexity of all components of a software product and refactoring the ones with the highest complexity, the changeability and the quality of the product can be improved.

Many software tools exist that support the automatic collection and presentation of software metrics. But most of them have one characteristic in common: They have a certain fixed predefined set of metrics they measure. However, the metrics needed to analyze a piece of software vary depending on the stakeholder performing the analysis. Every stakeholder may have a different viewpoint or purpose of analyzing. So an ideal metrics tool should be extensible, flexible, and adaptable to the stakeholder's needs.

Copyright © Fraunhofer IESE 2006

It is the goal of this bachelor thesis to design and develop such a metrics framework for measuring and visualizing a set of metrics independent of their implementation. It has to be:

- extensible: The set of metrics measured by the framework is not fixed but can easily be extended by new metrics. These metrics can be taken from other existing tools.
- flexible: The tool supports many types of metrics and many ways of presenting the results so it can be used for different purposes.
- adaptable: The analyzed parts of the software, the set of metrics used for a single examination, or rules for the calculation of a metric can be adapted to the context of the examination.
- customizable: The important aspects of an analysis, like the kind of diagram visualizing the results, can be specified, according to the user's needs.
- repeatable: An examination's configuration and results are made persistent to make the examination repeatable and for later refinement of the visualization due to possibly changed viewpoints

The Fraunhofer Institute of Empirical Software Engineering (IESE) developed a tool to statically visualize and evaluate the architecture of an existing software system (Software Architecture Visualization and Evaluation – "SAVE", see chapter 3.2). However, the use of software metrics is up to now not supported in this tool. SAVE is written as an Eclipse (see 3.1) plug-in so it can easily be extended by other plug-ins. The metrics framework developed in this thesis was realized and integrated as a new plug-in into SAVE to enable the measurement of metrics on an architectural level and on source code elements.

1.1 Software Metrics

[Fenton & Pfleeger 97] define the measurement of metrics as "the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules". According to this definition, there are many different kinds of metrics in software engineering. They can be distinguished by the attribute they measure: Some measurements collect metrics of the development process, e.g. the number of person-months needed to develop a product, the number of lines of code a developer produces, or the average team size. Others measure external attributes of the software product like the number of defects or the time the system needs to perform a specific task. Yet another class of metrics are metrics of a product's internal structure like the number of lines of code or the coupling between components. This thesis develops a framework to analyze this class, measuring structural properties of components.

Copyright © Fraunhofer IESE 2006

Regarding the scale of their value, metrics are classified into five different types:

- *Nominal scale*: The results of the measurements are categorized in different classes without any ordering between the classes. An example is the categorization of a defect into a specific phase of the development process where it happened: requirements, design, or implementation.
- *Ordinal scale*: The results are categorized into different classes with an ordering between them. For example the classification of a component into classes regarding its complexity (very simple, simple, medium, complex, very complex) would be an ordinal scale.
- *Interval scale*: A measurement with an interval scale produces a numeric value. Results of this metric can be compared regarding the difference between two values (subtraction is possible), but since no zero-number exists, ratios can not be compared (division is not possible). An example of an interval scale is the temperature in Celsius or Fahrenheit or the creation date of a document.
- *Ratio scale*: Additional to an interval scale, in a ratio scale there exist a zero-element. So we can compare two numbers by saying how many times on number is bigger as the other (division is possible). The length of a component in lines of code for instance would be a ratio scale.
- *Absolute scale*: Numbers in an absolute scale always come from counting elements. All mathematical operations comparing two elements are allowed. Examples of an absolute scale are the numbers of components in a class or the numbers of defects found.

Other classifications of metrics (as in [Fenton & Pfleeger 97]) divide them into *subjective* and *objective* metrics, regarding the fact if the value of a measurement is dependent on the person measuring it or not, or into direct and indirect metrics.

- *Direct metrics* measure only one single attribute without a relation to any other attribute. An example is the size of a component or the duration of a development process step.
- *Indirect metrics* can not be measured directly from a single attribute. They have to be calculated using a combination of other metrics. For example the efficiency of a programmer can be calculated by the lines of code produced divided by the effort needed.

A tool that automatically collects, calculates, and visualizes software metrics like the one developed in this bachelor thesis obviously shall only perform objective measurements since they are collected without anybody's influence. Direct metrics can easily be collected by the tool – only one single attribute has to be regarded. However, the measurement of indirect metrics has to be considered as

Copyright © Fraunhofer IESE 2006

well. For the measurement of an indirect metric, two ways are possible: One way is to only measure direct metrics and give the users the possibility to combine the visualization of different direct metrics so they can relate the results to get the indirect metrics their self. Another way is to deliver indirect metrics by measuring different attributes of the software, performing defined calculations on the results and visualizing the result.

1.2 Using metrics

There are many different metrics that can be measured, even within a specific class of metrics: An internal attribute of a software product, like for example the complexity, can be measured in many different ways. Some metrics measure, for instance, the complexity of code by calculating the cyclomatic complexity as defined by McCabe; others measure complexity by counting the relations between components (Fan-In, Fan-Out). For every purpose and every viewpoint, some metrics are suitable, others are not. So, everybody who wants to measure attributes of a software product, first has to define exactly what metrics should be measured and how they are defined.

As described above, measurements of internal attributes of software can be used to draw conclusions about external quality attributes. But drawing conclusions between internal and external attributes of a software product is not always so easy and can not be done automatically since there are many exceptions. For example, high complexity of some components doesn't always mean poor quality. The use of a design pattern like the mediator pattern [see GOF 1995, p. 273] improves the quality and the changeability of software but leads to some components, the *mediators*, that are highly connected to other components and by that have a very high complexity, if complexity is for instance measured by the fan-in or fan-out metric. In this case, there is no need to refactor them. Other components that aren't mediators should not have a high coupling between each other and should be refactored if their fan-in or fan-out is very high.

Deming (see [Deming 86]) suggests a four-step cycle for analysis and improvement. It consists of the steps plan, do, check, and act. Applied to the process of taking metrics of a software product's internal structure, the following four steps arouse:

1. Plan: Definition of the goals of the measurement process and selection and definition of adequate metrics. This can be done using the GQM approach (see 2.1). It includes the configuration of the chosen metrics if they are configurable.
2. Do: Collection and calculation of the chosen metrics.

Copyright © Fraunhofer IESE 2006

3. Check: Analysis of the result; drawing of conclusions about the products external qualities; identification of components that need to be refactored in order to improve quality. To analyze the results an adequate visualization of the values must be selected in this step.
4. Act: Refactoring of the components identified in the previous step.

Only the second step, the collection and calculation of metrics, can be done completely automatically by a software tool – as it is done by many existing tools. The first step can be supported by delivering a set of predefined parameterized metrics and specifying their options; the third step can be supported by giving different types of visualization of the measurement's results to help the user in drawing conclusions and by presenting components with unusual values of the metrics. The fourth step can not be supported by a tool as it is implemented in this thesis and therefore will not be discussed in detail.

In the context of the SAVE tool, the plug-in developed in this bachelor thesis provides a generic framework for the second step and supports the first and the third step of the process as described above.

1.3 Visualizing metrics

There are different reasons why the visualization of metrics is of a great importance:

1. Overview: Even for small software products, detailed measurement can gain a huge amount of results. Visualization of metrics helps getting an overview of the results.
2. Filtering: The results can be filtered to help the user focusing on relevant values
3. Conclusions: As described in the previous part, the result of a measurement of products internal attributes is used to draw conclusions about its external quality. A good visualization of the metrics can be very supportive during this step.

Each one of these aspects has to be considered during the design of measurements' visualizations.

To give an overview metrics can be presented in many different ways: For instance, *bar charts* show the absolute values of the results and give an overview of the relation between two results; *pie charts* show a result as part of a whole; *box plots* can give additional statistical information; *stock* and *line charts* can show a results development over a period of time. Each chart and diagram is

Copyright © Fraunhofer IESE 2006

suitable for different kinds of metrics and focuses on different aspects respectively.

The filtering of a measurement's results for its visualization is of great importance, too. A complex software product may typically consist of thousands of components. If a metric assigns a value to every component, the presentation of all results in one single diagram or chart would lead to an enormous, blurred image where nearly no information can be extracted. Various possibilities exist to filter such results in order to improve readability: Values like median and the standard deviation can give an overview of all results; only the top or bottom values can be shown to focus on unusual components; customizable filters can give the user the potential to concentrate only on selected components.

[Rook & Lippert 04] suggest a filtering that only shows the values that were "bad" (very high or very low) the last time and have become worse. They describe how, with an adequate filtering, the time for analyzing and interpreting metric values can be reduced from several hours to only half an hour.

The possibility to draw conclusions from a metric's visualization depends on the aspects described before. To successfully draw conclusions about parts of the software it must be possible to get different types of visualizations in order to have different viewpoints and to focus on relevant values. Another aspect that has to be considered in this case is the traceability: Only if the identified relevant results can clearly be traced to the belonging parts of the software, these parts can be identified as well.

For a single metric, many different ways of presenting the results are possible. The chosen diagram and abstraction level are dependent on the viewpoint and the purpose of the measurement. If users, for instance, want to find out components that need to be refactored using a specific metric they first might want to have a look at statistical information like the median of the results to get an overview. Then they could look at the top five values to identify some components that probably need refactoring. Finally, by regarding the values of different metrics for these selected components they could get further information helping them to decide what kind of refactoring needs to be done.

This example shows that the visualization of metrics in a software tool has to be very flexible and should support an easy selection between different kinds of visualization and filtering of the results.

1.4 Overview of this bachelor thesis

This section gave an introduction to metrics and some important facts of the measurements of metrics.

Copyright © Fraunhofer IESE 2006

The second chapter describes the related work. It consists of a review of work on finding adequate metrics, existing tools for the calculation of metrics, and tools and ideas for the visualization of metrics' results.

Chapter 3 explains the background of the implemented metrics plug-in. Here used technologies and the SAVE tool on which the plug-in to e developed is based on are introduced.

The next two chapters present the metrics plug-in developed in this thesis. Chapter 4 focuses on the functionality of the plug-in. The process of registering, calculating and visualizing a metric is defined and explained here as well. Chapter 5 then discusses the architecture and the implementation of the plug-in. Its main components and decomposition of these components are presented. After that the important solutions of the implementation are discussed in detail.

In order to validate the metrics plug-in, a set of metrics that cover the entire metrics framework's functionality was implemented. These metrics are listed chapter 6. The experiences of other people implementing metrics for the framework are also explained here.

Finally the last chapter summarizes the thesis and addresses some open issues and future work.

Copyright © Fraunhofer IESE 2006

2 Related work

Following the three regarded steps (plan, do, check) of the previously defined four steps of the Deming cycle in using metrics at a software product (see 1.2), this chapter is divided into three parts: the selection of useful metrics, the calculation, and the presentation of the metrics. Each part reviews the respective related work.

2.1 Finding metrics

A general top-down approach for finding metrics for specific goals is introduced with the Goal/Question/Metric paradigm by [Basili & Rombach 88]. They suggest a formal mechanism for planning measurement process. The process starts with defined

- Goals that are refined into a set of
- Questions which specify the
- Metrics that have to be measured.

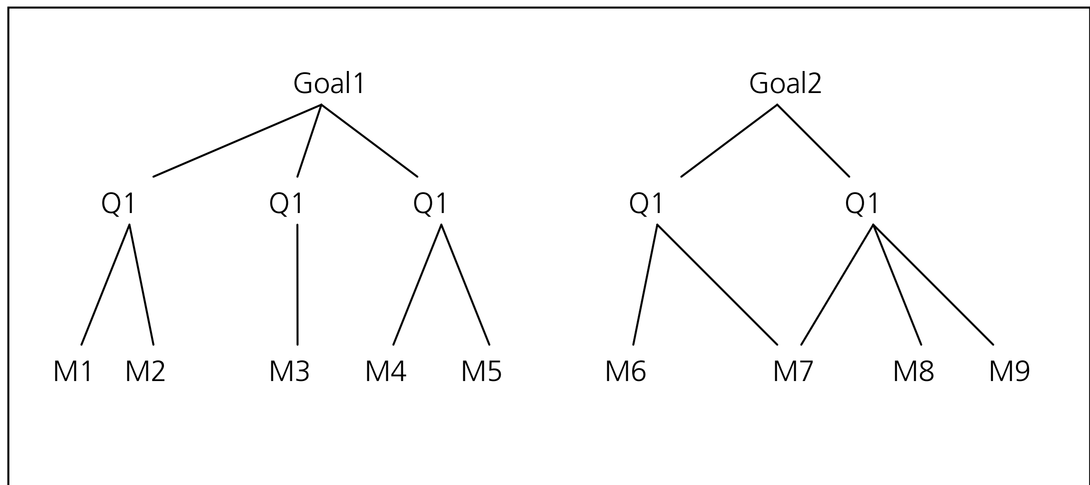

Figure 2.1: Refinement from goals into question and question into metrics

This method reduces the risk of taking the effort to calculate metrics without having a reason for using them. Every metric that is calculated is reasoned before use, because there exists a defined question this metric shall answer and

Copyright © Fraunhofer IESE 2006

for this question in turn a goal exists that is reached by answering this question. Thus, all metrics are validated before use.

On the other hand every goal is refined into a set of metrics; therefore no goal can be left out during the measurement process.

For the identification of goals templates are presented showing important attributes of the measurement process like its purpose and the quality attribute it focuses on.

[Grady 94] gives four major goals for the use of software metrics:

- Project estimation and process monitoring,
- Evaluation of work products,
- Process improvement through failure analysis, and
- Experimental validation of best practices

He points out that the use of metrics for experimental validation is the most successful one because it leads to quicker, widespread acceptance of best practices. For the goal of evaluation of work products (the use of metrics this thesis focuses on) he presents an example of a measurement process at Hewlett-Packard and suggests the use of the cyclomatic complexity metric at control-oriented applications. However, for the evaluation of other applications, for instance data-oriented ones, a broader set of metrics should be used. For design evaluation he suggests the fan-out squared metric because it correlates well to the probability of defects and can be calculated even before writing the code.

[Stark & Durst 94] describe how software metrics are used at NASA for the goal of identifying potential risk areas. They applied the cyclomatic complexity metric as well because it was successfully used at a number of projects at the Software Engineering Laboratory at NASA's Goddard Space Flight Center and the Johnson Space Center.

For object-oriented software systems, different metrics are needed than for systems written in procedural languages. [Rosenberg 98] describes a set of useful object-oriented metrics. They consist of traditional metrics that are adapted into an object-oriented environment, and of new object-oriented metrics.

[Martin 97] introduces the concept of "stability" for object oriented software. He suggests that a system should contain stable, abstract and instable, concrete packages. Instable packages should depend on more stable packages. For the measurement of stability and abstraction, he gives a set of metrics.

In [Lanning & Koshgoftaar 94], another way of finding adequate metrics is described. They use a correlation model to analyze the relationship between the

Copyright © Fraunhofer IESE 2006

values of different metrics for the goal of measuring complexity (for instance the number of operators, the number of statements, or the fan-out metric) and maintenance difficulty, described by another set of metrics (like the number of lines added / deleted or the number of program faults).

Nearly all papers and books on choosing adequate metrics show two different aspects:

- There is no perfect metric that can make correct predictions about a quality attribute of a software product at any time. The metric most suitable depends on various facts and has to be specified from case to case.
- However, there exists a set of "standard" metrics like lines of code, the fan-out metric, or the cyclomatic complexity. These metrics are used in most of today's software projects because they are clearly defined, easy to calculate, and make clear statements about the software to be analyzed.

The metric plug-in has to consider both aspects: On the one hand, most of the standard metrics must be implemented so they can be used without much effort. On the other hand, it must be possible to easily add a new metric to the plug-in if none of the implemented metrics is suitable for a given goal. Therefore, a requirement for the plug-in is having a comfortable interface to implement and add new metrics.

2.2 Calculating metrics

For the calculation of metrics, a broad number of tools exist. Since this bachelor thesis mainly focuses on the integration of a generic metric-framework into the SAVE (see chapter 1) and not just on the implementation of a set of metrics, the calculation of metrics can be done by an existing tool. The SAVE environment requires the fulfillment of three conditions:

1. The tool should be freely available. A list of commercial products can be found at [Fetcke 05]. But since the SAVE tool is free and shall not be dependant on commercial products, every used tool must be freeware, too.
2. The tool should support the programming languages supported by the SAVE tool. These are mainly Java, C, and C++ and besides Delphi.
3. The tool has to be able to deliver its data to the metrics plug-in. Two possibilities exist here: The tool could be written in Java with its source code available so that it can be imported into the SAVE environment and can be used automatically. Another way could be that the tool has the ability to save its data into a file where it can be read out by the metrics plug-in.

Copyright © Fraunhofer IESE 2006

[JMetric 00] is an analyzer that collects a great amount of metrics. It is under general public license so the source code is available. The tool displays the results in a Java-Swing user interface but the values can as well be saved into files. Since the tools architecture follows the model/view/controller-pattern it would also be possible to separate the core from the visualization and integrate it into the metrics plug-in. But since only software systems implemented in Java can be analyzed, JMetric is not sufficient.

[Metrics 06] is a tool for the collection and interpretation of metrics. It is written as an eclipse plug-in therefore eclipse-projects can easily be analyzed. New own metrics can be written and added via the eclipse extension point mechanism (the same mechanism is being developed in this bachelor thesis). Additional to the presentation of metrics, the architecture of a software system is visualized as well within separate views. However, since it has its own visualization there is no need to integrate it into the SAVE environment. Besides that, the plug-in again only analyses software implemented in Java and does not support C/C++.

Metrics - /net.sourceforge.metrics

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Packages	16					
Number of Methods (avg/max per type)	1310	6.65	8.553	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
tgsrc	489	7.191	11.544	76	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
src	761	6.238	6.553	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core.sources	108	15.429	12.129	45	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui	77	9.625	10.111	33	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.core	198	6.6	7.093	27	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.preferences	52	6.5	7.467	26	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.ui.dependencies	95	5.588	3.727	15	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.persistence	18	4.5	4.33	12	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler.implementa...	54	5.4	2.871	10	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.xml	41	4.1	2.022	9	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.calculators	79	4.158	2.254	8	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.propagators	31	5.167	1.067	7	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.tests	8	2.667	1.886	4	/net.sourceforge.metrics/src/net/sourceforge/...	
net.sourceforge.metrics.internal.prevayler	0	0	0			
classycle	60	8.571	2.556	13	/net.sourceforge.metrics/classycle/classycle/g...	
Lines of Code (avg/max per type)	6593	33.467	49.02	339	/net.sourceforge.metrics/tgsrc/com/touchgrap...	
Number of Interfaces (avg/max per packageFragment)	16	1	1.414	4	/net.sourceforge.metrics/src/net/sourceforge/...	
Lines of Code (avg/max per method)	6593	4.812	7.355	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
classycle	324	5.4	9.94	69	/net.sourceforge.metrics/classycle/classycle/g...	calculateAttributes
tgsrc	2321	4.661	8.278	59	/net.sourceforge.metrics/tgsrc/com/touchgrap...	scrollSelectPanel
src	3948	4.862	6.473	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
net.sourceforge.metrics.ui	544	6.8	8.707	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable.java	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
MetricsTable	194	10.778	13.831	52	/net.sourceforge.metrics/src/net/sourceforge/...	setMetrics
setMetrics	52					

Figure 2.2: The eclipse metrics plug-in (from [Metrics 06])

[CCCC 06] analyzes both Java and C/C++ files. It is not written in Java; therefore it cannot be integrated directly into Eclipse but the results can be stored into XML-files that could be read out. The tool is under public license so it can be freely used.

Copyright © Fraunhofer IESE 2006

After an overview over the existing metrics tools it was decided for this thesis to use CCCC as an external tool. The integration of the tool's data into the metrics plug-in can be achieved by using the XML file schema delivered by the tool. The implemented metric McCabe (see chapter 6.2.1) uses the CCCC tool for the calculation of its values.

2.3 Presenting metrics

For the presentation of the metrics' values there exist two general ways: The metrics can be visualized within an existing architecture visualization (like SAVE's) or by using separate diagrams. Both possibilities are discussed in various works

2.3.1 Within an architecture visualization

[Termeer et al. 05] developed a combination of presenting an UML-diagram together with metric information. They used overlay images to show the metric values and implemented many different ways of visualizing them. The values can be represented two-dimensionally by a set of objects the user can select from: rectangles (the value is shown via the color), bars (height), circles (radius), or pies (arc). Another way of presenting the metrics is by three-dimensional bars or cylinders where the height reflects the value.

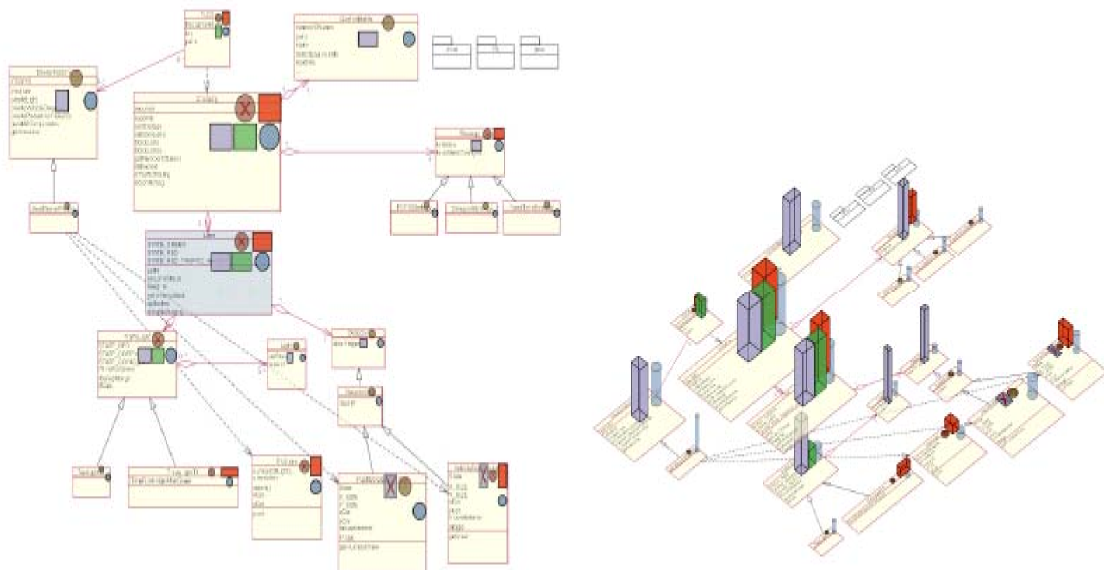

Figure 2.3: Metrics visualized 2- and 3-dimensional (from [Termeer et al. 05])

Copyright © Fraunhofer IESE 2006

[Lanza 03] suggests another possibility of adding metric information to an architecture diagram. He creates a "polymetric view" by settling the values of up to five attributes of a component's graphical representation on the results of metrics: Its position (from top and from left), size (height and width), and its color.

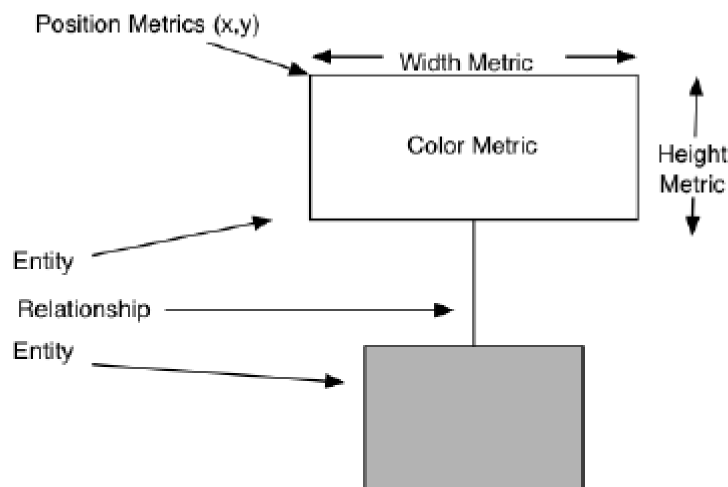

Figure 2.4: Five metrics shown at once (from [Lanza 03])

Not all five attributes can be used easily in an architectural visualization since a components position is determined by the relation to and the position of other components but at least three metrics can simply be visualized.

[Naab 05] already implemented a visualization of metrics into the SAVE tool. For a single metric, a threshold can be defined. Components with metric values above this threshold appear different to components with values below. The user can choose the visual appearance and color of both types of components.

Copyright © Fraunhofer IESE 2006

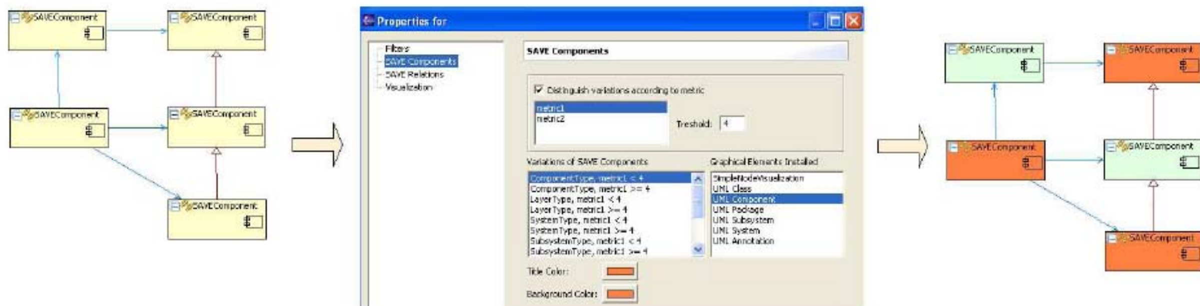

Figure 2.5: Appearance distinction according to metric values (from [Naab 05])

This visualization is only able to presents one single metric and two options (below or above) and is revised in this bachelor thesis.

2.3.2 Within separate diagrams

[Stark & Durst 94] propose a way of visualizing the cyclomatic complexity metric of a set of systems using a line chart. The x-axis shows the complexity in a logarithmic scale, the y-axis gives the percentage of modules in a system that has at most this complexity.

Copyright © Fraunhofer IESE 2006

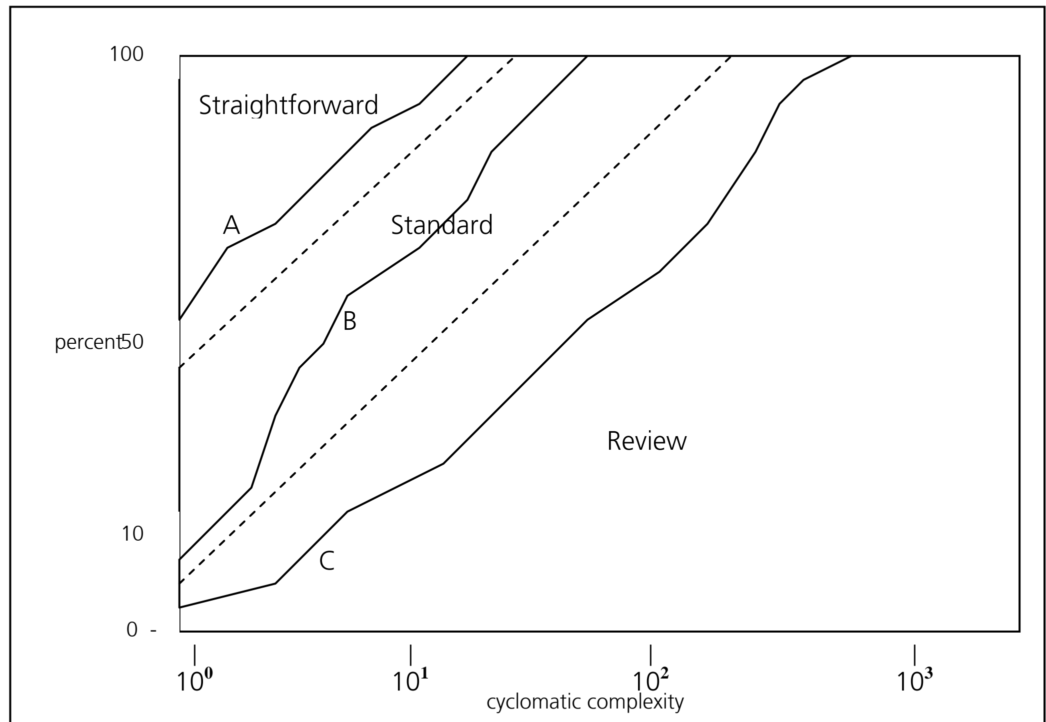

Figure 2.6: Complexity of software systems (from [Stark & Durst 94])

The diagram is divided into four parts, each of them standing for a different kind of system. Systems presented in the upper left corner, like system A, have a simple, straight forward logic; systems in the middle area, in this case B, have a standard complexity, and systems on the right-hand side, in this example System C, are very complex and should be reviewed.

[Pfleeger et al. 92] use a "multiple metrics graph", a graph similar to a Kiviat diagram, for the presentation of multiple metrics at once. A Kiviat diagram consists of a circle which is divided into a number of same-sized slices. At each line between two sections a value can be visualized: the farer the distance from the centre of the circle the higher the value of the metric. The maximum value of the metric is represented by the line of the outer circle.

Copyright © Fraunhofer IESE 2006

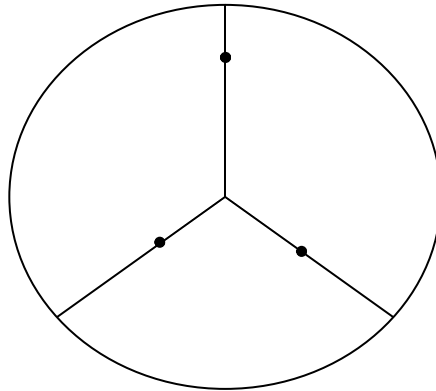

Figure 2.7: A Kiviat diagram showing 3 metrics

At a "multiple metrics graph" each metric is not visualized as a point at the line between the slices but in the middle of every segment. Each sections size reflects the importance of the metric: metrics of high importance lie within big sectors. To every metric a goal value can be specified which is represented by a smaller circle in the centre. Finally, lines are drawn form the points representing the metric values to the intersection between the goals arc and the edges of the slice. This representation creates a polygon whose size directly reflects the degree of accordance between the components performance and the goals set.

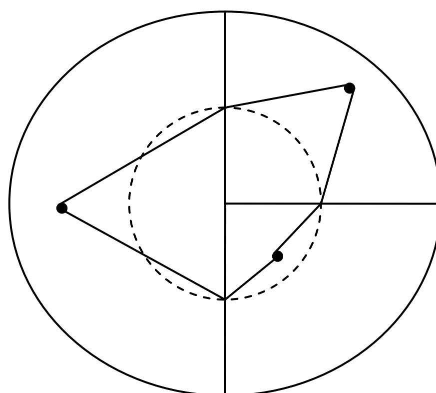

Figure 2.8: A multiple metrics graph showing 3 metrics. One metric is within goals

Copyright © Fraunhofer IESE 2006

[Lanza et al. 05] use a Kiviat diagram for the presentation of multiple metrics, too. Their diagram shows the evolution of a component by visualizing the metric values of different versions of the component at once. Each version has a different color, so the growth and the reduction of the components characteristics can be observed.

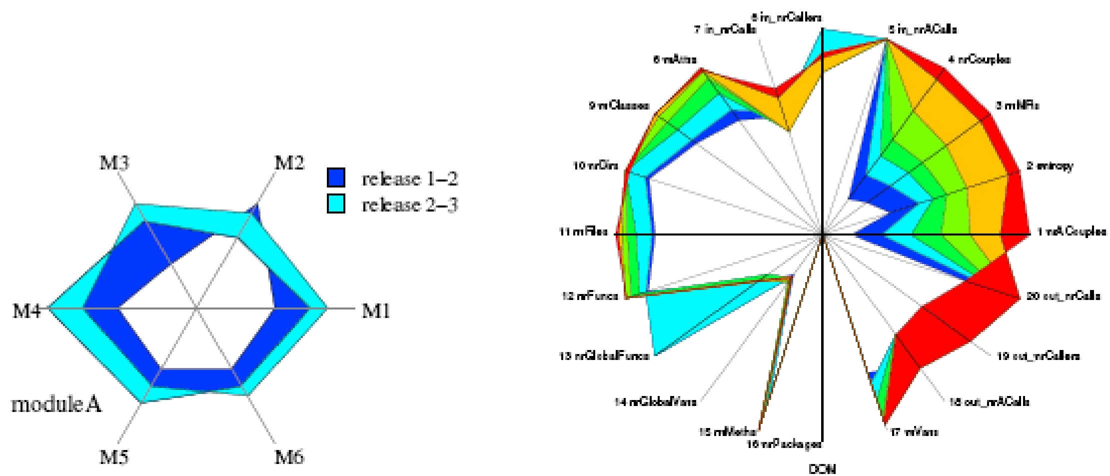

Fig 2.9: Evolution of component metrics ([from Lanza et al. 05])

These diagrams can provide much information at once. The right part of figure 2.9, for instance, presents the values of 20 metrics in 7 different versions. It can be seen very quickly that metric no.1 has grown a lot between these versions, whereas the values of metric 14 nearly stayed the same. On the other hand, if too many metrics and versions are illustrated at once, the benefits draw back due to the great complexity.

Copyright © Fraunhofer IESE 2006

3 Background

The metric plug-in developed in this bachelor thesis is based on a collection of software and analysis plug-ins. These plug-ins together form the SAVE tool developed by the Fraunhofer IESE. SAVE is plugged into the “Eclipse” platform using its extension point mechanism. Both, the metric plug-in and SAVE, use other open source plug-ins that extend the functionality of Eclipse.

This chapter gives an overview to Eclipse and the corresponding plug-ins the metric plug-in is based upon. It is dived into two parts: On one hand the open source tools needed for the operation of SAVE and the metrics plug-in – Eclipse and some of its plug-ins – are described, on the other hand the plug-ins developed by Fraunhofer IESE – the SAVE tool – are explained in more detail.

3.1 Eclipse, EMF, and BIRT

The Eclipse Foundation is a non-profit organization which hosts several different Eclipse projects. Eclipse is released under the “Eclipse Public License” which means that the source code of Eclipse is freely available. Four Eclipse projects are used for the development of the metric tool: the Eclipse platform, the Eclipse Modeling Framework (EMF), the Eclipse Business and Intelligence Reporting Tools Project (BIRT), and the Graphical Editing Framework (GEF).

3.1.1 Eclipse Platform

The main concept of Eclipse is the plug-in concept: The Eclipse platform consists of a small runtime environment, the Platform Runtime, and a set of plug-ins which are added to the Platform Runtime and extend its functionality. The basic architecture of the Eclipse platform is presented in the following picture:

Copyright © Fraunhofer IESE 2006

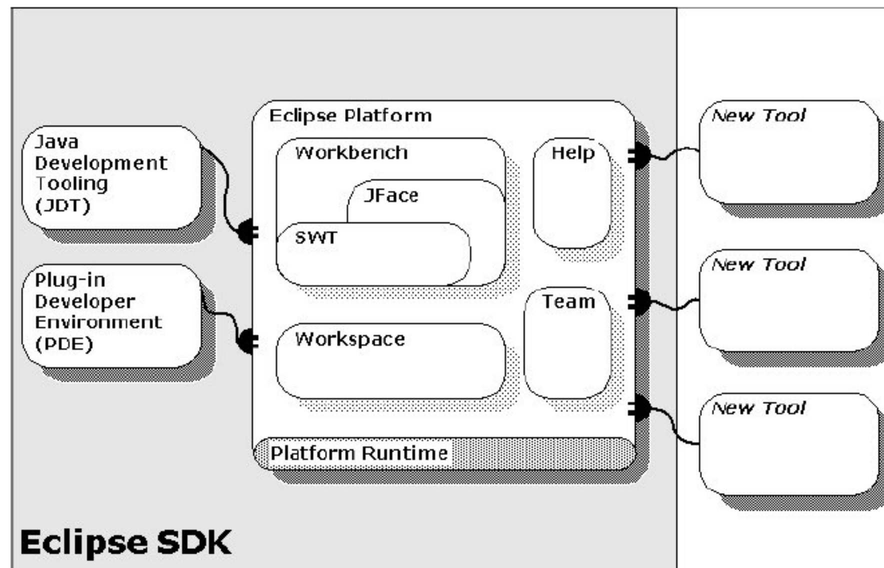

Figure 3.1: The Eclipse SDK and the Eclipse Platform (from [Eclipse06])

All parts of the architecture, except the Platform Runtime, are written as plug-ins. A plug-in can be distributed separately form other plug-ins or the platform. Plug-ins can use other plug-ins and add functionality to the Eclipse Platform. For the integration of plug-ins, Eclipse uses the "Extension Point" mechanism: Eclipse offers a set of points where new plug-ins can contribute to its functionality like for instance add an entry into a context menu. To use an extension point, a plug-in has to register at it using a specific schema that is defined by the extension point. Every plug-in can again define own extension points where other plug-ins can register at.

The Eclipse Platform offers a small set of editor functions. Its main plug-in, the workbench, contains the Standard Windowing Toolkit (SWT) and JFace, two frameworks that provide a set of Graphical User Interface (GUI) elements like wizards, dialogs and tree navigators.

The Eclipse Platform together with two additional plug-ins, the Java Development Tooling (JDT) and the Plug-in Developer Environment (PDE), are called the *Eclipse Standard Development Kit* (SDK). The Eclipse SDK can be used to develop new Eclipse plug-ins as done with the SAVE tool and the metric plug-in.

3.1.2 EMF

The *Eclipse Modeling Framework* (EMF, see [EMF 06]) is a framework that supports the generation and handling of a structured data model in java based on a specification. Different ways exist for the specification, like, for instance, an-

Copyright © Fraunhofer IESE 2006

notated Java-interfaces, XML schemata, or modeling tools like Rational Rose. The data model of the metric framework is specified using annotated interfaces: Every entity of the model is specified by a Java interface. The interface is marked as being part of the model by using special javadoc-annotations.

Based on such a specification, EMF can create Java classes representing the data model and a set of utility classes like factories for the creation of the data model, and item providers that support the visualization of the model. EMF provides as well a persistence mechanism that can store a model into xml files.

3.1.3 BIRT

The *Business Intelligence and Reporting Tools* (BIRT, see [BIRT 06]) support the creation of complex business reports and the visualization of the reports. Reports can be visualized with a suite of charts: bar charts, line charts, pie charts, scatter charts, and stock charts. BIRT has many functions to handle business reports, like access to the reports data, export of charts into PDF files, or the presentation of reports via the web but the metric tool only uses the charting engine of BIRT.

3.1.4 GEF

The *Graphical Editing Framework* (GEF, see [GEF 06]) offers support for creating a view and a graphical editor for a given model. The created view can be shown within an Eclipse view.

GEF was used to implement the architectural views at SAVE. The metric tool adds the visualization of metric values to the architectural view (see chapter 4.5.1) and uses the implementation of the SAVE view for this. Therefore, the metrics plug-in developed here uses GEF indirectly for its visualization.

3.2 SAVE

The *Software Architecture Visualization and Evaluation* tool (SAVE) is a tool that helps analyzing the architecture of a software product. It is controlled and developed by the Fraunhofer Institute of Experimental Software Engineering. The initial version of SAVE was developed by Miodonski in 2004 [Miodonski et al 04]. Naab [Naab 05] added a new graphical interface and Knieling [Knieling 06] extended SAVE with a trend analyzer component. Rost [Rost 06] gives a detailed description of SAVE and the data model of SAVE.

Copyright © Fraunhofer IESE 2006

3.2.1 Functionality of SAVE

The static architecture of an existing software system can be extracted from the source code using the SAVE tool. It is possible to parse source code in the languages Java, C/C++, and Delphi. The analyzed system can be displayed at the visualization component of the SAVE tool in a representation similar to UML (see [Naab 05]). A software architecture can be evaluated against a reference architecture to detect where deviations from the intended architecture are. Apart from that, the visualization can also be used to browse the architecture that was extracted from an existing software system. As highly abstracted views can be offered, an experienced architect can quickly see where problems in the architecture could arise. The trend analysis component adds the functionality for comparing different states of an architecture over time to SAVE (see [Knieling 06]).

3.2.2 Architecture of SAVE

From the architectural point of view, SAVE is not only a singular plug-in but consists of a collection of different plug-ins using each other. The core data model is an internal representation of extracted software architecture (a so called "SavePackage").

The core model consists of two different hierarchies: On one hand the "FSModel" stores information about the file system of the analyzed software: A package can consist of different folders which themselves contain other folders or compilation units (procedural or object-oriented). A compilation unit contains types (if it is object-oriented) or routines and variables. The SAVE-model, on the other hand, represents the abstract architectural view on the software. It consists of components (which can contain other components) and relations between them.

The mapping between the FS and the SAVE model is achieved by the SAVEFSConnector. It consists of connection between elements of the SAVE model and its representative at the FS model. FS model, SAVE model and connector are together contained in a "SAVEPackage". The core model of SAVE is shown in the following picture:

Copyright © Fraunhofer IESE 2006

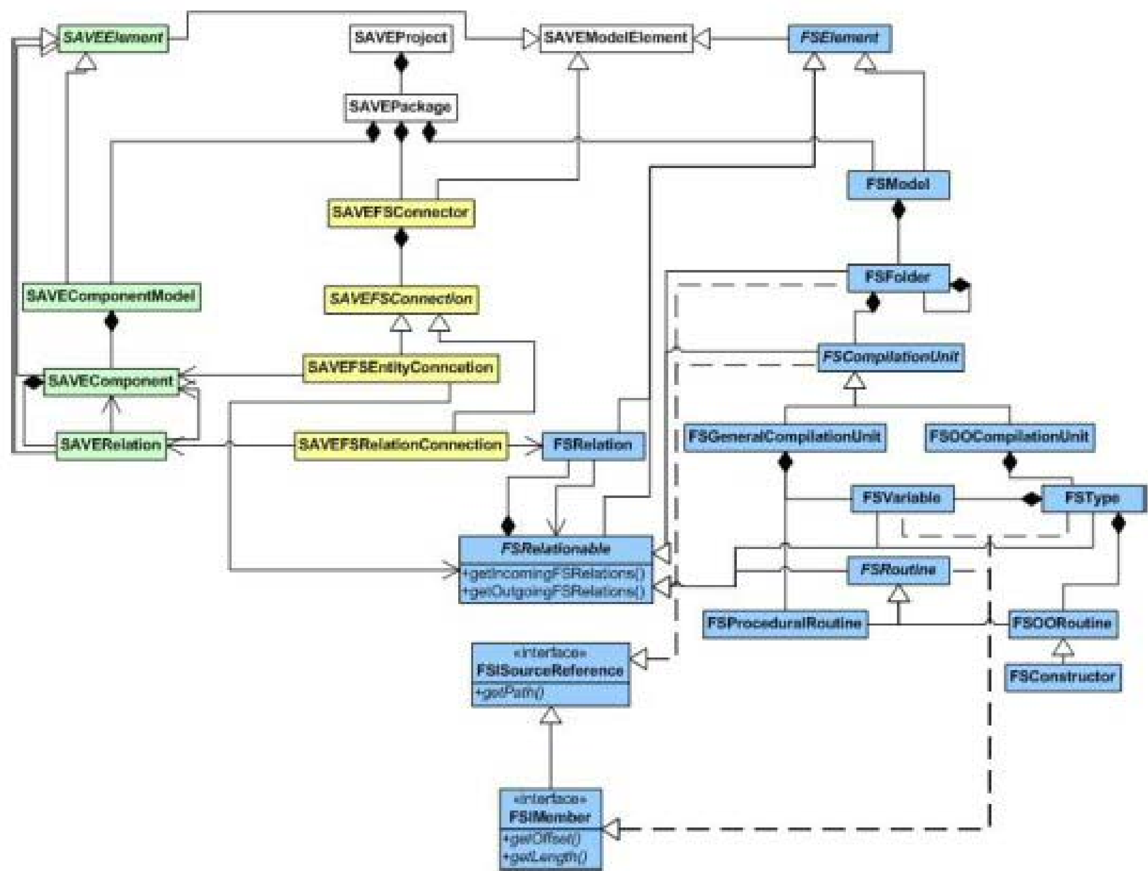

Figure 3.2: SAVE core model (from Rost 06])

The other plug-ins of SAVE hold different parts of the functionality like the view, the extraction of a model from java files, and so on. But since the core model is most important for the metrics plug-in implemented in this thesis, the other plug-ins will not be discussed here.

Copyright © Fraunhofer IESE 2006

4 Designing a customizable Metric plug-in for SAVE

For the integration of a customizable metric component into the SAVE system several aspects have to be considered. First, the term “customizable” has to be specified in more detail. Then, the steps performed for a metric’s measurement at the metrics plug-in have to be explained in detail.

To add a custom metric to the metrics plug-in and perform a measurement of that metric, a process was defined which is shown in figure 4.1. It consists of four steps. (The process does not address problems of finding and implementing the concrete algorithm of the metric since this is not covered by the metrics plug-in and thus is not in the scope of this thesis.)

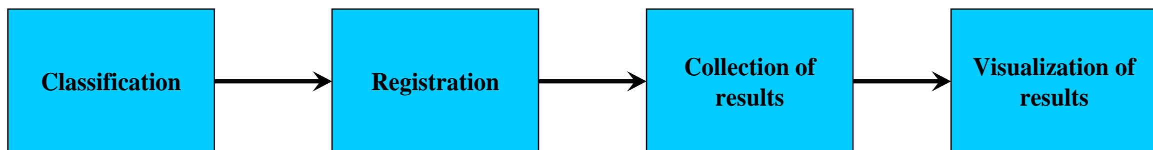

Figure 4.1: Metric design process

At first, the metric has to be classified. This is done regarding facts like the level of the metric or the kind of results and is described in chapter 4.2. After that, the metric can be registered at the metric plug-in (see chapter 4.3) according to the classification done before. When the metric is registered, a calculation of the metric results can be started (see 4.4). The visualization of the results (see 4.5) was designed in an extra step, separated from the collection step. This means that the visualization has to be started by the user within a new action different from the action starting the calculation and was done for different reasons:

- Performance: At large projects or complex metrics the calculation of the metrics’ values can consume much time. But users might want to see the results not only once. If calculation and visualization were done in a single action, they would have to wait for the calculation every time they want to see the results again.
- Customization: Since the amount of results of a single metric calculation can get quite large it should be possible to visualize the results from different viewpoints (i.e. different subsets of the results or different kinds of diagrams). So one calculation has to be visualized in different ways at different times which would not be possible if calculation and visualization were done in one step.

Copyright © Fraunhofer IESE 2006

- Persistency: If a calculation's result is made persistent and can be visualized at later times without recalculation, it is possible to calculate a certain metric, change the analyzed product, calculate the metric again and visualize both calculations. Therefore, it is possible to see the metrics' values at different stages of the product.

4.1 A "customizable" metric framework

The main goal of this bachelor thesis is the creation of a metric framework that can be easily extended by new metrics. So, in this case, "customizable" means that the set of metrics the tool provides should easily be modified: Existing metrics should be removable, new metrics should be possible to add to the tool without much effort or caring about the tool's integration. Therefore the framework should split the commonalities from metric-specific aspects and provide these commonalities. Commonalities are: selecting the parts of the analyzed software a metric takes as input, delivering them to the metric, taking the results and making them persistent, visualizing the metric results and so on. The only metric-specific aspect that has to be implemented separately for every metric is the algorithm that calculates the results for this metric. That means only one interface has to be implemented. The analyzed element is delivered to the interface and a result with the value of the metric must be returned. To add a metric only this implementation has to be provided and the metric has to be registered at the framework, after that the new metric can be calculated and their results can made persistent and visualized like all existing metrics.

Another requirement given by the term "customizable" is that as much options of the calculation and visualization of metrics as possible should be editable by the user. The user should be able to "customize" the metric plug-in to his needs and his viewpoint. That means that both the calculation and the visualization have to be configurable. This is done by presenting a wizard for each of these steps. Here, options like the set of elements the metric is calculated at or visualized, or specific options for each metric can be defined by the user. The wizards are explained in more detail at the chapters 4.4. and 4.5.

4.2 Classification of metrics

There are two general levels of metrics in the SAVE-tool:

1. *Measurements of the source code elements*. These measurements like, for instance, the Lack of Cohesion Metric or the number of LOC can only be found by analyzing the source code of components (e.g. Java class files). Other metrics, like the cyclomatic complexity, only take a single procedure as input. Higher-level elements in the architectural hierarchy like packages and subsystems (or classes for routine-level metrics like the cyclomatic complex-

Copyright © Fraunhofer IESE 2006

ity) do not have their own singular source code element belonging to them but consist of many different source code elements. Obviously, it doesn't make sense to get source code measurements directly of these higher level components. Instead, such a measurement must be taken from every source code component. Higher components can derive numbers from the values of all sub-components belonging to it. For the aggregation of higher-level results from lower-level ones, different possibilities exist that will be discussed later.

2. *Measurements of model elements*. These measurements have to be calculated separately for every architectural level. As an example the coupling between single classes can be measured by counting the relations that go from and to every class. The coupling between packages on the other hand can not be measured by just summing up the coupling of all classes in a package as it would be done with metrics of the first type. Instead, this metric has to be calculated independently by counting all relations that go from and to a package (and leaving out the relations between classes or packages within that package).

In SAVE, metrics of the first type have to be calculated via the FSModel, the second type can be calculated using the SAVEModel.

Each level can be partitioned further: Source code metrics can be metrics of a module or of a routine; architectural metrics can be measured at single components or at the whole architecture. So in the end there are 4 general types of metrics.

	Level the metric operates on	SAVEModel / FSModel	SAVE-element used as input	metric example
1.	Routine	FSModel	FSRoutine	Cyclomatic complexity
2.	Module	FSModel	FSCompilationUnit	lack of cohesion
3.	Single component	SAVEModel	SAVEComponent	fan-in, fan-out
4.	Architecture	SAVEModel	SAVEPackage	Degree of similarity between two architectures

Table 4.1: Levels of metrics

The last type is different to the others since the output of these metrics has to be handled differently: All metrics of one of the first three types can take in every execution step exactly one element as input and produce one result as output. The Fan-out metric, for instance, can be calculated step by step where in each step a single component is taken, the fan-out for this component is calculated and a result containing the calculated value is delivered. A metric on the architectural level, on the other hand, may want to deliver more than one

Copyright © Fraunhofer IESE 2006

result for a single architecture. A metric, for instance, that calculates the degree of similarity between two different architectures might not only deliver a number for each architecture but also a number for every component representing the degree of similarity between this component and another one within the other architecture. So, at these metrics, passing one element into an execution step can lead to more than one result.

Another way of classifying metrics that could be measured by the metric-tool is the kind of result they produce: Two kinds of metric results are supported by the tool:

- Simple results that contain only a single number for each component / compilation unit / routine, and
- Results that contain a number and a list of components, like for instance the other components of a reference cycle (as described above). These components can be shown when such a results is visualized.

A metric in the literal sense produces only a "metric" value, so every result consists of only a number. But at some metrics more information would be needed to use the values of its results. A metric, for example, that counts for every component in how many reference cycles it takes part would be much more useful if it also told what other components belong to these reference cycles. So, in this case a result should not only consist of the numeric value but also of these other components and these components should be presented when this metric is visualized. There exits many other examples of different kinds of complex results that should be supported and visualized each in a different way. But since the metric-tool has to be generic and new metrics should be easily addable a trade-off has to be found between the flexibility to define metrics with complex results that have an own special visualization and the possibility to easily add new metrics to the tool. Therefore only the result with a list of components was implemented

Source code level metrics can be further classified regarding the aggregation of higher-level results as described above. For this aggregation, different possibilities exist: The lines of code of a component for instance consist of the sum of the lines of code of all its subcomponents. However, for the cyclomatic complexity metric taking the sum of all subcomponents results would be inadequate: All results of this metric are contain a numeric value between 0 and 1, and the sum of these values would carry nearly no information. So, in this case, taking the average value of the subcomponents results is a more appropriate way to derive the results of higher-level components. All in all, four kinds of aggregating higher-level values were identified:

1. Addition of all lower-level results,
2. Calculating the average value of all lower-level results,

Copyright © Fraunhofer IESE 2006

3. Taking the minimum value below, and
4. Taking the maximum value below.

4.3 Registering metrics

Since the tool is implemented as an Eclipse-plug-in, an extension point can be written where new metrics announce themselves. For the calculation and the definition of specific attributes of the metrics, new metrics have to implement some given interfaces. Therefore, at the extension point the following attributes have to be specified: The level of the metric, the metric's name, a short name, the kind of result, a rule for the aggregation of higher-level results, a wizard page class, and the metric class. Figure 4.2 shows the extension point for registering a compilation unit metric. The attributes will be explained afterwards.

Extension Element Details

Set the properties of "CompilationUnitMetric"

id*:	UnitSize
name*:	Unit Size in LOC
metricClass*:	de.fhg.iese.pulse.SAVE.computations.metrics.concreteMetrics.loc.LOCMetric [Browse...]
shortName*:	LOC
aggregateResultsBy*:	ALLPOSSIBLE
wizardPageClass:	de.fhg.iese.pulse.SAVE.computations.metrics.concreteMetrics.loc.LOCWizardPage [Browse...]
resultType:	NUMERIC NUMERICANDLISTOFCOMPONENTS

Figure 4.2: Registration of a metric at compilation unit level

1. level: The level of the metric defines which elements are input for the metric (see section 4.2 for the metric levels). Depending on the metrics level, different interfaces have to be implemented by the metric class since different elements of the analyzed software become the input for an execution step (for instance a single routine or a component). The metric level is not an attribute of the Eclipse extension point. Instead, depending on the metric level, the metric has to be registered at different extension points.

2. name: The name of the metric is the name that is shown to the users when they select the metrics they want to calculate. It has to be unique among all metrics of one level so the metrics can be distinguished.

Copyright © Fraunhofer IESE 2006

3. shortName: The short name is a three-letter abbreviation of the metrics name. It is used when the metric is visualized within the SAVE-architectural view (see 4.5.1).

4. AggregateResultsBy: If the level of the metric is the source code (routine or module), it has to be specified how higher level components aggregate their values from lower ones (see 4.2). Five cases exist: The metric uses the sum, it uses the average value, the minimum, the maximum, or all ways are possible. In the last case the user has to select the aggregation rule for every metric calculation.

5. resultType: The kind of result can be a simple result with only a numeric vale, or a result with a list of components (see chapter 4.2). If the result contains a list of components the metric class can add arbitrary components to every result. These components can be presented during the visualization step (see 4.5)

6. wizardPageClass: If the metric has some metric-specific options that should be specified before each single metric calculation, the metric can contribute a wizard page. This class has to be subclass of a given wizard page and can add GUI elements to the page (labels, check boxes, text fields etc). When the user edits these elements during his selection of metrics (see 4.4) their values can be stored in attributes classes that are delivered to the metric class.

7. metricClass: The metric class does the calculation of the metrics values. It has to implement a specific interface (depending on its level) with a "calculate" method to which an element (like a routine or a component), a list of metric-specific attributes (if defined in the wizard page class) and a result are transferred. The method has to set a numeric value at the result (if the metric is on the architectural level it has to create a list of results and return it, see 4.2). This method is called for every element that is specified by the user (see 4.4)

Eclipse offers an easy way to set attributes while registering at an extension point so the registration is very comfortable and can be done quickly without much effort. For every metric level the metric tool offers a separate extension point so the metric level does not have to be an attribute of the extension point.

4.4 Collecting metrics

When some metrics are registered at the extension point, the metric tool offers the user the opportunity to calculate metrics. This can be done using a tabbed wizard. The wizard consists of four main tabs, one for every metric level. At every level, all registered metrics of this level are presented and can be selected.

Copyright © Fraunhofer IESE 2006

If a metric comes along with an implementation of a wizard page (see 4.2) the user can open this page and edit the metric-specific options. An example of a metric with such an option could be a metric that counts the lines of code. Here a wizard page could ask the user to specify if the metric should consider empty lines or not, using a checkbox.

For every metric level the user can as well specify the elements of the software project that should be analyzed. Since these elements are hierarchically contained into each other, they are presented in a checkbox tree. The depth of the tree depends on the level of the metric: At an architectural metric, the tree consists of only one level, the different SAVE-packages. At component metrics the tree contains all components of a SAVE-package that can be freely selected. At compilation unit and routine metrics, the tree contains not only components but also the compilation units or routines below the components. For these metric types, higher level elements can not be selected without selecting all elements below them, as they need the results of the lower elements to derive their own values (see 4.2). If, during registration of a metric, both average and sum where specified as possibilities to derive higher values, the user can select one way at these wizard.

The configuration made at the wizard (the set of selected metrics, the selected elements, the metric options) is made persistent and shown at SAVE's element browser within a new node, a "Metric Container". Each container represents a single metric measurement.

Copyright © Fraunhofer IESE 2006

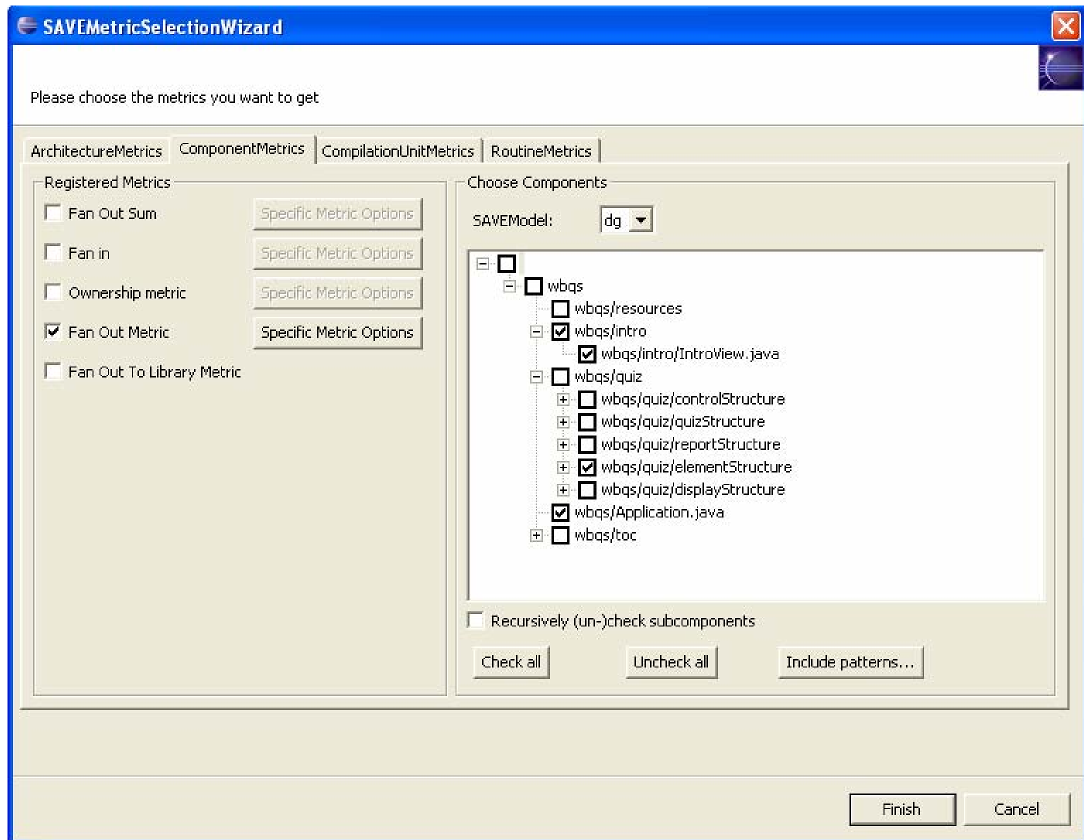

Figure 4.3: Wizard for the selection of component-level metrics

When the user finishes the wizard, the system starts calculating all metric values by passing the selected elements and the metric-specific options to the selected metrics. The results are stored in the metric container. At every metric container the wizard can be reopened to change the selections and the results of the metric calculation can be visualized (see 4.5). The user can start these actions via the SAVE browser's context menu. The created diagrams as well appear below the metric container.

4.5 Visualizing metrics

As described in section 2.3, two general approaches are possible to visualize the values of metrics: within architecture visualization or within separate diagrams. Both ways were implemented. For the presentation of metrics within an architecture visualization, the SAVE tool's architecture visualization was used.

Copyright © Fraunhofer IESE 2006

4.5.1 Visualization within SAVE

SAVE already offers a view on the analyzed architecture in an UML-like visualization. The implemented visualization of metrics within this view adds a label to every component showing the numeric value of the metric. To every metric value a traffic light was added, indicating with a green, yellow, or red circle if this value is optimal, acceptable, or not tolerable.

Chapter 2.3.1 gives a list of ideas for adding metric information to architectural views. A way to present metric values was already implemented at SAVE. But since way is not sufficient, it was not used for this metric tool. Other literature references suggest changing the color, size or form of a components representation according to the metric values. These ideas were not implemented either because they would affect many other parts of the SAVE's visualization and make it very confusing to add metric values to an existing visualization (all components would change their visual representation at once). Therefore, it was decided to add only a label with the numeric value. By this way, many metrics can be presented in a single architecture view without changing it much. Another advantage is that the user can see the exact value of a metric and not just its magnitude as it would be the case with a representation of the values by color or size. On the other hand, numeric values are harder to see and unusual results can not be identified very fast. Therefore, the traffic light was added.

The ranges of optimal and acceptable values have to be defined for every metric. This can be done by to ways:

- Every metric can override methods defining the default min and max value of optimal and acceptable values. If they don't override these methods, the default values all are zero.
- At the general preference and every views property page these four values can be changed to specify new values for each project.

When at set of metrics was calculated, the user can open an SAVE view and add the results of one or more metrics. This can be done via the context menu where all calculated metrics appear and those metrics that should be visualized can be selected. For every selected metric the short name, the components' value, and the circle mentioned above are shown at every component. If the metric was not calculated for a component they appear in grey.

Copyright © Fraunhofer IESE 2006

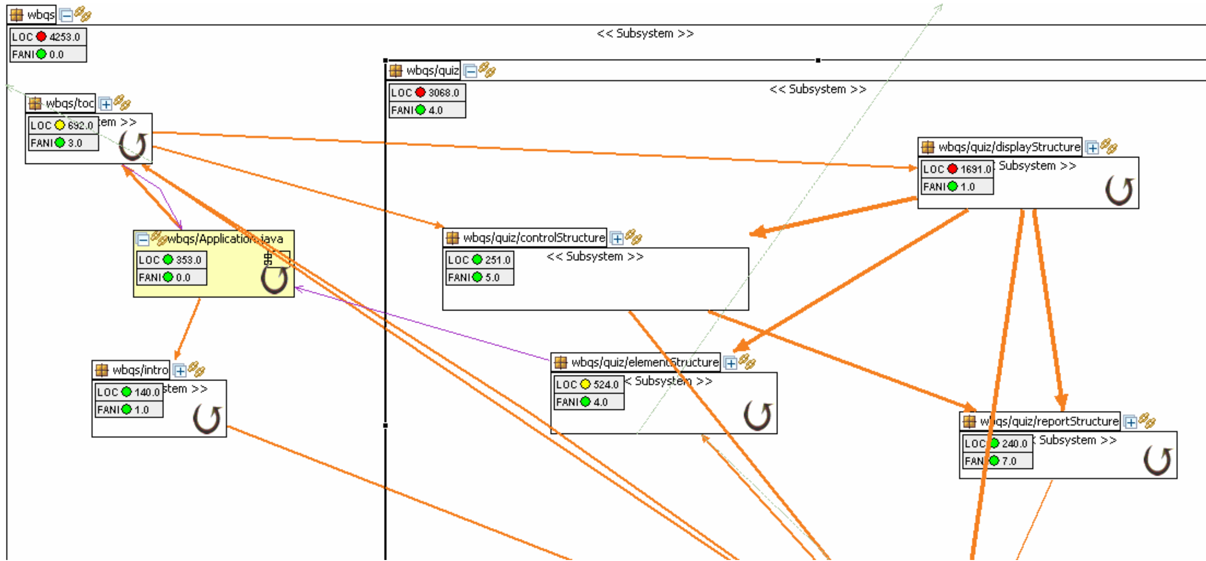

Figure 4.4: Visualisation of Fan in and LOC within a SAVE view

The SAVE-architecture view shows only components of the analyzed software. Single compilation units or routines are not visualized. Therefore, their metric values can not be shown within a SAVE view. These values can be presented using separate diagrams.

4.5.2 Visualization using diagrams

There are two general types of diagrams implemented in the metrics plug-in: diagrams of a single metric and diagrams concerning more than one metric. For both types the user can open a wizard via the context menu. For these wizards, the kind of diagram that shall be created and all elements that shall be shown in the diagram can be selected specifically. Since only elements whose metric values where calculated before can be visualized, only these elements can be selected. Therefore, the hierarchical tree of elements can be modified showing only these elements. After finishing one of these wizards, the created diagrams are shown in the SAVE model browser and can be reopened with a simple double click. All diagrams are shown in a new view of Eclipse and can be saved as bmp-files.

4.5.3 Single metric diagrams

Three diagrams can be created to visualize a single metric:

Copyright © Fraunhofer IESE 2006

The first diagram type simply presents the values of the selected components (or routines or compilation units) with a bar chart. Every result is shown as a bar, its height representing the respective value

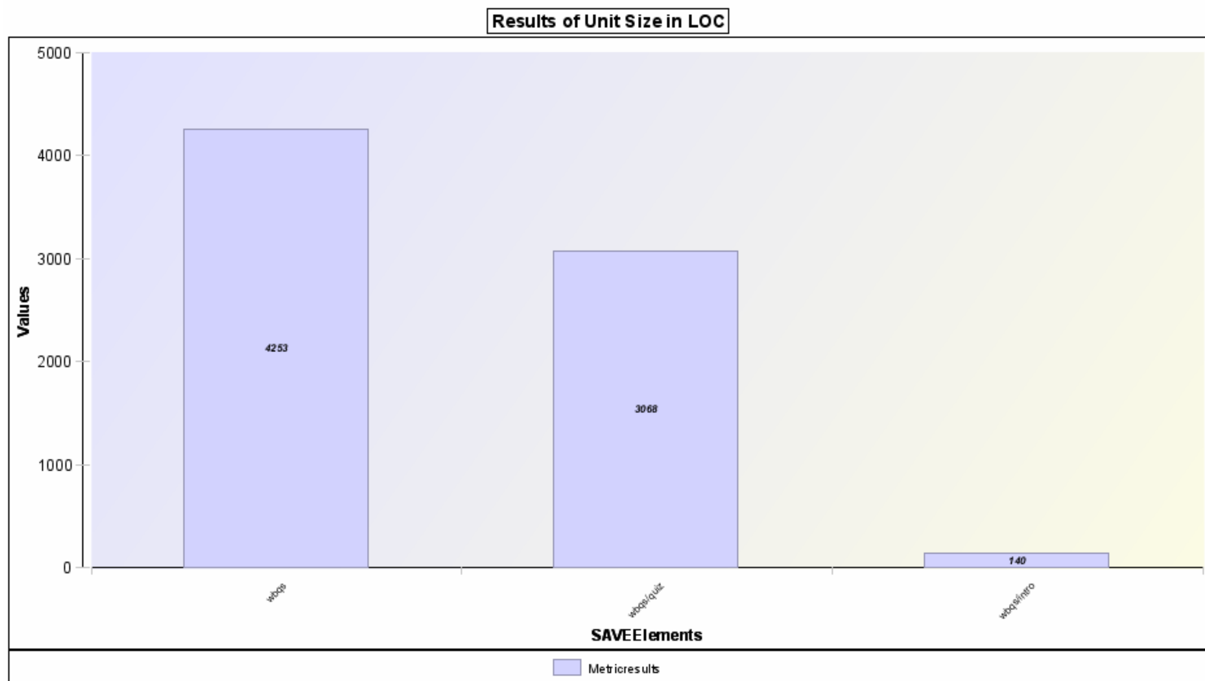

Figure 4.5: A single metric shown in a bar chart

Another diagram can show the top and the bottom values out of the set of selected components. The user has to specify how many top and how many bottom values are presented. The results are visualized in a bar chart similar to the first diagram.

Copyright © Fraunhofer IESE 2006

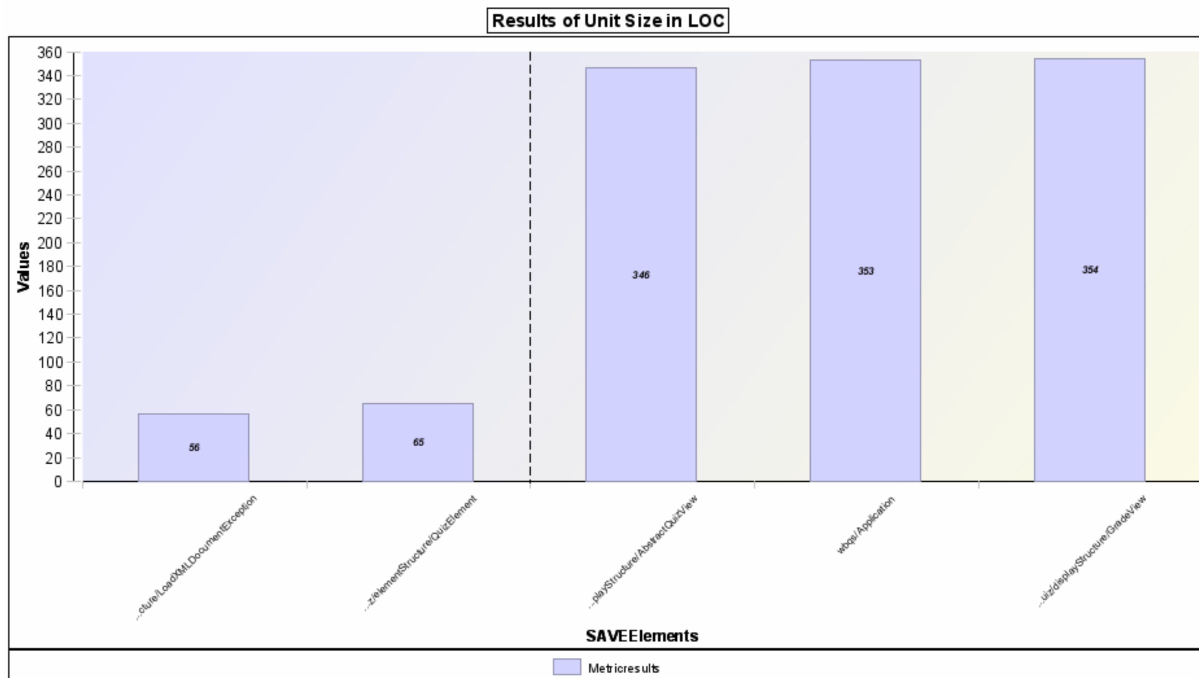

Figure 4.6: Top 3 and bottom 2 of a single metric

The third diagram type shows statistical information to the selected elements: For every element it presents the highest, the lowest and the average value of all subcomponents below it. At the Lines of Code metric for instance this diagram can present the length of the biggest unit, the smallest unit and the average length of all compilation units. Here the user can select if all elements below the selected elements should be considered or only the elements at the lowest level of the hierarchy tree: At the Fan-in metric, for instance, all lower elements should be considered when presenting the min, max, and average value – at the LOC metric, on the other hand, higher elements get their values from the sum of all lower values, therefore the maximum value would always be the value of the highest element in the hierarchy and the average value would be much higher then the average of all compilation units if not only leaf elements of the hierarchy were taken into consideration.

Copyright © Fraunhofer IESE 2006

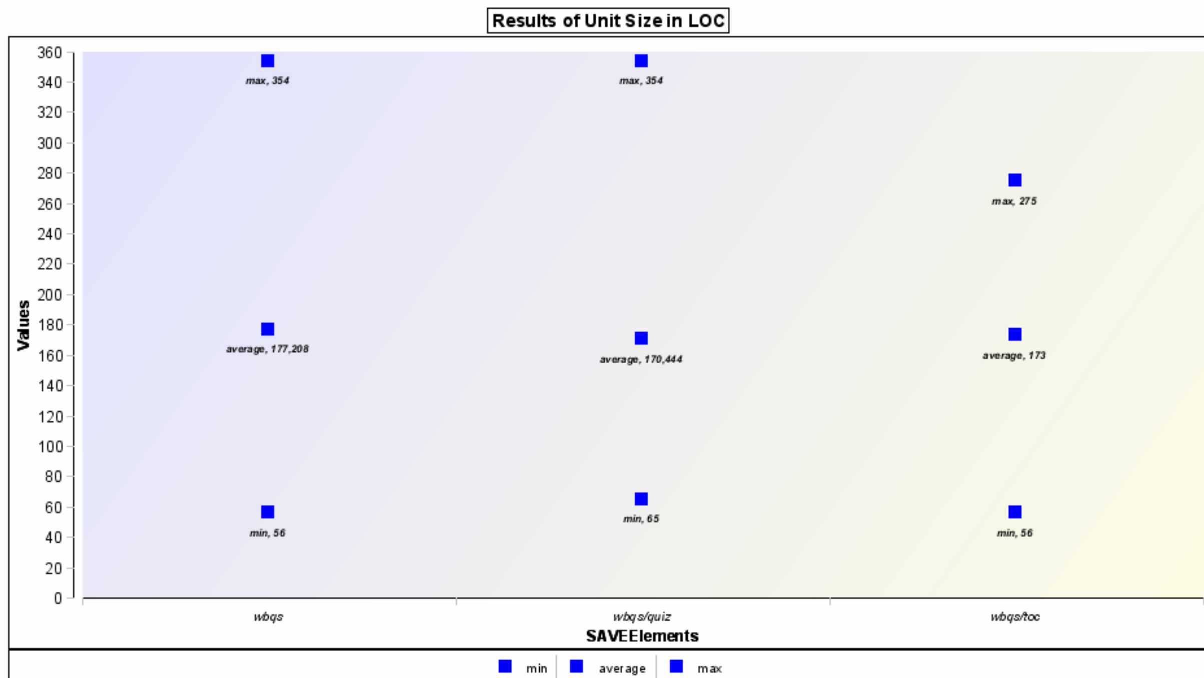

Figure 4.7: Highest, lowest and average value below 3 selected components

If the result of the metric has not only a numeric value bit also a list of components another way of visualizing the metric exists: A single component can be selected and all other components that are listed in the components result are shown in an new diagram (this diagram is created using the SAVE-architectural view as in 4.5.1)

4.5.4 Diagrams of a multiple metrics

For the visualization of multiple metrics there exist two diagrams:

A *scatter plot* shows the values of two metrics against each other. Two metrics can be selected that are represented by the x- and y-axis of the chart. All selected elements are shown as crosses in this chart.

Copyright © Fraunhofer IESE 2006

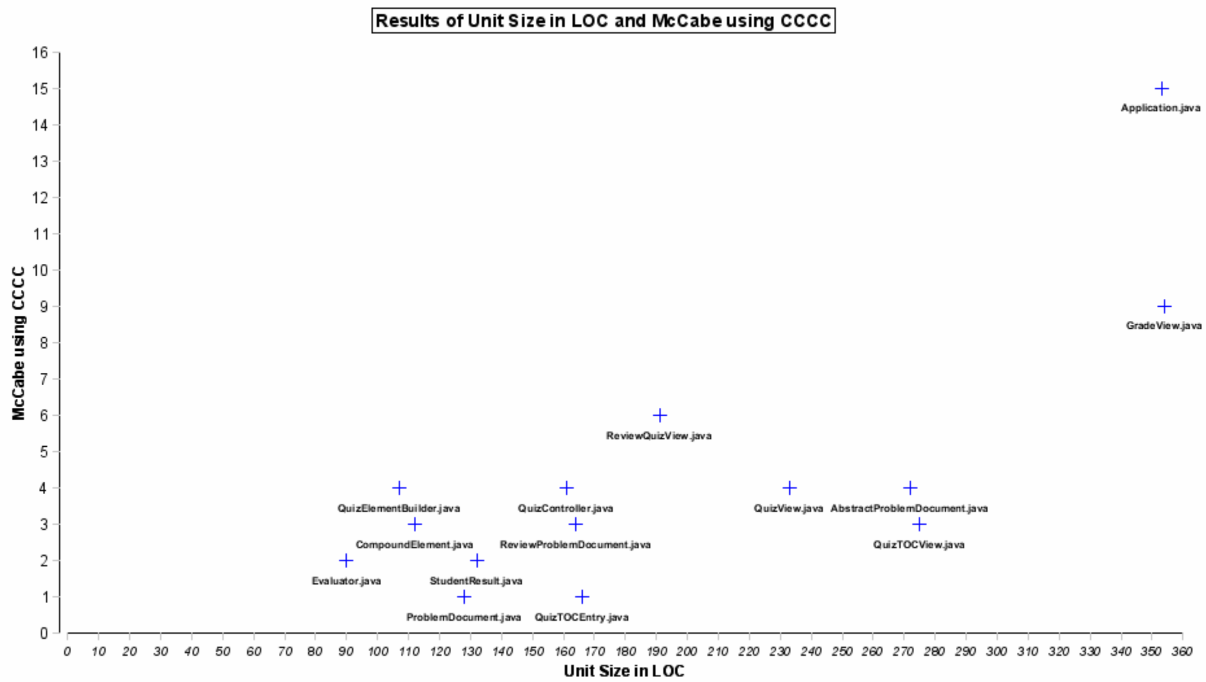

Figure 4.8: A scatter plot showing two metrics against each other

A *kiviat diagram* (see 2.3.2) shows all calculated metrics of a selected element in one diagram. For every metric the diagram visualizes not only the value of the element but also the minimum and maximum values of all elements. The length of each branch displays the relation of the elements' value at the corresponding metric to the other element's values: The nearer its value is to the maximum value the longer is the branch.

Copyright © Fraunhofer IESE 2006

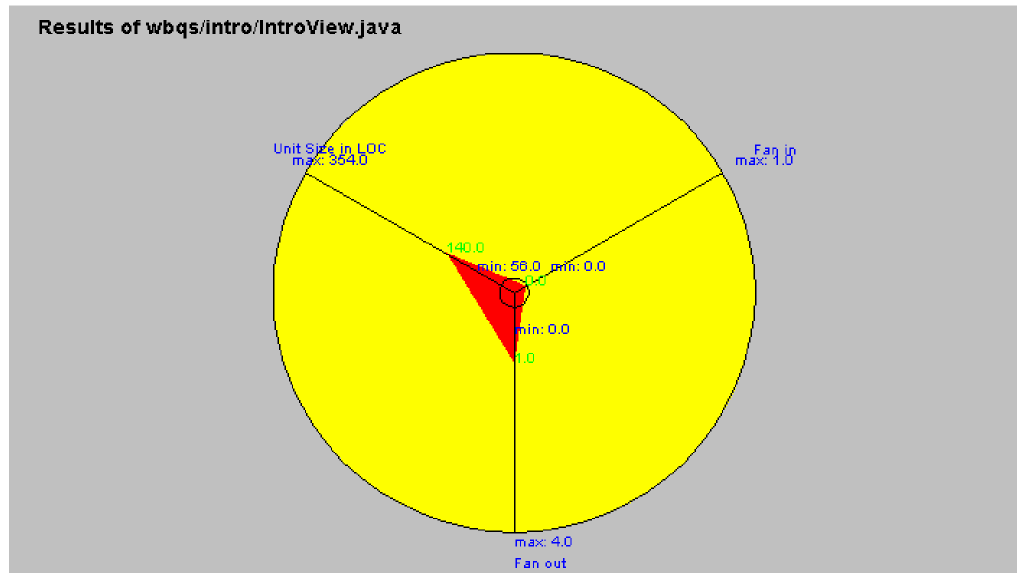

Figure 4.9: A kiviat diagram showing three metrics at at a compilation unit

Copyright © Fraunhofer IESE 2006

5 Architecture and Implementation of the Metric Plug-in

The metric plug-in developed in this Bachelor thesis is decomposed into four Eclipse plug-ins that depend on each other and use the SAVE-tool plug-ins for visualization and access to the SAVE data model. Chapter 5.1 describes the architecture of the tool by showing its decomposition into different plug-ins and their main components in static and dynamic aspects. After that the important aspects of the tools' implementation are illustrated.

5.1 Architectural views

Following the Fraunhofer IESE's architecture view approach (see [Bayer et al. 04]) the architecture can be described in three ways: The *conceptual view* gives an overview of the whole metric tool. The *structural view* describes the main components of the metrics plug-in and their subcomponents below. Finally the *behavioral view* explains how these components interact during operation.

5.1.1 Conceptual view

The conceptual view of the metrics plug-in can be seen in figure 5.1.

Copyright © Fraunhofer IESE 2006

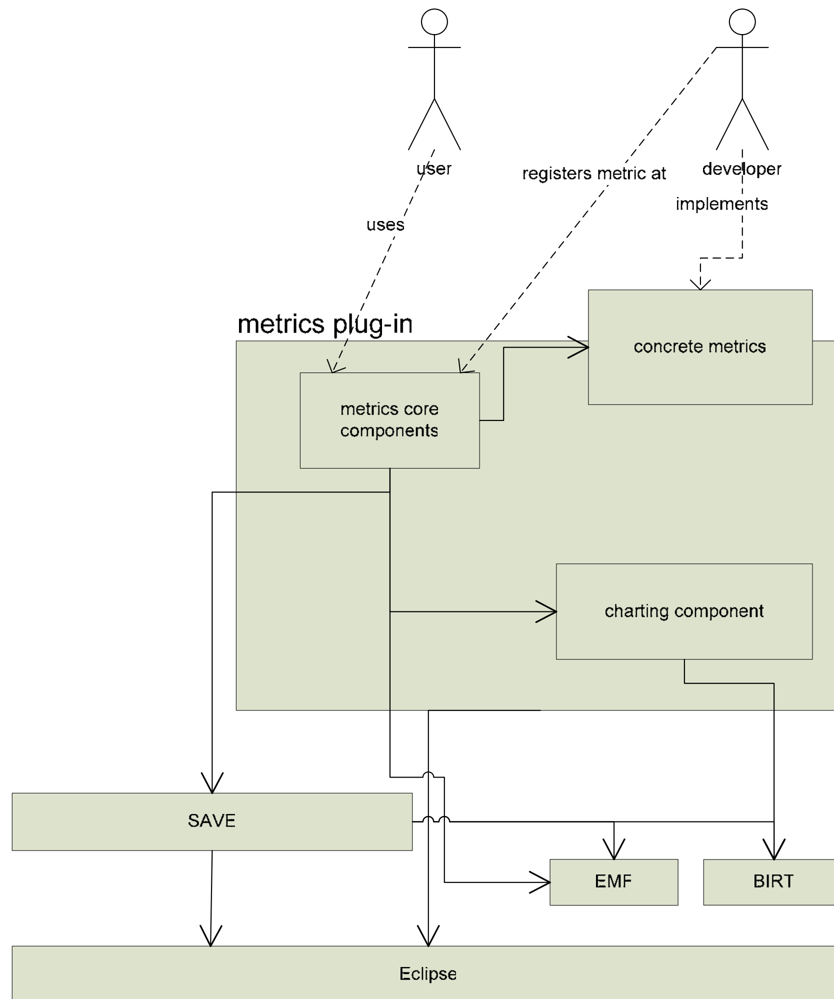

Figure 5.1: Conceptual view

The metric plug-in consist of two core components, the charting component for the visualization of chart presenting the metric results, and a set of implemented metrics. The developer of metrics can implement own concrete metrics which can but don't have to be part of the metrics plug-in. At the core components the developer can register the implemented metrics and the user can start the metric calculation. The core components use the charting component for visualization of the results and the EMF framework for storing the results, the metric options, and the created diagrams. SAVE is used by the metric core components to get access at the SAVE model and to visualize the metric results. The charting component uses the BIRT framework to create diagrams. The whole metrics plug-in is registered at different Eclipse extension points.

Copyright © Fraunhofer IESE 2006

5.1.2 Structural view

The structural view describes the main components, their decomposition, and the relations between them. First, all packages of the metrics plug-in are presented; then a more detailed look into the relevant parts of the architecture is given.

5.1.2.1 Overview

The structure of the tools' top level packages is shown in figure 5.2

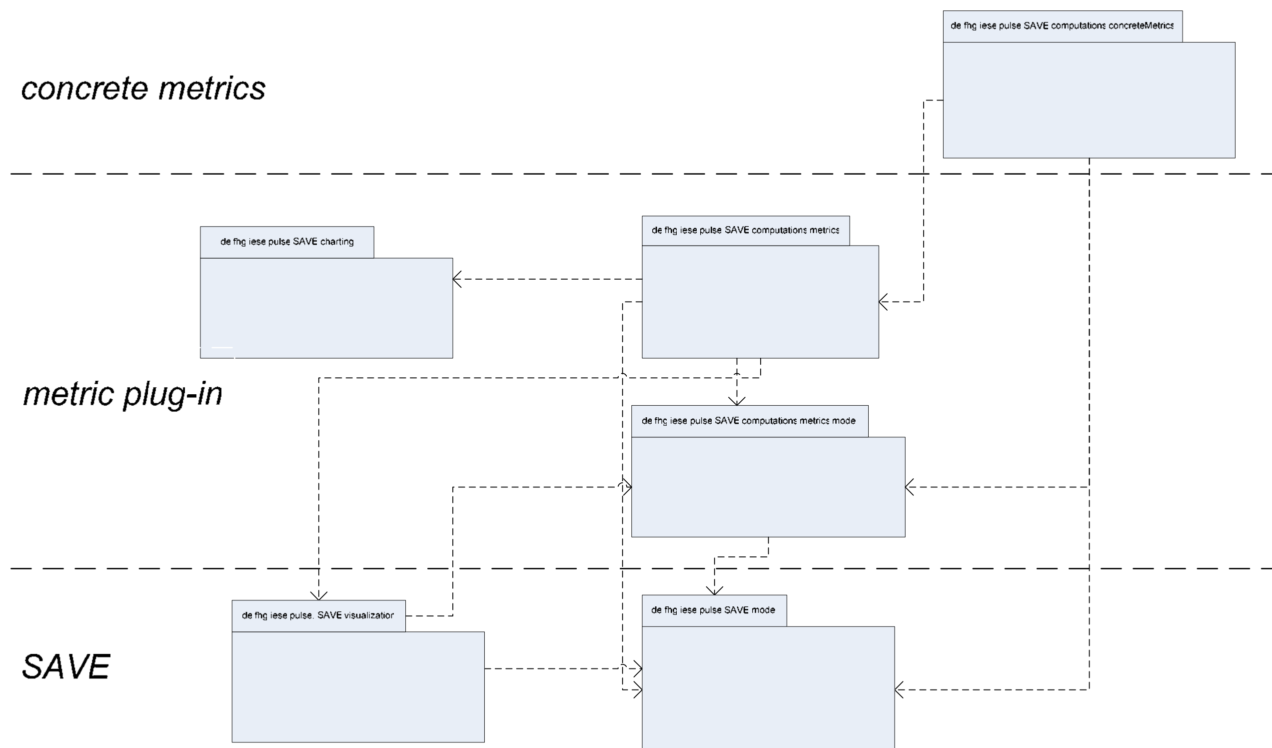

Figure 5.2: Main components

The architecture can be divided into three layers: the highest layer contains the concrete metrics. They are based on the components of the metrics plug-in which itself is based on the SAVE tool. Only one dependency violates the standard layers architecture: The SAVE visualization component needs the metrics model to visualize the metric results.

Copyright © Fraunhofer IESE 2006

Two core components realize most of the functionality of the metrics tool: The `computations.metrics.model` package hosts the data model for the metrics (see chapter 5.1.2.3). It uses the `SAVE.model` plug-in to get access to the SAVE data model. The `computations.metrics` component (see chapter 5.1.2.2) is responsible for the application logic like delivering the selected elements to the implemented metric classes, storing the metric results into the SAVE data model, or passing the results to selected diagrams. The wizards for calculating and visualizing metrics are as well implemented in the `computations.metrics` package.

For the visualization of metric results via charts the `SAVE.charting` package was build. It is independent from the other packages of the metrics tool to make it usable for other parts of SAVE. The `computations.metrics` package takes care of extracting the needed data from the metrics model and delivering it to the charting component.

The visualization of results within a SAVE view was implemented by adding functionality to the `SAVE.visualization` component. It uses the `metrics.model` package to get the results for visualization. The metrics component uses the `SAVE.visualization` component to set the options for the visualization and to create new views for visualizing results with lists of components.

Implemented metrics are within the ConcreteMetrics package. They depend on the `computations.metrics` component because they have to register at its extension point. The metrics.model package is needed to access the model.

5.1.2.2 The metrics component

To get a more detailed look at the architecture, the internal structure of the main components is explained in the following. Figure 5.3 shows the structure of the `computations.metrics` package.

Copyright © Fraunhofer IESE 2006

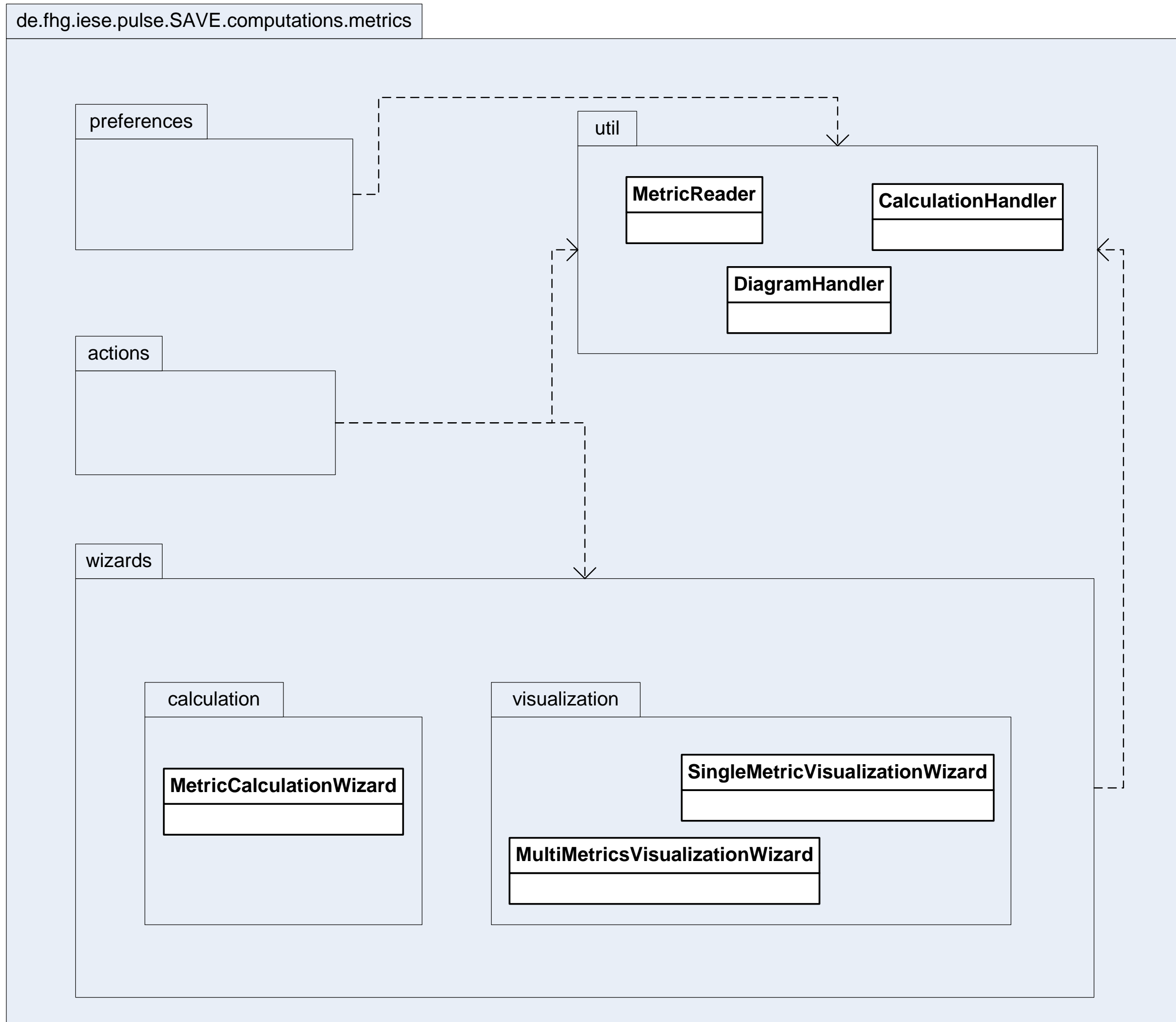

Figure 5.3: The metrics package

The action package hosts the classes that implement the user's actions. There are five actions implemented in this package: Starting a metric calculation (`CreateNewMetricCalculationAction`), changing the configuration of an existing calculation (`ReCreateMetricCalculationAction`), starting a visualization (`VisualizeMetricsAction`), opening a diagram (`Double-ClickAction`), or deleting a metric element like diagram or a calculation from the element tree (`DeleteElementAction`).

All wizards are realized in the `wizards` package. The sub-package `wizards.calculation` holds the wizard for calculating metrics or changing the calculation options and its helper classes. The wizard for visualizing the metric

Copyright © Fraunhofer IESE 2006

results is positioned in the `wizards.visualization` package (For an explanation of the implemented wizards see chapter 4).

The `preferences` package contains the implementation of the preference page for visualizing metrics within a SAVE view.

In the library package, the functionality of calculation and visualization is stored. The `CalculationHandler` delivers the selected elements to the selected metrics and creates the Result classes. The `DiagramHandler` class is responsible for the creating and opening diagrams. Creation of metric classes for registered metrics is done by the `MetricReader`

5.1.2.3 The metrics.model component

The main structure of the metrics data model is shown in figure 5.4.

Copyright © Fraunhofer IESE 2006

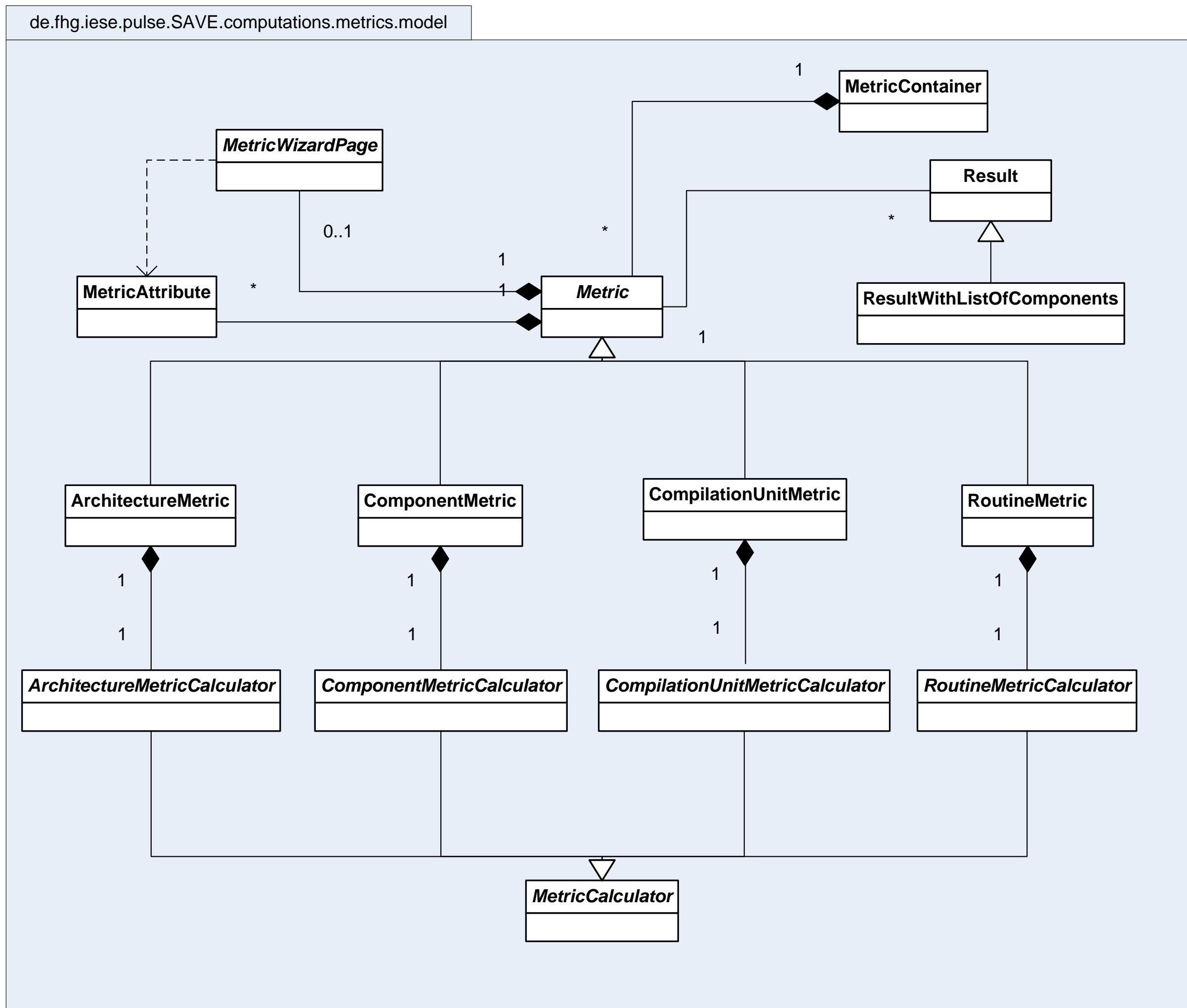

Figure 5.4: The metrics data model

The abstract class `Metric` is specialized into four classes, one for every type of metric. It is contained in a `MetricContainer` where all metrics of a single metric calculation are stored. For every metric type, there exists an abstract `MetricCalculator` class which calculates the metric value for a given element and has to be specialized for every registered. If a wizard page was registered for the metric, it has to be a subclass of the abstract class `MetricWizardPage` and is as well contained in the Metric class. All choices made on the wizard page are stored into `MetricAttribute` classes and delivered to the calculator class by the Metric class. The results are saved into Result classes (or `ResultWithListOfComponents` if necessary), one result for each element and each metric.

Copyright © Fraunhofer IESE 2006

Diagrams shall as well be made persistent, that is why their data is also stored into classes of the model. Figure 5.5 shows the structure of the diagram classes.

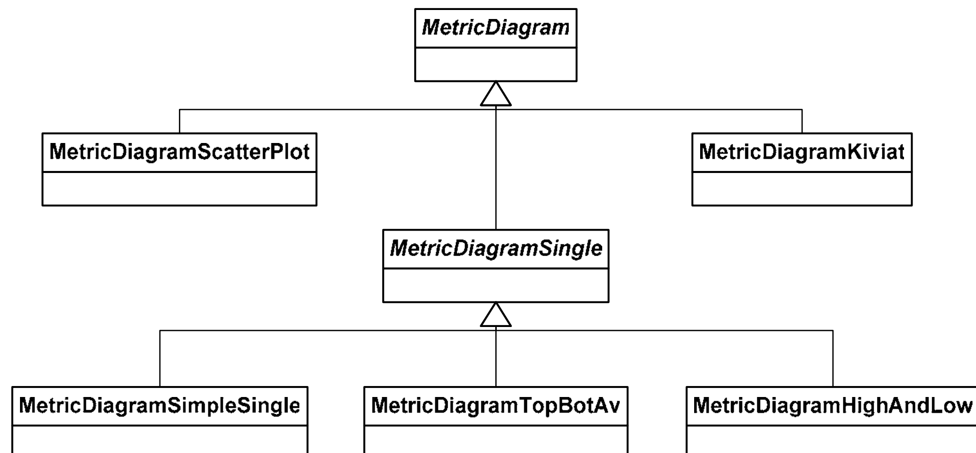

Figure 5.5: Diagram class hierarchy

For every kind of diagram there exists a class that is subclass of the abstract class `MetricDiagram`. Classes of diagrams representing only a single metric are subclasses of the `MetricDiagramSingle` class.

5.1.2.4 The charting component

To display a metric diagram, its values have to be passed to the charting package which then creates charts and displays them in Eclipse views. Its structure and the main classes were extracted from [Knieling 06] However, its modules were refactored to make them independent from the other parts developed in that diploma thesis, and its functionality was extended by the newly implemented kiviat and scatter chart. Figure 5.6 shows its structure.

Copyright © Fraunhofer IESE 2006

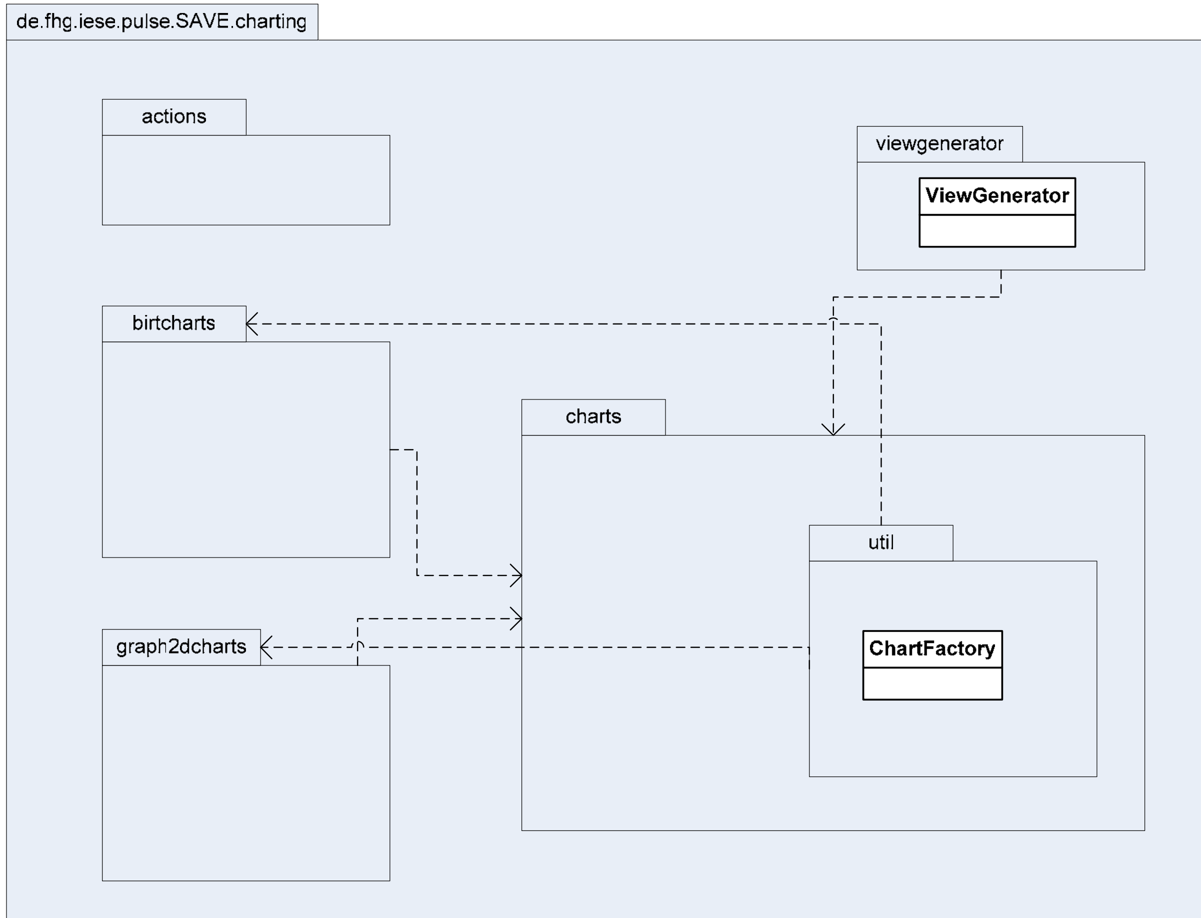

Figure 5.6: The charting package

The `charts` package contains interfaces for all supported kinds of charts. These are implemented by the packages `birtcharts` and `graph2dcharts`. The `ChartFactory` class is responsible for creating charts and adding them to the `ChartRegistry`. If a chart was created the `ViewGenerator` can open a new Eclipse view and display the chart. The `actions` package contains an action for saving the charts as picture files.

5.1.3 Behavioral view

The behavioral view on the architecture explains how the classes work together during active use of the metrics tool. Since not all aspects of the dynamic behavior can be presented, this section focuses only on the important parts of the component's interaction.

Copyright © Fraunhofer IESE 2006

Figure 5.7 shows the interaction while calculating a metric. The interaction is simplified to make it understandable: Only the calculation of a single component metric on a single component is regarded. In reality, the calculation handler calculates all selected metrics at all selected components.

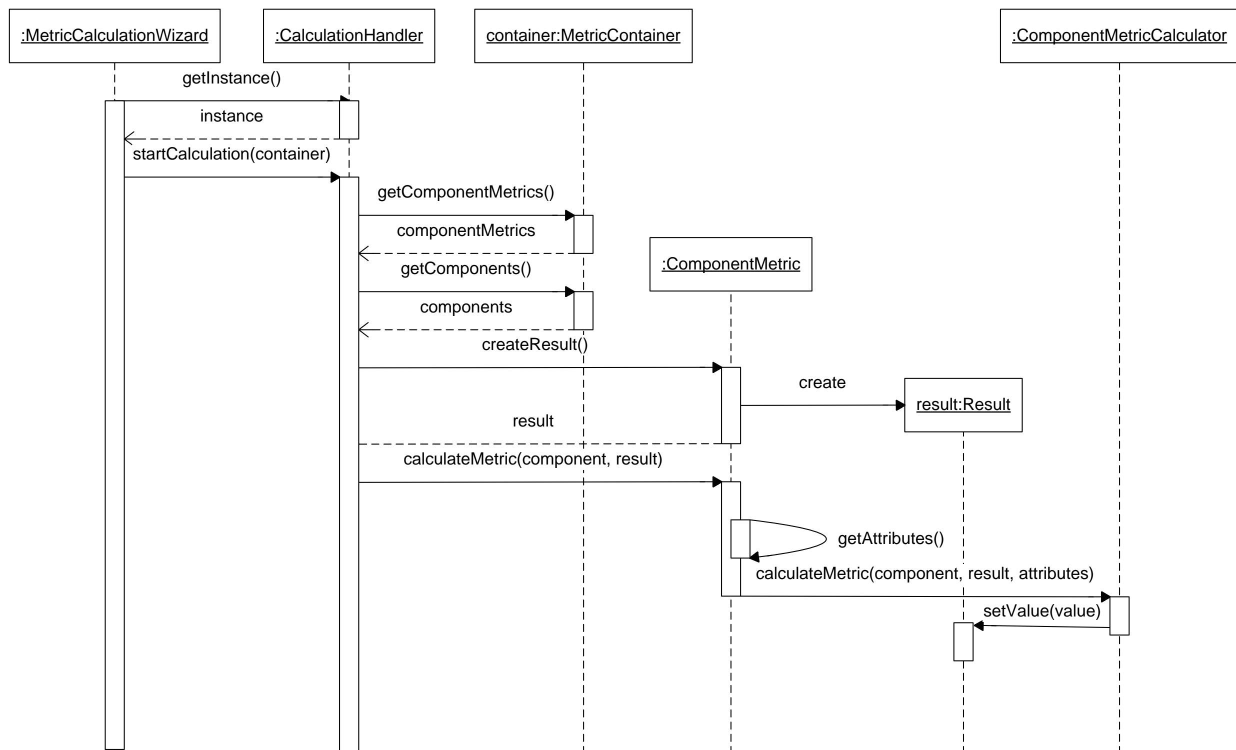

Figure 5.7: Scenario: calculation of a metric

After the wizard added all selected components and metrics to the metric container, it gets an instance of the `CalculationHandler` and calls the start of the calculation. The calculation handler catches all components and component metrics from the container. For every metric and every component it gets a result from the metric class. Then the calculate method is called on the metric calls. The metric gets the metric attributes, if the wizard page set some, and delegates the calculation to the calculator class that sets the value at the result.

The creation and display of a single metric diagram is shown in the following figure. This interaction is again simplified since all aspects regarding the interaction with BIRT and Eclipse to create an Image from the diagram and deliver it to the Eclipse view are not shown here. This was left out in order to focus only on the interaction between the metric plug-ins' modules and not on the interaction between implemented modules and external components.

Copyright © Fraunhofer IESE 2006

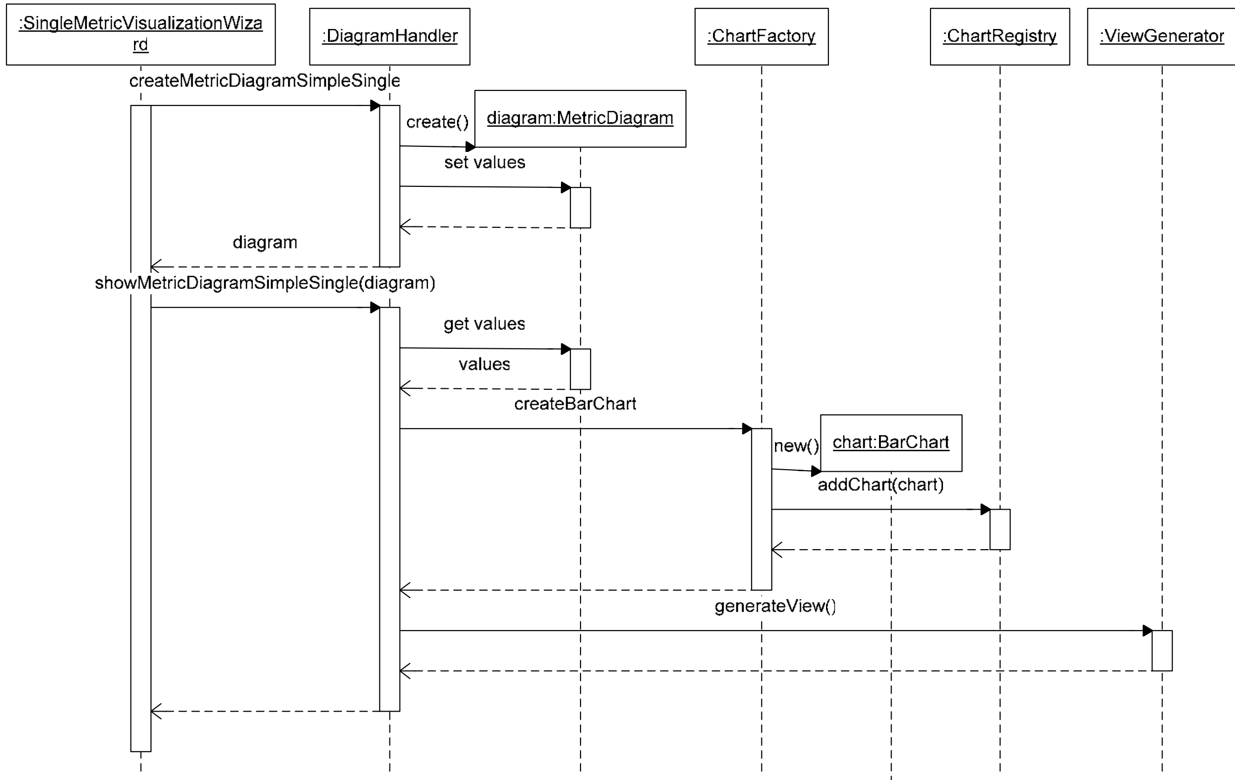

Figure 5.8: creation of a diagram

After the user finished the wizard, the wizard calls the `DiagramHandler` and passes the selected results to it. The diagram handler extracts the values needed for the selected diagram from the results and creates an instance of the diagram to which it passes the values. After that, he returns the diagram to the wizard (In order to make it not too complex, it is not shown in the figure that the diagram handler stores the diagram at the selected metric to make it persistent). Then the wizard calls the diagram handler again to display the created diagram. (The creation and display are separated into two methods to make it possible to show a diagram several times without having to recreate it).

The diagram handler receives the values from the diagram and by calling the creation of a bar chart it passes them to the `ChartFactory`. Form the chart factory the values are passed to the newly created `BarChart` that builds an image of the chart. The `BarChart` class is added to the `ChartRegistry` which is later used by the then called `ViewGenerator` to get the chart and display it within a new Eclipse view.

Copyright © Fraunhofer IESE 2006

5.2 Implementation

Obviously not all aspects of the plug-in's implementation can be discussed here – all details of the implementation can be found in the implemented classes and their respective code comments. Therefore, this chapter discusses only the most relevant parts of the implementation. An overview of the size of the implemented plug-ins can be seen at the following table.

Plug-in	Lines of code	Nr of compilation units	Nr of classes	Nr of routines	Annotation
metrics	5644	51	61	178	
metrics.model	10627	53	54	392	Most of the code generated by EMF.
charting	3329	31	33	133	Parts of the code reused from [Knieling 06]
concreteMetrics	1369	21	27	42	
total	20969	156	175	745	

Table 5.1: Size metrics at the implemented plug-ins

The metrics presented in this table were calculated using the metric plug-in. The plug-in's visualization of the lines of code can be seen in the following figure.

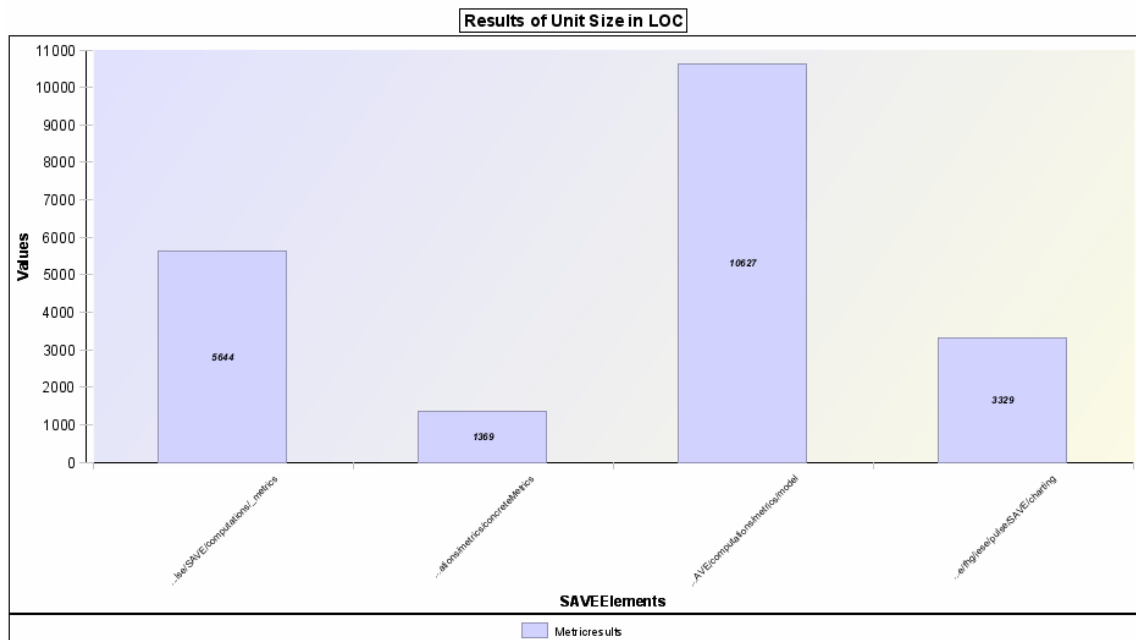

Figure 5.9: Lines of code of the plug-in's components

Copyright © Fraunhofer IESE 2006

5.2.1 Metrics model

The architecture of the metric model can be seen in chapter 5.1.2.3. For the implementation of the model EMF (see chapter 3.1.2) was used. EMF offers a very easy way to specify a model using annotated java interfaces, an approach which was used here: All model classes were specified as interfaces with some annotations as information for EMF. Then the EMF generator created an EMF model and java classes implementing these interfaces. Additional methods that should not be considered by the model were as well specified in the interfaces and implemented manually. (The respective methods can be found at the javadoc comments.)

EMF creates a hierarchical model from the specified elements: If one model element references another element (or a list of other elements) it can be specified by an annotation flag on the Java interface that the referenced element is contained in the referencing element. When the element is saved using EMF's persistence mechanism, all its contained elements are automatically saved, too.

The core model of SAVE (see 3.2.2) is also implemented via EMF. So the advantage of using EMF for the metrics model was that it could very easily be integrated into the existing SAVE model: only by specifying the metrics model elements as being contained in the SAVE model elements, they belong to the same model. Therefore these elements are as well presented in the SAVE model explorer and can made persistent together with the SAVE model by only saving the highest element in the EMF hierarchy, the SAVE project.

The containment hierarchy of the metric model elements corresponds to the element composition presented in figure 5.4: A metric is contained in a metrics container and contains its attributes and diagrams. However, the results of a metric are not contained in the metric: Since a single result belongs to both a metric and a SAVE component (or compilation unit or routine), it was decided, to let the result be contained in the corresponding SAVE model element. The SAVE model already offers the possibility to add attributes to each model element, so the results could be added as attributes to the model elements without additional effort. The metrics class only contains a list of references to all its results.

5.2.2 Wizards for calculation and visualization

All wizards of the metric plug-in are subclasses of the Eclipse JFace (see 3.1.1) class Wizard. This class offers much functionality for wizards like adding pages to the wizard, opening or finishing the wizard.

One aspect nearly all wizards have in common is an element tree: For the calculation wizard, this tree shows all elements of the analyzed software (compo-

Copyright © Fraunhofer IESE 2006

nents for component metrics, components and compilation units for compilation unit metrics and components, compilation units, and routines for Soutine metrics). The user can select for which elements the selected metrics are to be calculated. For the visualization wizards, the trees show only those elements the selected metric was calculated at. Here, the user can select the elements to be visualized. These trees are also supported by Eclipse: To create a tree only a content provider and a label provider have to be implemented that specify the elements shown in the tree and the label of the shown elements. Figure 5.9 shows such an element tree.

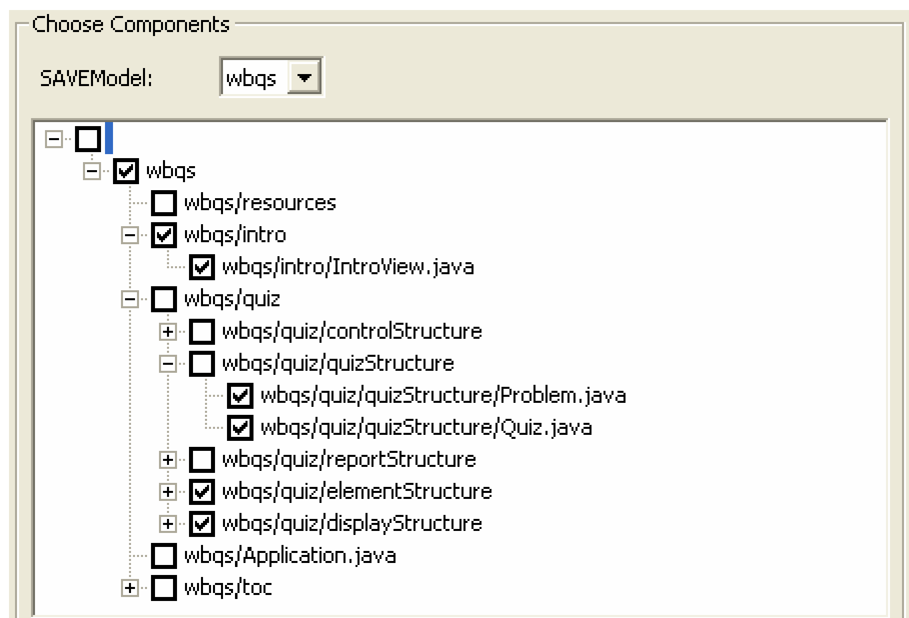

Figure 5.10: An element tree in a calculation wizard

As explained in chapter 3.2.2, every analyzed piece of software is represented by two different hierarchies within the SAVE model: One component hierarchy and one hierarchy for the file system. Therefore, at the calculation wizard the tree of components was build using the component hierarchy, the trees for compilation unit and routine metrics on the other hand were build using the file system hierarchy.

For the trees at the visualization wizards, two problems arose:

1. For component metrics it is possible to calculate metrics only at some components without calculating the metric for their parent components. How can a component tree for the visualization be build when only components the metric was calculated at shall be shown in the tree?

Copyright © Fraunhofer IESE 2006

2. When the wizard page for the visualization of two different metrics in a scatter plot is shown, the tree should only show components (or compilation units or routines) both metrics were calculated at. But if one metric was at component level and the other at compilation unit level – which one of both hierarchies shall be shown in the tree?

To solve the first problem, it was decided to modify the component hierarchy if necessary: if a component has no result for the selected metric but its children have, these children are presented in the tree at the position of their parent. So the algorithm that gets all children `children` of a component c in the visualization page tree is (given a map m that contains all components with results):

```
List getChildrenForTree(Component c):
FOR component child OF c.childrenComponents:
    IF m.contains child:
          children.add(child)
    ELSE
          children.addAll(getChildrenForTree(child))
    END IF
ENF FOR
RETURN children
```

Figure 5.10 shows the outcome of this algorithm: The original component hierarchy as shown in Figure 5.9 was mutated to present only those components the result was calculated for.

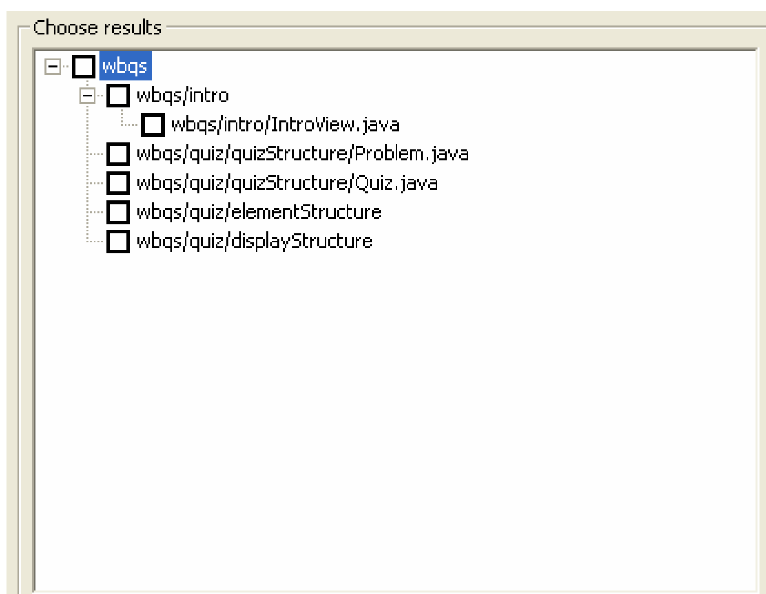

Figure 5.11: An element tree in a visualization wizard showing only components selected in fig 5.10

Copyright © Fraunhofer IESE 2006

The second problem was solved using the `SAVEFSConnector` of the SAVE core model that gets the element of the file system hierarchy for every given component. Since the element tree should only show elements where results exist for both selected metrics, and component level metrics have no result on single compilation units or routines, it was chosen to use the component hierarchy as input to the element tree. Therefore, nearly the same algorithm as in the first problem was used. The only difference is that in this case the algorithm does not only check if for a component a result exists at the selected component metric. It also checks if the second metric contains a result for the file system element the connector delivers for the given component. Only if for both metrics there exists a result the component is added to the tree.

5.2.3 Metric diagrams

Most of the metric diagrams were implemented using BIRT (see 3.1.3). BIRT can be used to implement five kinds of charts: bar charts, line charts, pie charts, scatter charts and stock charts.

Both the visualization of the selected results of a single metric and the diagram showing the highest and lowest values were implemented using BIRT bar charts for a list of diagram types see chapter 4.5.2). The diagram showing the top, bottom and average value below selected components and the scatter chart, showing two metrics against each other use the BIRT scatter chart in the implementation.

A kiviat diagram is not supported by BIRT, so the diagram showing one component in all metrics could not use BIRT for implementation. So, this chart was implemented "manually" using the elements of Java's Draw2D, such as lines, circles and text labels.

5.2.4 Visualization within SAVE view

The implementation of a metric's visualization within a SAVE view was easy to do since SAVE was build very generic: SAVE already offers a mechanism to add overlays to the component's visualization. This mechanism called "FigureDecorator" could be used here. So everything that needed to be done was implementing the context menu where the user can select the metrics he wants to be displayed and implementing the decorator that draws the visualization of the selected metrics (again using Draw2D).

When a metric is selected via the context menu to be displayed, its results are transferred into new classes, similar to the `Result` classes, called `ResultForVisualization`. To every component presented in the view a `ResultForVisualization` is added, containing either the value of the selected

Copyright © Fraunhofer IESE 2006

metric or a flag indicating that no value exists for this metric at this component. The figure decorator creates a row in the overlay image for every `ResultForVisualization` found at a given component.

At first view it seems to be a inefficient overhead to copy every value from every `Result` class into a new `ResultForVisualization` class and not just to add a flag to every result indicating if it shall be displayed or not. But for two reasons the new `ResultForVisualization` class was created:

1. Only metrics at component and architecture level store their results already at components. Compilation unit and routine metrics store them at the file system elements. So, these values have to be transferred to the components anyway since only components are shown within a SAVE view.

2. If a metric was not calculated at a specific component, there exists no `Result` class for this metric at this component. But since in the SAVE view it shall be explicitly displayed that this metric has no value at this component (by showing a grayed line for the metric at this component), there has to be a `ResultForVisualization` class with a "no value"-flag, indicating that this metric shall be displayed and that this metric has no value.

So for metrics at file system level and for components without a metric result there had to be a new `ResultForVisualization` class anyway, therefore this class was created for all components.

When the user deselects the metric in the context menu, all `ResultForVisualization` classes are removed from their components, thus the decorator does not display the values at the SAVE view any more.

5.3 Extensibility

As explained in various aspects at this bachelor thesis, the main goal of designing the metric plug-in was to make it extensible by new metrics. This goal was realized as explained above, so new metrics can be added without even knowing anything of the tools' architecture or implementation. But new metrics are not the only aspect of extending the metric plug-in. Other aspects such as new diagrams or new kinds of result could as well extend the plug-in during its lifecycle. Since these extensions affect the plug-in's architecture and implementation it is not possible to use the simple extension point mechanism here. However, the architecture of the plug-in should make it extendable. The extensibility of the plug-in is demonstrated in the following in three different scenarios of adding functionality to the tool.

Copyright © Fraunhofer IESE 2006

5.3.1 Adding a new diagram

A new diagram type has to be implemented as a chart in the charting component. It has to implement the `IChart` interface in the `charting.charts` package. To create a diagram, two factories have to be extended by new methods: the `ChartFactory` at the `charting.charts.util` package which creates charts for a SAVE view and the `DiagramHandler` at the `metrics.util` package which creates diagram classes and delegates the creation of a chart to the `ChartFactory`. The reason that there exist both charts and diagrams is that different diagrams can be presented at the same chart. The diagrams showing simple single results and showing the highest and lowest values (see chapter 4.5.2.1) for instance both use a bar chart.

For every diagram, a diagram class specializing the class `MetricDiagram`, has to be created at the model using EMF to make the diagram persistent. Then, the visualization wizard has to be extended to offer the user the new diagram type. Finally, the delegation of the diagram creation and display to the `DiagramHandler` has to be implemented.

5.3.2 Adding a new aggregation rule

First, a new aggregation rule must be added to the `AggregationStyle` enumeration at the model plug-in. Then, the extension points for compilation unit and routine level have to be modified to offer the new rule. The `MetricReader` class at the package `metrics.util` has to read out the new aggregation rule from the extension point and set it at the respective metric. Finally, the `CalculationHandler` class has to implement the new rule of aggregation.

5.3.3 Adding a new type of result

If a new result type shall be added to the plug-in it has to be a subclass of the existing Result class at the model. To be able to make the result persistent, the new result has to be modeled using EMF and has to have references only to data types that are known by EMF. The extension point has to be modified to make the new result selectable; therefore, the MetricReader class has to be extended as well. The Metric class at the model plug-in has an attribute that stores the type of result. This attribute has o be able to store the new result. It also contains a routine createResult() that returns the selected type of result; this routine has to be modified as well. To visualize the new result an adequate visualization has to be implemented and made selectable from the visualization wizard.

Copyright © Fraunhofer IESE 2006

6 Implemented metrics

The main goal of the metric plug-in developed in this bachelor thesis is the extensibility: It should be possible to add new metrics easily without caring about the details of selecting metrics and elements, storing the results, or visualizing the results. To validate this goal, a set of metrics was implemented and added to the plug-in. Furthermore, a document was created, explaining how to create a new metric and add it to the plug-in (see Appendix). That document was passed to 2 persons of the Fraunhofer IESE who got task to implement a metric just by this information and knowledge about SAVE.

This chapter introduces the implemented metrics that validate the metrics plug-in. Each metric is presented in a structural way, showing its important facts:

Name: the metric's name

Short description: A brief description of the metric, what it measures, and why it was implemented.

Implemented: self-implemented or implemented by others

External tool: yes / no (does this metric use an external metric tool?)

Level: the level of the metric (routine / compilation unit / component / architecture)

Result: simple / list of components

Wizard page: yes / no (does this metric have a wizard page with metric-specific attributes?)

Aggregation rule: addition, average, min, max, or all possible (for compilation unit and routine metrics)

Algorithm: a short description of the implemented algorithm

6.1 Equivalence classes

To satisfyingly validate the metrics plug-in the implemented metrics have to cover all classes of metrics. So, the following table shows all equivalence classes of metric types and which metric covers which equivalence class.

Copyright © Fraunhofer IESE 2006

type metric	Level	Result	Wizard page	Aggregation rule	Uses external tool	Implementation
McCabe	Routine	Simple	X	Average	X	Self
LOC	Compilation Unit	Simple	X	Addition		Self
Fan-out	Component	Simple	X	-		Self
Fan-in	Component	Simple	X	-		Self
Ownership	Component	Simple	X	-		Self
Dependency cycles	Architecture	List of components	X	-		Self
Hierarchy depth	Architecture	Simple		-		Self
Best-mach	Architecture	List of components		-		Self
Fan-out to library	Component	Simple		-		Others
Size in kilobyte	Compilation Unit	Simple		Addition		Others

Table 6.1: Equivalence classes of metrics

As the table shows, the implemented metrics cover all equivalence classes of metric types.

6.2 List of metrics

The following parts of this chapter explain the implemented metrics in the structured way described above. The metrics Fan-in and Size in kilobyte are not explained in detail since they are analogue to the metrics Fan-out and LOC.

6.2.1 McCabe

Name: McCabes' complexity metric

Short description: This metric calculates a routines' complexity by counting the number of branches.

Implemented: self-implemented

External tool: yes: the tool CCCC (see chapter 2.2.) was used

Level: routine

Copyright © Fraunhofer IESE 2006

Result: simple

Wizard page: yes: the user has to specify the path to the files produced by the external tool so the metrics plug-in can read them.

Aggregation rule: all are possible

Algorithm: The algorithm to actually calculate the metric value is delivered from the external tool. The implemented algorithm does the following:

- `Read all files from CCCC that are located at the specified path.`
- `build a hashmap with all routines and their values read out from the CCCC files.`
- `for every delivered routine: if that routine is contained in the hashmap, set the value from the hashmap at the routine's result. If not, don't return a result.`

6.2.2 LOC

Name: Lines of code metric

Short description: This metric counts a compilation unit's lines of code

Implemented: self-implemented

External tool: no

Level: compilation unit

Result: simple

Wizard page: yes: the user has to select if the metric shall count or ignore empty lines

Aggregation rule: addition

Algorithm:

- `open the file of the given compilation unit`
- `set counter := 0`
- `for every line in the file: if the line is not empty or the user didn't select to ignore empty lines: add 1 to counter`
- `result := counter`

Copyright © Fraunhofer IESE 2006

6.2.3 Fan-Out

Name: Fan-out metric

Short description: This metric counts the fan-out of a component; that is the number of relations going out of the component

Implemented: self-implemented; a fan-out metric was already implemented by [Knieling 06], but the metric was re-implemented using a slightly different algorithm.

External tool: no

Level: component

Result: simple

Wizard page: yes: the user has to select what type of relation should be regarded

Algorithm:

- `Set counter := 0`
- `For all outgoing relations of the component: if type of relation = selected type and target component is no subcomponent of the given component then add 1 to counter`
- `Repeat last step recursively for all subcomponents of the given component and the subcomponents' subcomponents.`
- `result := counter`

6.2.4 Ownership

Name: ownership metric

Short description: For analyzed software that is stored in a SVN repository, SAVE offers a mechanism to get the "owner" of every component (the SVN user that performed most of the commits at this component). This metric tells for every given owner name the percentage of components he owns below a given component.

Implemented: self-implemented

External tool: no

Copyright © Fraunhofer IESE 2006

Level: component

Result: simple

Wizard page: yes: the user has to input the name of the owner

Algorithm:

- `Set counter := 0`
- `For all leaf components l below the given component: if there exists a ownership relation from l and the name of the owner is equal to the given name then add 1 to counter`
- `result := 100 * counter / number of leaf components below given component`

6.2.5 Dependency cycles

Name: dependency cycles metric

Short description: This metric checks to all components of a given package if they belong to one ore more relation cycles and gives the number of circles every component belongs to.

Implemented: self-implemented

External tool: no

Level: architecture

Result: list of components: all components that belong to a cycle together with the results' component

Wizard page: yes: the user has to select what type of relation should be regarded

Algorithm:

- `Set cycle-list := null`
- `While not all components of the given package have been checked:`
  - `Take a component c that has not been checked`
  - Set history-list$_c$:= null
  - Check the component c with history-list$_c$:
    - If history-list$_c$ contains c then add history-list$_c$ from the first occurrence of c to the end as cycle to the cycle-list

Copyright © Fraunhofer IESE 2006

        - Else: For all outgoing relations from c: If the target component c2 hasn't been checked, check c2 with new history-list$_{c2}$:= history-list$_c$ + c
- For all components c in the given package:
    - Create a result result$_c$ for c
    - Set result$_c$:= number of cycles c belongs to
    - For all cycles cy c belongs to: add all components of cy to the result$_c$
- Return all results

6.2.6 Hierarchy depth

Name: hierarchy depth metric

Short description: This metric tells for all components of a given package the depth of the component tree below them.

Implemented: self-implemented

External tool: no

Level: architecture

Result: simple

Wizard page: no

Algorithm:

- For all components c in the given packages that have no super-component: perform createResult(c)
- Method createResult(c):
    - For all subcomponents csub of c: perform createResult(csub)
    - Create a new result result$_c$
    - If c has subcomponents: result$_c$:= (maximum of all results of the subcomponents of c) + 1
    - Else: result$_c$:= 0
- Return all results

6.2.7 Best mach

Name: Best mach metric

Short description: This metric compares two packages and gives for every component the most similar component in the other package and the percentage of similarity between both components.

Copyright © Fraunhofer IESE 2006

Implemented: The metric was part of the implementation of [Knieling 06]. It was adapted to the metrics plug-in by delegating the calculation of the results to the already existing component.

External tool: no

Level: architecture

Result: List of component

Wizard page: no

Algorithm:

- `For all components c in the first package:`
- `For all components c2 in the second package: Calculate the degree of similarity by delegation to the existing component and mind the component csim with most similarity to c.`
- `Create Result with degree of similarity for both c and csim.`
- `Add c to the list of components of the result of csim and csim to the list of components of the result of c`
- `Return all results`

6.2.8 Fan-Out to library

Name: Fan-out to library metric

Short description: This metric tells for all components the number of relations going from the component to an external library component.

Implemented: implemented by a student at the Fraunhofer IESE to validate the understandability of the metric plug-in. The student had good knowledge about Java programming, little knowledge about the SAVE tool and no knowledge about the metrics plug-in. After a 20 minute introduction to the metrics plug-in and after reading the explanation "How to add a new metric to the metrics plug-in" (see appendix), it took him one hour to successfully implement the metric and register it at the metrics plug-in.

External tool: no

Level: component

Result: simple

Copyright © Fraunhofer IESE 2006

Wizard page: no

Algorithm: Analogue to the fan-out algorithm

- Set counter := 0
- For all outgoing relations of the component: if target component is an external component then add 1 to counter
- Repeat last step recursively for all subcomponents of the given component and the subcomponents' subcomponents.
- result := counter

Copyright © Fraunhofer IESE 2006

7 Conclusion

This chapter concludes the bachelor thesis with a summary of the work's contents and the results received. After that some open issues and new ideas are presented for future work.

7.1 Summary

The goal of this thesis was the design and implementation of a customizable metrics plug-in for Eclipse in the context of the SAVE tool.

As described in the first chapter, the measurement of metrics is an essential part of analyzing software and software quality. Many tools for the collection and presentation of software metrics exist, but they usually only have a fixed set of metrics that they measure. On the other hand, there exists a great number of different metrics and definitions for metrics. For every viewpoint and purpose, some specific metrics are suitable, others are not. So, a generic metric framework, where customized metrics can be added and removed, was to be designed and implemented.

The implementation of the metrics plug-in designed in this thesis work is based on different technologies, namely Eclipse, EMF, GEF, and BIRT. It is embedded into the SAVE tool, an Eclipse plug-in for the visualization and evaluation of software architectures which was developed at Fraunhofer IESE.

The process of analyzing software using the metrics plug-in consist of four steps: classification of the metric, registering a metric at the plug-in, calculating the metric, and visualizing the metric's results. The calculation and visualization step can be customized using wizards. For the visualization of metrics, the user can select between diagrams showing a single metric, diagrams showing multiple metrics or the visualization of the results within an existing SAVE view.

In the introduction (see chapter 1) several requirements of a metric framework were described. The implemented metrics plug-in fulfils all of these requirements. It is:

- Extensible: New metrics can be classified and added to the plug-in using the extension point mechanism as described in section 4.3
- Flexible: Different types of metrics are supported (see 4.2) and their results can be visualized in multiple ways (see 4.5)

Copyright © Fraunhofer IESE 2006

- Adaptable: Every metric calculation can be adapted by wizards (see 4.4) were the scope of the calculation and the options of the selected metrics can be specified.

- Customizable: All parts of a metric calculation and visualization can be customized using wizards (see section 4.1, 4.4, 4.5)

- Repeatable: Since both the calculation's configuration and results are made persistent, a configuration can be repeated and the result can be visualized repeatable, each time using different types of visualization.

In order to validate the metrics plug-in, a set of 10 metrics was implemented (see section 6). These metrics were chosen to cover all equivalence classes of metric types, so all parts of the metric plug-in are executed by these metrics. Some metrics were implemented by other people at Fraunhofer IESE to validate the understandability of the framework. These metrics could be implemented without much effort: To implement the new metrics and add them to the tool they needed not more time than 1 hour, including about 20 minutes for the understanding of the framework. Most of the problems they faced were due to the algorithm of the concrete metric and not to the handling of the framework's mechanisms.

The maturity of the framework is already relatively high: Fraunhofer IESE already plans to apply the metrics-plug-in and framework in a set of projects with industrial partners to define project-specific measurements.

7.2 Future work

During the implementation of the metrics plug-in, the following ideas and open issues were raised:

- At the moment for a concrete metric the access to results of other metric calculations is not explicitly possible (metrics can read out the results from the given elements, but they don't know the other metric these results belong to; metrics as well can use and invoke other metric's calculation methods). Future versions of the framework should explicitly support this access to make metrics possible that calculate values from a set of other metrics' results.

- The diagrams presenting the metric results are currently only static. In the future, it would be nice to add dynamic aspects to the diagrams, like presenting additional information in a tooltip when the mouse moves over a result, or navigating to the component when the user clicks on its result. BIRT offers some support of dynamic charts. However, since, at the moment, the created charts are stored into picture files and only these pictures, not the

Copyright © Fraunhofer IESE 2006

BIRT charts, are shown in the Eclipse view, the charting component has to be changed severely to use the dynamical aspects of BIRT in the charting view.

- The configuration of the diagrams can, at the moment, only by done by editing xml files or the code. An Eclipse configuration page should be written where attributes like, e.g., color or layout of the charts can be specified.

- In addition, a visualization of the metric values' evolution could be added. At the moment, a single metric can be measured at different times and both results can be visualized in a SAVE view, so the user can compare the change between the metric results. However, the measurement of one metric at different stages of a software system is not explicitly supported. It would be nice to add a functionality to the plug-in which allows for comparing two or more measurements of one metric at one software system and visualize aspects like the components the metric values changed most, or the evolution of the metric results in a line chart.

- At the moment the external tool CCCC is used by starting CCCC manually and delivering the resulting xml-files to the metric using the tool. In addition to that, a metric could be implemented that starts an external tool automatically and cares itself about taking the metric results from the external tool, so the user would not need to do this manually.

- The metrics plug-in is up to now not used in real projects. But Fraunhofer IESE already plans using it in several industrial projects. During daily use new requirements and refinements will arise that need to be integrated into the framework.

Copyright © Fraunhofer IESE 2006

8 Literature

[Basili & Rombach 88] Basili, V., Rombach, D., *The TAME Project: Towards Improvement-Oriented Software Environments*. IEEE Transactions on Software Engineering, June 1988. Reprinted in: Oman, P., Pfleeger, S. (1997): *Applying Software Metrics*. IEEE Computer Society Press

[Bayer et al. 04] Bayer, J., Forster, T., Ganesan, D., Girard, J., John, I., Knodel, J., Kolb, R., Muthig, D. (2004): *Definition of Reference Architectures based on Existing Systems*. IESE-Report, 034.04/E

[Beck 00] Beck, K. (2000): *Extreme Programming Explained*. Addison-Wesley

[BIRT 06] The BIRT Project (2006): *Business Intelligence and Reporting Tools*. http://www.eclipse.org/birt/phoenix/

[CCCC 06] *C and C++ Code Counter* http://sourceforge.net/projects/cccc

[Deming 86] Deming, W. (1986): *Out of the Crisis*, MIT Center for Advanced Engineering Study, Cambridge, USA.

[Eclipse 06] The Eclipse Project (2006): *Eclipse Platform Plug-in Developer Guide*. Eclipse 3.1 documentation: http://www.eclipse.org/documentation/

[EMF 06] The Eclipse Project (2006): *Eclipse Modeling Framework*. http://www.eclipse.org/emf/

[Fetcke 05] Thomas Fetcke: *Software Metrics Sites* http://measurement.fetcke.de/

[GEF 06] The Eclipse Project (2006): *Graphical Editing Framework*. http://www.eclipse.org/gef/

[GOF 95] Gamma, E., Helm, R., Johnson, R,. Vlissides, J,: *Design Patterns*. Addison Wesley

[Grady 94] Grady, R.*: Succesfully Applying Software Metrics*, Computer, Vol.27, No.9, Sept. 1994. Reprinted in: Oman, P., Pfleeger, S. (1997): *Applying Software Metrics*. IEEE Computer Society Press

Copyright © Fraunhofer IESE 2006

[Fenton & Pfleeger 97] Fenton, N., Pfleeger, S (1997): *Software metrics. A rigorous & practical approach.* PWS Publishing Company. International Thomson Computer Press.

[JMetric 00] JMetric project. http://www.it.swin.edu.au/projects/jmetric/

[Knieling 06] Knieling, S (2006): *Static Analysis of Software Architecture Trends with Eclipse*. Diploma Thesis, Fraunhofer IESE

[Lanning & Koshgoftaar 94] Lanning, D., Koshgoftaar, T.: *Modeling the Relationship between Source Code Complexity and Maintenance Difficulty*. Computer, September 1994, IEEE Computer Society

[Lanza 03] Lanza, M. (2003): Object Oriented Reverse Engineering. University of Bern

[Lanza et al. 05] Lanza, M., Ducasse, S., Gall, H., Pizger, M. (2005): *Visualizing Multiple Evolution Metrics*. 27th International Conference on Software Engineering (ICSE 2005), St Louis, Missoure, USA, ACM Press

[Martin 97] Martin, R.: Stability. C ++ Report, 1997

[Metrics 06] Eclipse Metrics plug-in. http://metrics.sourceforge.net/

[Miodonski et al. 04] Miodonski, P., Forster, T., Knodel, J., Lindval, M., Muthig, D. (2004): *Evaluation of Software Architectures with Eclipse*. IESE-Report No. 107.04/E, Fraunhofer IESE

[Naab 05] Naab, M. (2005): *Evaluation of Graphical Elements and their Adequacy for the Visualization of Software Architectures*. Diploma Thesis, Fraunhofer IESE

[Pfleeger et al. 92] Pfleeger, S., Fitzgerald, J., Rippy, D. *Using Multiple Metrics for Analysis of Improvement*. Software Quality Journal, Volume 1, 1992. Reprinted in: Oman, P., Pfleeger, S. (1997): *Applying Software Metrics*. IEEE Computer Society Press

[Rook & Lippert 04] Roock, S., Lippert, M. (2004): *Refactorings in großen Softwareprojekten*. dpunkt.verlag Heidelberg

[Rosenberg 98] Rosenberg, L.: *Applying and Interpreting Object Oriented Metrics*. Presented at the *Software Technology Conference*, Utah, April 1998. http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo.html

Copyright © Fraunhofer IESE 2006

[Rost 06] Rost, D. (2006): *Bericht zum 2. Praktischen Studiensemester*. Hochschule Mannheim. Fakultät Informatik

[Stark & Durst 94] Stark, G., Durst, R.: *Using Metrics in Management Decision Making*. Computer, September 1994, IEEE Computer Society

[Termeer et al. 05] Termeer, M., Lange, C., Telea, A., Chaudron, M. (2005): *Visual Exploration of Combined Architectural and Metric Information*. Technische Universiteit Eindhoven

Copyright © Fraunhofer IESE 2006

9 Appendix

9.1 How to add a new metric to the metrics plug-in

If you want to add a new metric to the tool you should put it into the "computations.metrics.concreteMetrics" plug-in of SAVE and create a new package with the name of your metric. However, if you put a new metric elsewhere, remind that every metric has dependencies to the projects "computations.metrics", "computations.metrics.model", and "model" of SAVE.

The first thing you have to define at your metric is the level it operates on. There are four levels of metrics, depending on the input element it needs for calculation: **Routine metrics**, if the input is a single FSRoutine, **compilation unit metrics**, if the input is a FSCompilationUnit, **component metrics** for SAVEComponents as input, and **architecture metrics**, they take a list of SAVEPackages as input.

9.1.1 Calculator class (required)

For every metric you have to specify a calculator class. This class has to be a subclass of the abstract class "…MetricCalculator". The … stands for the level of the metric as described above, for every level there exists a different abstract calculator class.

All abstract calculator classes have a method "calculateMetric" you have to implement. This method does the calculation of the metric, so write the algorithm of your metric here.

There exists a significant difference between the metric levels at this method:

- For routine, compilation unit and component metrics this method is executed several times – once for every routine, compilation unit, or component that shall be calculated. So at these levels the method has a routine / compilation unit / component and a result as parameters. What the method has to do is calculating the metric value for only this component and saving the value at the result (via "result.setValue(…)". If the metric can't create a result value for the given element, it can throw a ResultCannotBeCreatedException; then the metric plug-in does not store a result for that element and delivers the next element to the calculator class.
- For architecture metrics this method is executed only once. All SAVEPackages the metric shall be calculated on are given to the metric at once and the method has to care itself about creating results. So here nor results are

Copyright © Fraunhofer IESE 2006

delivered, instead the method has to create own results (using the method createResult(component, value) – do not use "new Result()"!). All architectural results have to belong to a specific SAVEComponent, but there exists no restriction what or how many results have to be created, so you can for instance set a result at all components or just at the top-level component of every SAVEPackage.

9.1.2 Registration at extension point (required)

All metrics have to be registered at an extension point of the "computations.metrics" project. There exists one extension point for every metric level, so you have to register your metric at the corresponding extension point. While registering you have to specify the following attributes:

- An ID
- A name for the metric. This name has to be unique among all metrics on the same level
- The calculator class (as described above)
- A short name for the metric. This name will be shown when the metric is visualized within a SAVE view. Only 3-4 letters will be displayed.
- An aggregation rule (only for metrics at compilation unit and routine level): Here you can specify how higher-level elements in the architecture hierarchy like components and subsystems should aggregate their value from the values of their contained elements. Possibilities are addition, average, the minimum, or the maximum value. You can also specify that everything shall be possible - then the user has to select the aggregation rule at every metric calculation.
- A wizard page class (will be explained later) – this is optional
- A result type (will be explained later) – this is optional, if you don't specify a type, the result will be only numeric.

Implementing a calculator class and registering at the extension point you have to do for every metric. The following features are optional; you can use them but as well can ignore them

9.1.3 Wizard page (optional)

If you want your metric to have some metric-specific attributes (like for instance a selection of the analyzed relation types at a fan-out metric, or a path to external files that are needed), you can implement a wizard page class and register it at the extension point. Wizard pages have to be subclasses of the abstract MetricWizardPage (which itself is a subclass of the JFace WizardPage).

Copyright © Fraunhofer IESE 2006

If a wizard page is registered, the user can open it before starting the metric calculation. To pass the attributes to the calculator class there exists the class MetricAttribute which can store strings, booleans, and numeric values. The wizard page has a list of metric attributes it can change, according to the user's inputs.

Every wizard page has to implement the method initializeAttributes(), where the attribute classes have to be instantiated, added to the attribute list and set to their default values. To create a new attribute you have to use the createAttribute() method.

The list of attributes is passed to the calculate method of the calculator class as a parameter, so it can use the attributes. There exists no type safety at the attributes, you can read out a numeric, a boolean, or a string value out of every attribute; so you have to take care yourself that you read the attributes at the calculator class in the same order as you wrote them at the wizard page class.

9.1.4 Results with list of components (optional)

If you want the results to be more complex, you can use results that are able to store a list of components. In this case you have to set the result type to "numeric and list of components" at the extension point. Then you can cast every result object to a ResultWithListOfComponents object at the calculator class. Results with lists of components take a numeric value exactly as the simple results, but here references to components can also be added. If a metric has results with a list of components, a single result of the metric can be visualized in a new SAVE view showing the component of the result and all components of the result's list of components.

9.1.5 Metric initialization (optional)

You can override the initialize-method at the calculator class if you want to. This method is empty in the abstract calculator classes and is executed once before the calculation begins. Here you can put code you want to be executed before the "real" calculation starts, like reading out external files and storing their values as class variables. You have access to the metric attributes, since they are passed to the method as parameters

9.1.6 Setting optimum and tolerable values (optional)

While a metric is visualized within a SAVE view, a small traffic light indicates if the metric value is optimal, tolerable, or not tolerable. The borders for the optimal and tolerable range can be set at the preferences pages. But if you want the metric to have default values for these borders you can set them at the cal-

Copyright © Fraunhofer IESE 2006

culator class. In this case you can override the methods getMinTol, getMinOpt, getMaxOpt, and getMaxTol, which return the minimum / maximum values for the tolerable and optimal range. If you don't override them, the default value is 0 for all.

Be sure that minTol <= minOpt <= maxOpt <= maxTol, otherwise your settings won't have an effect.

9.1.7 Using an external metric (optional)

The use of an external metric tool is not explicitly supported by the metric plug-in. However, you can write a metric that takes its results from an external metric: At the initialize method the metric starts the external tool or reads the files produced by the tool. These values can for instance be stored into a hash-map. The calculator method reads the value from this map and sets it to the result.

An example for the use of an external metric can be found at the McCabe metric.

Copyright © Fraunhofer IESE 2006

Document Information

Title:	Design and Implementation of a customizable Metrics Plug-in in Eclipse
Date:	July 24, 2005
Report:	IESE-091.06/E
Status:	Final
Distribution:	Public

Copyright 2006, Fraunhofer IESE.
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means including, without limitation, photocopying, recording, or otherwise, without the prior written permission of the publisher. Written permission is not needed if this publication is distributed for non-commercial purposes.