Växjö University

Rüdiger Lincke

Validation of a Standard- and Metric-Based Software Quality Model

Creating the Prerequisites for Experimentation

Licentiate Thesis

School of Mathematics and Systems Engineering

Reports from MSI

Validation of a Standard- and Metric-Based Software Quality Model

Creating the Prerequisites for Experimentation

Rüdiger Lincke

Validation of a Standard- and Metric-Based Software Quality Model

Creating the Prerequisites for Experimentation

Licentiate Thesis

Computer Science

2007

Växjö University

A thesis for the Degree of Licentiate of Philosophy in Computer Science at Växjö University.

Validation of a Standard- and Metric-Based Software Quality Model
Creating the Prerequisites for Experimentation
Rüdiger Lincke

Växjö University
School of Mathematics and Systems Engineering
SE-351 95 Växjö, Sweden
http://www.msi.vxu.se

Reports from MSI, no. 07046
ISSN 1650-2647
ISRN VXU-MSI-DA-R--07046--SE

© 2007 by Rüdiger Lincke. All rights reserved.

To my parents Gerd and Gudrun, my brother Volker, and all the people I am connected with by blood or friendship

Abstract

Our long term research goal is to validate a standard- and metric-based software quality model. Today, ambiguous metric definitions lead to incomparable implementation variants in tools. Therefore, large scale empirical experiments, spanning different independent research groups, evaluating software quality models are not possible, since the generalization of the results of individual software measurements is impossible.

We propose a public and extensible knowledge base – a compendium – for the unambiguous definition of software quality metrics and their connection to a standard-based software quality model. This constitutes the basis for the measurement and prediction of software quality in current and legacy systems.

This compendium is published as specification for the metrics implementation. In order to provide well-defined metrics an unambiguous description framework for software metrics had to be developed. It includes formal definitions based on an extensible language-independent meta-model and language mapping definitions. We formalize an existing and well-defined meta-model for reverse engineering. Then, we define some software quality metrics based on this meta-model, as well as language mappings for Java and UML/XMI. A full set of metric definitions together with a quality model is provided in the compendium. Since we want our compendium to be a "living document", we need to maintain the provided knowledge base, the meta-models, language mappings, and metrics. Therefore, we design a maintenance process assuring the safe evolution of the common meta-model. This process is based on theorems controlling changes to the common meta-model, as well as abstraction mechanisms in form of views, hiding modifications to the common meta-model from existing metric definitions. We show feasibility by implementing our definition framework and the metrics specified in the compendium in a well known reverse engineering tool, the VizzAnalyzer. We evaluate our efforts in two industrial case studies supporting our claims and solution.

We suggest that our description framework eliminates existing ambiguities in software metric definitions, simplifying the implementation of metrics in their original sense, and permitting the definition of standardized metrics. The compendium supports exchanging experiments and the evaluation of metrics. The maintenance process allows us to extend and modify definitions, e.g., add new programming languages, in a controlled and safe way. With the tools developed, it is possible to conduct the experiments we envision, and to evaluate the suitability of software quality metrics, in a next step.

***Key-words*:** Software Quality, Metrics, Quality Model, Meta-model, ISO 9126.

Acknowledgments

First of all, I want to thank my adviser, Professor Welf Löwe, without whom I would not be a PhD student in the first place. His invaluable knowledge, guidance, and encouragement made this thesis possible.

I would also like to express my thanks to:

- The Computer Science department at Växjö University for providing a good working atmosphere and for being great and supportive colleagues.
- The Mathematical and System Technical Profile of the School of Mathematics and System Engineering (MSI) for providing financial support.
- The Knowledge foundation for financing my research project "Validation of metric-based quality control", 2005/0218.[1]
- Artisan Global Media (Sören Strömberg, Slavica Stjepanovic), Combitech AB (Kjell Karlsson, Andreas Nilsson), Power Alstom Sweden AB (Per Ranstad), and .windh AB (Emil Eifrém) for cooperation and support in the aforementioned research project.
- Combitech AB (Mikael Olofsson, Lars Svensson) for cooperation and support.
- My friends Constance, Hannah, and Katie for proofreading my papers and thesis, even though scientific computer science articles are not their favorite reading material.
- My colleagues Joakim and Tobias for proofreading my thesis and for providing interesting feedback and valuable discussions.
- My parents, my brother, relatives, and other close friends including new (Sweden) and old (Germany) for providing a rich leisure time and plenty of opportunities to forget about computer science.

To all of you, Thank You!

[1] KK-stiftelsen, Sverige: Projekt "Validering av mätningsbaserad kvalitetskontroll", Diarienr 2005/0218.

Contents

Contents

Part I

The Thesis

Chapter 1

Introduction

Our long term research goal is to validate a standard- and metric-based software quality model. This validation takes place in form of experiments involving industry partners. In this research project,[1] we will validate the automated assessment of the internal quality of software according to the ISO 9126 quality model. In selected real world projects, automatically derived quality metric values will be compared with expert opinions and information from bug and test databases. As a side effect, we create a knowledge base containing precise and reusable definitions of automatically assessable metrics and their mapping to factors and criteria of the ISO quality model.

In these sections, we will provide an outline of our long term research goal giving the context for this thesis. Further, we will define the goals for this thesis and the criteria upon which we will evaluate the success of our present work. We motivate our research and limit the scope of this thesis to the investigation, documentation, and creation of the prerequisites, which need to be done prior to conducting the experiment we are outlining.

1.1 Motivation

Different studies show that currently more than half of the total costs of ownership of a software system are maintenance costs [Erl00, LS81]. Hence, it is important to control software qualities like maintainability, re-usability, and portability directly during the development of a software system.

Software quality is defined in the ISO/IEC 9126 standard [ISO01b], which describes internal and external software qualities and their connection to attributes of software in a so-called Quality Model, cf. ISO/IEC 9126-1:2001. The Quality Model follows the Factor-Criteria-Metric model [MRW77] and defines six quality characteristics (or factors), which are refined into sub-characteristics (or criteria). Sub-characteristics in turn are assessed by metrics; they measure the design and development process and the software itself.

[1] The Knowledge Foundation, Sweden: Project "Validation of metric-based quality control", 2005/0218.

ISO/IEC 9126-3:2003 provides a set of internal metrics for measuring attributes of the six defined sub-characteristics.

These measurements are currently intended to be performed manually as they require human insights, e.g., with code and document reviews. Manual approaches, however, have a series of drawbacks:

1. They are error-prone since they highly depend on the subjects performing the measurement. Hence, they are not measurements in the strict sense, which are required to be objective and repeatable. Humans might oversee or even deliberately ignore certain problems.

2. They are time consuming. When taking, e.g., code reviews seriously, people have to read and understand code that they haven't created in the first place.

3. They might cause tensions in the organizations. There is a conflict of interest when, e.g., the project/quality manager requests reports from a developer, which are later used to evaluate the performance of that developer.

These drawbacks are getting more severe considering current trends in software development, like outsourcing of development and integration of open source components into proprietary systems. For a reliable software production, it is essential to guarantee not only the functional correctness, but also the internal qualities of external components. Manual quality measurement is not an option in these settings.

Finally, many customers of software systems, especially governmental organizations or those operating in security and safety critical areas, demand certified process maturity from their vendors, e.g., as specified in the ISO 9000 series [ISO05, ISO01a, ISO00] or the Capability Maturity Model Integration (CMMI) [SEI06]. They require quantitative reasonable statistical control over product quality as a basis for continuous quality improvement in the software and the software development process. This is, for the aforementioned reasons, hard to establish with manual quality control.

To provide a solution to the previously mentioned drawbacks of the current approaches, we replace the ISO/IEC 9126 metrics manually assessing internal qualities with metrics allowing automatic measurement. By doing this, we resolve the problems described above. We offer a solution which is:

1. less error-prone, since it does not depend on subjects performing the measurements;

2. less time consuming, since, e.g., code reviews are more focused;

3. and neutral to tensions, since it is not persons that assess performance, but neutral machines.

The remaining problem is, however, to ensure that this automated approach is appropriate, i.e., that it provides a reliable assessment for at least some (sub-)characteristics. Our long term research goal is to validate our proposed solution in form of experiments.

In the future, we expect to have a set of tools and methods allowing the automated assessment of software systems. Possible usage scenarios could be:

- monitoring of software quality goals during the development process resulting in early corrective actions;
- assessment of maintenance effort for change or redevelopment decisions;
- control if subcontractors or outsourced software developers meet the agreed quality goals;
- foundation to choose between different software products offered by competing companies. Is the cheaper product really the better choice in the long run?

In all these activities, the tools and methods are expected to indicate bad quality through reports listing outliers, which violate the defined software quality model. With a list of critical modules and classes at hand, reviews can be made more efficient and directed. Managers have the quantitative information needed as input for their management processes to schedule and focus effort on the parts of the system needing more attention than others. Thus, existing resources can be used more efficiently. The success of applied quality assurance measures can be controlled through trends visible in the quantitative data collected. This data can be used, besides supporting management decisions, to justify quality assurance methods toward stakeholders.

Software engineers receive on-time warnings during the development process if they violate architectural contracts, and they are informed if classes grow in complexity, allowing an early refactoring. During maintenance, of legacy systems in particular, the collected data will deliver valuable information for understanding the system, and for planning maintenance tasks.

In any case, manual reviews are needed to validate an issue or identify false positives, but these are much more directed, not relying on intuition. We do not expect to create a fully automated assessment of software quality which can make decisions completely independently of human insight. But we expect to create tools and methods that will prove valuable in supporting

and improving the quality of software development and maintenance efforts, thus saving costs and resources.

1.2 Long Term Goal of our Research

We need to distinguish three kinds of projects. Our long term research takes place in terms of a *research project*, which is supported by the Knowledge Foundation.[2] We replace metrics defined in ISO/IEC 9126 relying on manual assessment of internal qualities with metrics allowing automatic measurement. This defines an adapted Quality Model. We validate the significance of this Quality Model with experiments in four selected *software development (validation) projects* of four Swedish companies[3] ranging from a small software company to a large company from the SAAB group. We create the prerequisites for the research project and the validation projects in the present *thesis project*, which can be seen as a sub-project of the research project.

The research project will deliver research insights along with practical methods and tool support for the participating companies.

On the *research side*, we expect two major contributions:

1. We define an adapted Quality Model, assessing internal quality (sub-) characteristics as defined by an industry standard with well-established metric analyses as proposed by the research community. This quality model is published as a compendium of software quality standards and metrics [LL05] – a knowledge base.

2. We validate the significance of that novel Quality Model, i.e., we corroborate or disprove the hypothesis that static metric analyses allow an assessment of (some) internal qualities of software systems.

Together, (1) and (2) provide the theoretical basis for quality management assessing industrially standardized software qualities in an effective way, since they are significant and objective, and in an efficient way, since they are automated.

On the *practical side*, we produce tools and methods supporting the quality assessment of software under development having a thorough theoretical basis. By implementing them in the participating partner companies, we gain understanding of their practicality and usefulness.

[2] The Knowledge Foundation is a foundation for knowledge and competence development. It is working on making Sweden more competitive, `http://www.kks.se`.

[3] Respecting our Intellectual Property Rights agreement, we do not name the companies together with their projects.

3. We get insights on how our theory, tools and methods integrate with different quality management processes existing in industry. This includes insights on initial efforts for integration and possible/necessary adaptations of these processes.

4. We understand the speed-up in performance of assessing internal quality automatically vs. manually, since we implement both approaches: the manual standard approach for validating the significance of the new automated approach.

Regarding the practical contributions we can expect that the participating companies already have quantitative information available, which can be understood as manual metrics. Of course, the quality of the available information cannot be expected to be sufficient for our evaluation, and modifications will be necessary. Therefore, both approaches will be implemented and adjusted in parallel.

As a side effect, we expect a higher awareness of internal quality issues in participating companies as a first result. We even expect improvements of software quality in the companies as well as improvements in their development process. These effects, however, will be evaluated qualitatively and subjectively in the first place, and not validated statistically.

1.3 Research Problem and Goals of this Thesis

In this section, we define the problem tackled by this thesis. It can be summarized as:

> *The problem tackled by this thesis is the unambiguous definition of Software Quality Metrics, the definition of a standard-based Software Quality Model (SQM) that relates the well-defined metrics to the criteria of the SQM, and the development of appropriate tool support in order to facilitate the conduction of our research experiment in a repeatable and efficient way.*

To solve the problem we need to reach certain goals. Whether a goal is successfully completed can be checked with goal criteria, which will be discussed together with the goals.

The experiment outlined in the previous section is rather complex. To be able to conduct the experiment successfully, a number of prerequisites need to be fulfilled. In particular, we intend to define an adapted Quality Model to which we connect well-established metric analyses. We validate

the significance of that novel Quality Model comparing our own experiment results with results from other research groups. This requires that we are able to define the metrics and the Quality Model in an unambiguous way, so that our results and the results of third party research groups are well defined and directly comparable. Moreover, we plan to extract data from different industrial sized projects. The data will be collected continuously during the development process over multiple versions. Additionally, we want to include data from version control systems, thus reaching back into the history of the software development project. To achieve all this, including integration of the extraction process into existing processes and work flows, the data extraction needs to be automated.

Further, the software projects we are going to deal with in our partner companies are written in different programming languages. This means that we need to define and reuse one and the same metrics on software systems written in these different languages, which so far include C, C#, and Java. At last, we need to be flexible in our approach to allow for extensions with new metrics and programming languages. Not all companies could give us reliable answers about the programming languages in use, and the current set of metrics is likely to be incomplete or inappropriate for our validation, making it necessary to take new elements and relations into consideration, without invalidating existing definitions and implementations.

In detail, the main goals connected with this problem are:

1. **Well-definedness**, that is, definition of the metrics and the measurement process in an unambiguous way allowing only one interpretation during implementation, leading to one and the same result giving the same software system as input, thus making measures repeatable in terms of getting uniquely defined results. This is necessary since we want to compare our results to results from other research groups, and to increase the transparency of the measured data easing their interpretation. This is our *main goal*. It is fundamental for comparing values obtained in different software systems over time to be certain about what they represent. Furthermore, in order to allow external validation of our experiments by independent research groups, it is crucial to define how we measure the values in the software systems.

2. **Standard-based**. The goal of the experiment is the validation of a standard- and metric-based software quality model. Therefore it is necessary to identify and use appropriate standards and metrics within the experiment and to define a connection between the metrics and the standard.

3. **Repeatability** is to permit repetition of each single part of the experiment and the experiment itself, in an **efficient way**. This is necessary since we cannot afford time consuming preparation, redefinition, or reimplementation of metrics, quality models, or tools to adapt them to new software systems or software systems in different languages. This includes automatically conducting the same measurements repeatedly on the same software system, on different versions of the same software system, or different software systems, always and for each instance getting the same results.

The goals described influence each other and a common approach considering all of them at the same time is needed.

In the following we present concrete criteria, which can be used for checking if the previously stated research goals have been met.

Well-definedness This goal is reached if the complete data extraction and metrics computation process is described in such detail that it is possible for different research groups to implement their tools in such a way that different tools measuring one and the same software system calculate the same metric values.

Standard-based This goal is reached if the quality model used is based on an existing standard summarizing the state of the art/best practices of the last decades, thereby gaining a high trust level. The metrics are selected from well known and discussed metrics in current literature.

Repeatability In general, this goal is reached if the complete data extraction, metrics computation, and measurement process can be repeated in an efficient way and at will, leading to the same results for first and third parties. The criteria for this goal can be split into three sub criteria: Automation, Reusability and Maintainability.

Automation It is possible to extract data and compute metrics in an automated way. Thus, once defined sequences of data extractions and metrics computations can be applied in random order and frequency on the same or different software systems, leading to the expected results. The time spent on the initial definition is saved in consecutive uses, and user interaction is reduced to a minimum or not needed at all. It is possible to integrate the measurement process as a fully automated part into the software development process.

Reusability Metrics can be defined and implemented once in a generic way, supported by a meta-model. Briefly, a meta-model defines the

rules and elements according to which models can be constructed, see Section 2.6. This makes the metrics applicable (and thereby reusable) to different software systems written in different programming languages. This means that, e.g., one and the same *Number Of Methods* metric (cf. Section 12.1), can be applied regardless if it is the number of methods in a class provided in Java Source Code or a class in a UML model. The metrics are defined once only, and can be applied to software systems represented in any existing or future software systems and languages, under the condition that the metric makes sense in that context at all.

Maintainability Mechanisms are provided, which allow metrics to be defined and implemented once in a generic way, so that they keep up with changes and extensions in the meta-model without becoming invalidated. This means that metrics are so robust and flexible that they do not break if they encounter new, unknown elements and relations from new programming languages, and that these elements and relations are considered by the metrics if appropriated. E.g., a metric is defined upon a meta-model (Section 2.6) containing classes, it is then extended to distinguish classes and interfaces. The metric just expecting classes must not break, if it finds an interface.

1.4 Context of the Thesis

This thesis is to be seen in the context of the long term research, which was outlined in Section 1.2. To provide some more background about the future application of the outcome of the thesis, we present an overview of our scientific approach, the validation idea, goals and questions, and the participating companies delivering the raw data for the experiments described in the following subsections.

Scientific Approach

This section describes the scientific approach for validating the significance of the metric-based Quality Model. First, we informally summarize the general idea for the validation. Thereby, we also give some examples of the metrics we use. Secondly, we define the precise goals and questions for the validation. Finally, we provide the background for the statistical analysis.

Validation Idea The idea for the validation is to use static and dynamic (metric) analyses applied on the version history of particular software sys-

tems, and additional information sources like bug databases, and even human insights.

To avoid confusion, we distinguish model metrics from validation metrics. The former are automated metrics in the new Quality Model mapped to sub-characteristics. The latter are metrics assessing the (sub-)characteristics directly, but with much higher effort, e.g., with dynamic analyses or human involvement, or a posteriori, i.e., by looking backward in the project history.

Example 1.1. A model metric for the sub-characteristic "Testability" might assess the number of independent paths in a method like the McCabe Cyclomatic Complexity metric [WM96]. A corresponding validation metric of this sub-characteristic might count the actual coverage of test cases for that method. The former can be assessed easily by static metrics; the latter requires dynamic analyses.

Example 1.2. A model metric of the sub-characteristic "Changeability" might assess the change dependency of client classes in a system triggered by changes in their server classes like the Change Dependency Between Classes [HM95] metric. A corresponding validation metric of this sub-characteristic might observe the actual change costs when looking backward in the project history. Again, the former can be assessed easily by static metrics; the latter requires version analyses and human effort in documenting programming cost for changes.

Example 1.3. The model metric of the characteristic "Maintainability" is some weighted sum of the aggregated values of the metrics assigned to the sub-characteristics of "Maintainability" (to be defined precisely in the software quality model). The validation metric of "Maintainability" could compare change requests due to bug reports, bug-fixing changes, and new bug reports, which again requires backward analyses and human annotations.

For each single version in the version history of particular software systems, source and binary codes are available, which form the input to our model metrics. Additional information about bugs reported, test reports informing about failed/passed test cases, and costs of work spent on the system for maintenance or development is available too and can be associated with a certain version. This information is input to our validation metrics. Based on this, we can, on the one hand, determine the quality of each version according to our Quality Model, and, on the other hand, determine the quality based on the additional information. We assume that a higher quality according to our model correlates with fewer bugs, fewer failed test cases, and lower maintenance and development costs. Opposed to this, a low quality,

according to our model, would correlate with more reported bugs, failed test cases, and higher costs for maintenance and development, etc.

Our validation succeeds if software systems having high quality according to our model metrics also have high quality according to the validation metrics and vice versa.

Validation Goals and Questions The project goal is a Quality Model allowing automated metrics-based quality assessment with validated significance.

For validating the significance, we apply the Goal Question Metric (GQM) approach [BR88]. The GQM approach suggests defining the experiment goals, to specify questions on how to achieve the goals, and to collect a set of metrics, answering the questions in a quantitative way.

The goal is to validate the significance of our Quality Model based on the model metrics. Questions and sub-questions are derived from the ISO/IEC 9126 directly:

Q1: Can one significantly assess re-usability with the model metrics proposed in the Quality Model?

Q1.1 - Q1.4: Can model metrics significantly assess operability, learnability, understandability, and attractiveness, respectively, in a reuse context?

Q2: Can one significantly assess efficiency with the model metrics proposed in the Quality Model?

Q2.1 - Q2.2: Can model metrics significantly assess time behavior and resource utilization?

Q3: Can one significantly assess maintainability with the model metrics proposed?

Q3.1 - Q3.4: Can model metrics significantly assess analyzability, changeability, stability, and testability?

Q4: Can one significantly assess portability with the model metrics proposed?

Q4.1 - Q4.2: Can model metrics significantly assess adaptability, and replaceability?

For answering each sub-question, we need both a number of model metrics, which are defined in our Quality Model, and validating metrics, which are defined in our experimental setup, see examples 1.1–1.3.

Background for Statistical Analysis The basis for the statistical analysis of an experiment is hypothesis testing [SSS01]. A null hypothesis is defined formally. Then, the data collected during the experiment is used to reject the hypothesis, if possible. If the hypothesis can be rejected, then intended positive conclusions can be drawn.

Our null hypothesis H_0 states that correlations of model and validation metrics are only coincidental. This hypothesis must be rejected with as high significance as possible. We start for all our analyses with the standard borderline significance level of 0.05, i.e., observations are not coincidental but significant with at most a 5% error possibility. The alternative hypothesis H_1 is the one that we can assume in case H_0 is rejected.

To define the hypothesis, we classify the measured values as high, average, and low. For this classification we use a self-reference in the software systems under development: systems are naturally divided into sub-systems, e.g., packages, modules, classes, etc. More precisely, for each (sub-)characteristic c and each sub-system s:

1. We perform measurements of model and validation metrics.
2. The weighted sum as defined in the Quality Model defines aggregated values $VM(c, s)$ from values measured with model metrics. We abstract even further from these values and instead classify them with abstract values $AM(c, s)$. It is:
   - $AM(c, s)$ = high iff $VM(c, s)$ is among the 25% highest values of all sub-systems,
   - $AM(c, s)$ = low iff $VM(c, s)$ is among the 25% lowest values of all sub-systems, and
   - $AM(c, s)$ = average, otherwise.
3. The validation metrics provide values $VV(c, s)$ for direct assessment of (sub-)characteristics.

Abstraction and normalization from the precise metric values VM to the abstract values AM are necessary, since VM delivers values in different ranges and scales. For instance, the *Line Of Code* (LOC) metric (Section 12.1) has positive (in theory infinite) integer values, whereas the *Tight Class Cohesion* (TCC) metric (Section 12.2) delivers rationale values between 0.0 and 1.0. However, they all have in common that there is a range in which values are acceptable and outlier ranges which indicate issues.

Selecting 25% as boundary values seems to be a suitable first assumption. We expect that the majority of the values vary around an (ideal) median,

whereas the outliers are clearly distinguished. The values will be adjusted if first analysis results suggest other boundaries.

Our statistical evaluation studies the effect of changes in $AM(c, s)$ (independent variables) on $VV(c, s)$ (dependent variables). The hypotheses H_0 and H_1 have the following form:

H_0 : There is no correlation between $AM(c, s)$ and $VV(c, s)$.

H_1 : There is a correlation between $AM(c, s)$ and $VV(c, s)$.

In order to find out which dependent variables were affected by changes in the independent variables, we may use, e.g., the Univariate General Linear Model [Wal02], as part of the SPSS system [Inc05], provided the obtained data is checked for test condition suitability.

The Participating Companies and Projects

Each software system assessed in our project implies other constraints, and the emphasis on the parts of the applied Quality Model varies. This is because the participating companies have distinct expectations and priorities on the quality factors and criteria. Additionally, individual requirements for the application of the quality model result from the architecture and design, programming languages, and development environments.

All selected projects allow quantitative and qualitative analysis. It is possible to look back in time by accessing the software repositories of the participating companies and to observe their long term evolution during the three years the project will go on. During this time, the metrics (Section 12) can be validated empirically for their usefulness, their limits, and their applicability in new areas like web based applications.

The first company is developing Web applications with C# as implementation language, running on .NET Framework (2.0, Windows) and Mono (Linux pendant). Because one and the same application is required to run on both systems, portability is a particularly important quality factor. We plan to assess in particular the adaptability and replaceability characteristics according to our quality model described in the compendium (Section 4.2). Relevant metrics are, among others, WMC, TCC, LCOM, and LOC. Additionally, we need to define appropriate new metrics, assessing peculiarities with C# code written for .NET and Mono, since not all code running on .NET runs without problems on Mono, unless some special rules are followed. The details for these metrics still need to be discussed and formalized with the company.

The second company is developing Web applications with Java 1.4 and JSP for JBoss using Eclipse and Borland Together J. The product is currently in the maintenance phase. The main interest is on assessing the maintainability and on preventing decay in architecture and design. As the maintainability factor is of highest interest, its characteristics of analyzability, changeability, stability, and testability are assessed. The compendium connects them in particular, with LOC, WMC, and CC assessing complexity, and TCC and LCOM assessing cohesion. A particular challenge is that Java Applets, JSP, and HTML are part of the design, which needs to be taken into account when assessing the maintainability of the software system. Other metrics might be included to adapt the Quality Model to the product specific needs.

The third company is developing Web applications with Java. They are currently interested in software quality in general how, it can be automatically assessed, and what it can do for them. They do not have particular expectations or a need for an emphasis on a certain quality factor; therefore, the complete Quality Model, as described in the compendium, will be applied, covering quality factors like (re-)usability, maintainability, and portability, but also reliability and efficiency. Once some higher awareness of internal quality has been achieved, the quality model will be adjusted.

The last company is developing embedded control software with C. Their main concern is the quality of design and maintainability. This means that the focus lies on the maintainability factor with an emphasis on the architecture and design. Suitable metrics assessing inheritance, coupling, and cohesion are NOC, DIT, DAC, CDBC, LCOM, and TCC, as described in the compendium. Complementing these metrics, *design patterns*[4] and *anti-patterns*[5] might become interesting as well.

Currently, we do not expect particular problems with collecting data for the model metrics, since the source code for all projects is available in version management systems. Collecting data for the validation metrics is expected to work without problems as well, but might involve more human effort, since the data is not always available in an easily processable way.

1.5 Summary

In the introduction, we described the outline for an experiment in quality assessment we are about to conduct with industry partners. In contrast to other experiments of this kind, it addresses two concerns, which are usually on the common wish list of experiments in this area: comparing automated

[4] `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`
[5] `http://en.wikipedia.org/wiki/Anti-pattern`

metrics collection to manual metrics collection, and the involvement of industry partners. It should answer the question: Is it possible – in general or to some degree – to automatically assess quality of software as defined by the ISO 9126 standards using appropriate metrics?

We are aware of threats to our approach that are hard to control, like the influence of the projects used for validation (their size, programming language, duration, maturity of company and programmers, etc.) and the validity of the version history and additional information sources. Other problems, like a precise definition of standards and metrics (and their information bases), appear only to be resolvable as a community activity.

Altogether, we aim at entering the discussions on usefulness of such a validation in quality assessment, threats, and possible common efforts.

Further, it became clear that certain prerequisites need to be fulfilled prior to conducting the described experiment. We need to investigate existing work to see if the prerequisites are already fulfilled by some framework or tool, so that we can reuse it for our research; or, if this is not the case, we need to extend existing approaches or develop our own solutions.

1.6 Outline

The thesis is divided into two parts. The remainder of Part I is structured as follows: Chapter 2 provides terms and definitions of frequently used concepts. Chapter 3 investigates related work and evaluates it according to the goal criteria defined in Section 1.3. It describes what the current solutions are lacking in order to satisfy the prerequisites. Chapter 4 describes our answer to the identified problems. Chapter 5 describes how we want to define software metrics. Chapter 6 describes the maintenance process needed to keep the compendium and the metric definitions up-to-date. Chapter 7 describes the tool support implemented. Chapter 8 summarizes the evaluation of the framework in two case studies, and finally, Chapter 9 concludes the thesis and discusses future work.

Part II contains the compendium referenced in Part I, and is structured as follows: Chapter 10 is an introduction to the compendium, discussing how it is used. Chapter 11 describes Software Quality ISO Standards, and Chapter 12 provides definitions for Software Quality Metrics.

Chapter 2

Terms and Definitions

This chapter briefly discusses terms and definitions frequently used in the remainder of the thesis.

2.1 Quality and Quality Management

The term *quality* is a frequently used expression in all kinds of situations, businesses, or every day situations. Despite this, or maybe exactly because of this, a generally applicable, exact, and accurate definition of the term is not easy.

The Microsoft Encarta Dictionary defines quality among other things as *distinguishing characteristic* or *essential property* [Cor03]. In Wikipedia, it becomes clear that quality needs to be seen in a certain context, since it provides, besides its general explanation, context related information for Quality in Business, in Engineering and Manufacturing, in Music, in Phonetics, and in Philosophy [Wik07]. Quality can refer to: A specific characteristic of an object, the achievement or excellence of an object, the essence of an object, the meaning of excellence itself.

This view is supported by the ISO 9000 standard that defines quality as a *"... degree to which a set of inherent characteristics fulfills requirements"* [Bea00, p. 173].

The American Society for Quality defines quality as: *"A subjective term for which each person has his or her own definition. In technical usage, quality can have two meanings: 1. the characteristics of a product or service that bear on its ability to satisfy stated or implied needs. 2. a product or service free of deficiencies."* [fQ07a] This means that the meaning of the term *quality* depends on its usage/context, and often in a technical context, of how a product relates to how well the product serves the needs of the customer.

The German DIN standard 55350 has a similar description of quality: *"Totality of properties or features of a product or an activity, which relate*

to their suitability to satisfy given requirements.[1] *"* This corresponds to the international standard ISO 8402, which defines quality as: *"... the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs."* [ISO94]

All these definitions show that quality is a rather abstract term, requiring a definition according to a certain context. This means that it is necessary to define the given requirements in their context, and if and to which degree the requirements are satisfied. Different procedures/techniques for concretizing the quality term exist, e.g., the Goal-Question-Metric (GQM) or Factor-Criteria-Metric (FCM) methods.

These well known definitions of the term quality give an idea about what quality is, but no description on how to measure or determine quality. Commonly known is only that it is often too late to improve quality, once deficits in a product are detected. Quality has to be realized before the product is finished, thus during its design and development. It is often not possible to build in quality after the product has been finished. Quality has to be part of the product and process right from the beginning.

The process management premise suggests what quality improvement and the more general expression quality management are about: *"The quality of a system is highly influenced by the quality of the process used to acquire, develop and maintain it."* [CMU07a, p. 8] This premise is also supported by the Total Quality Management (TQM), which is a management strategy that aims at embedding awareness of quality in all organizational processes. The TQM movement was established in the late 1980s in the United States as an answer to the Japanese leadership in automobile and electronic markets. Japanese companies focused earlier on processes improvement and therefore American companies could not compete [fQ07b]. The current quality management movement has matured, especially with the intense focus on customer satisfaction [fQ07c].

Our definition of quality within this thesis is ISO 8402. It will be redefined in the context of software in Definition 2.2, Section 2.2. If we talk of quality in general we mean:

Definition 2.1. *"[Quality is] ... the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs."* – ISO 8402 [ISO94].

[1] German original text: *"Gesamtheit von Eigenschaften oder Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht."*

2.2 Software Quality

In the previous section, we discussed the term *quality* in general. We defined quality in Definition 2.1. In this section, we discuss quality in the context of software and its creation/development.

Software quality means, for some, a strict conformance to the requirements; for others it implies that the software subjectively fulfills its practical tasks efficiently and satisfactorily. The *IEEE Standard Glossary of Software Engineering Terminology* defines the discipline of software quality as a planned and systematic set of activities to ensure that quality is built into the software [IEE]. It consists of software quality assurance, software quality control, and software quality engineering. The glossary defines software quality in the following way: *"**[Software] quality.** (1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations."* [IEE, p. 60]

ISO 9126-1 [ISO01b] introduces a more detailed description of software quality. It introduces a standard for Software Engineering Product Quality. This standard views quality from four different viewpoints: Process quality, Internal quality, External quality, and Quality in Use. It is discussed in detail in Section 3.4.

Our definition of software quality within this thesis is based on ISO 8402, in the context of software. The definition has been adapted to suit our understanding and to involve the aspects of internal quality defined in ISO 9126-3 [ISO03], in the form of a *Software Quality Model*, cf. Section 2.3. Our definition of *Software Quality* is:

Definition 2.2. *Software Quality* is the set of characteristics of a software system that bear on its ability to satisfy stated needs. These needs are defined as characteristics and sub-characteristics and possibly assessed by metrics. The ability to satisfy stated needs is expressed in terms of metric values in the context of their characteristics and sub-characteristics.

The notion of Software Quality is operationalized in Quality Models as discussed below.

2.3 (Software) Quality Model

In ISO 9126, a quality model is defined as *"The set of characteristics and the relationships between them, which provide the basis for specifying quality requirements and evaluating quality."* [ISO03, p 66] We will use this definition

when we write about a Software Quality Model (SQM) or short Quality Model (QM). We discuss ISO 9126-3 in detail in Section 3.4, and show the factors and criteria in Figure 4.1. Our detailed description of the software quality model used in the compendium described in Part II, is discussed in Section 4.2 and shown in Figure 4.2.

For clarification, our definition of software quality model within this thesis is based on ISO 9126-3 [ISO03]. The purpose is to assess internal software quality. The software quality model has been adapted to suit our understanding; cf. Section 11, in particular, of a redefinition of usability as re-usability. We document our software quality model in Part II. Our definition is formalized as follows.

Definition 2.3. A *Software Quality Model* is a set of factors, criteria and metrics (characteristics) and the relationship between them. These relations provide the basis for specifying quality requirements and evaluating quality. The relationship between the factors, criteria, and metrics is weighted. Value ranges are defined for evaluating if needs or expectations are met.

2.4 Good vs. Bad Quality

As we discussed in Section 2.1 and 2.2, the meaning of the term quality depends on its context. In our context, the quality requirements to meet are the characteristics summarized in a Software Quality Model. For each characteristic, there is a (maybe context related) range of desired and undesired values defined. We desire a good/high quality and we talk of good/high quality if the values of the characteristics we measure for a particular software system are within the range of desired values. We do not desire bad/low quality and we talk of bad/low quality if we measure values which are in the range of undesired values. Therefore:

Definition 2.4. *Good/high quality* describes a software system fulfilling many or all of the requirements expressed in its assigned quality model.

Definition 2.5. *Bad/low quality* describes a software system fulfilling few or none of the requirements expressed in its assigned quality model.

2.5 Software Metrics

A *Software metric* is a measure of some property of a piece of software or its specification. Its purpose is to provide quantitative data as input to quality assurance and other software development processes, since quantitative

information is necessary to control projects, or as Tom DeMarco expressed it: *"You can't control what you can't measure"* [DeM82, p 3]. A detailed discussion of existing software metrics can be found in Section 3.1 and 3.2. Theory about software metrics, in particular about object-oriented metrics, is discussed by M. Bauer in Chapter 2.1.1 and 2.1.2 of the FAMOOS Reengineering Handbook [BBC+99, p 22]. There he summarizes the definition of software metrics as software metrics measure certain properties of a software project by mapping them to numbers according to well-defined, objective measurement rules. The software metrics relevant in our research and their description are presented in Chapter 5 and Chapter 12.

In the context of our thesis we limit ourselves to metrics measuring properties of software systems with the purpose of assessing the quality of the software system. We define a software quality metric, or just short metric, as follows.

Definition 2.6. A *software metric* is a measure of properties relevant for the determination of the internal quality of a piece of software or its specification. It can be measured from binary code, source code, or other representations like UML models, or other suitable representations of software.

2.6 Meta-Models

A *meta-model* is an explicit model, which describes how to build other specific models. It provides the constructs and rules needed for building these specific models. The meta-model and specific models are often domain specific. Some informal discussion regarding meta-models in relations to other terms can be found in [Pid03]. We will discuss different meta-models in Section 3.3, and we present our own meta-model in Section 5.2 and 5.3.

In the context of our thesis we define (specific) models and meta-models (explicit models) as:

Definition 2.7. A *model* is a set of entities and relations abstracting from program constructs and their relations.

Definition 2.8. A *meta-model* defines all possible models by defining admissible entity types and relation types.

We will distinguish different meta-models for front-end specific models, common models, and metric specific models.

2.7 Language Mapping

A *language mapping* is a set of defined functions specifying how elements belonging to one meta-model are mapped to elements of another meta-model. The theory behind this is described in detail in Section 5.2.

In the context of this thesis, the source set, for which a mapping is specified, is typically a programming language and its elements.

Definition 2.9. *Mapping functions* map elements from a front-end specific meta-model to elements of the common meta-model. They also map elements from the common meta-model to elements of the views we use for defining the metrics.

2.8 Validation

Validation confirms or rejects the correctness of an implementation according to its specification. Correctness means that the implementation fulfills its purpose in the specified way and delivers the specified results. For instance, the implementation of an algorithm delivers exactly, for a particular input the output as defined by the specification of the algorithm.

In the context of this thesis, we define validation as:

Definition 2.10. *Validation* of a software metric confirms or rejects the correctness of a given implementation of that particular software metric regarding its specification. That is, the calculated metric values correspond to values which are expected by the metric specification.

2.9 Evaluation

Evaluation confirms or rejects correlation among measured values and other attributes. Once a metric is successfully evaluated, it is possible to assess, control, and predict these attributes from the metric values.

In the context of this thesis, we define evaluation as:

Definition 2.11. *Evaluation* of a software metrics confirms or rejects a correlation between internal software attributes assessed by software metrics and other external quality attributes. Software metrics are evaluated in a controlled, reliable, and repeatable (by third parties) way.

Chapter 3

State of the Art

In this chapter, we discuss related work. We investigate existing approaches and describe how far the criteria, defined in the introduction, are met.

We are interested in relating software quality metrics to standard-based software quality models. Our criteria are well-definedness, repeatability, and *validity* (see Definition 2.10) of the metrics, and the use of a well established (standard-based) software quality model, which allows us to draw conclusions from the individual metrics to the product quality through *evaluation* (see Definition 2.11).

For this purpose, we present an overview of current software metrics and their definition. We investigate the metric definitions for a sound and unambiguous definition in Section 3.1. Since the research community already recognized that existing metric definitions are not always well-defined, we summarize existing approaches, aiming at improving the metrics themselves and their definitions, in Section 3.2. Certain research groups found that the unambiguous definition of metrics requires a sound meta-model. Current efforts are discussed in Section 3.3. Since we are interested in using standards for the definition and evaluation of software quality, we present an overview of standards and processes currently used within software quality assessment in Section 3.4. Starting from this, we continue by presenting current approaches regarding software quality models of which some have become standard in Section 3.5. We will discuss the results of our investigation in Section 3.6.

3.1 Software Metrics and their Definition

Software metrics have received much attention in the last three decades, and numerous articles have been published by the research community proposing new metrics, validating and categorizing existing metrics, and discussing whole metrics suites. Older popular works targeting the procedural programming paradigm focused mainly on complexity. They were published in the 70th and 80th among others by Bail and Zelkowitz [BZ88] and Halstead [Hal77], who based their metrics on lexical tokens, by McCabe [McC76], basing the metrics on control graphs, or interconnection among statements

and functions like Henry and Kafua [HK81], McClure [McC78], Robillard and Boloix [RB89], Woodfield [Woo80], Adamov and Richter [AR90]. Since we are more interested in object-oriented metrics, we do not go into details of their metrics.

Among the most popular object-oriented metrics is the metric suite of Chidamber and Kemerer, who proposed a language-independent metric suite for object-oriented design in 1991 [CK91] and evaluated it in 1994 [CK94]. They try to improve on existing metrics, which *"... have been subject to serious criticisms, including the lack of a theoretical base."* [CK94, p 1]. They followed Wand and Weber, selecting the ontology of Bunge as the theoretical base for the metrics. They developed six object-oriented design metrics, which were analytically evaluated against Weyuker's proposed set of measurement principals. The metrics suite contains: Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Object classes (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM). These metrics are defined using natural language, integer counts, simple arithmetic operations, and statistics, which hereafter will be referred to as *simple mathematics*. Viewpoints were presented for all six metrics, which aid in the interpretation of results. Furthermore, all metrics were validated and evaluated analytically and empirically. Yet, for the definition of the metrics no meta-model was presented. As language-independent metrics, they leave much freedom to the user implementing a particular language mapping (cf. Definition 2.9). No implementation details of their validation tools are known. Therefore, the authors neither validate nor evaluate the metrics according to our definitions and criteria. In our understanding, neither is a validation possible, because no sufficient specification is given, nor is an evaluation carried out, because the described evaluation is not repeatable by third parties.

In 1993, Wei Li and Sallie Henry proposed a set of "Maintenance Metrics for the object-oriented Paradigm" [LH93a]. They found that several studies have validated various metrics as useful indicators for maintenance effort in the procedural paradigm, but that there is a severe lack of metrics and studies related to the object-oriented paradigm. In reaction, they evaluated five of six existing metrics from the Chidamber and Kemerer metrics suite [CK94]. These were WMC, DIT, NOC, RFC, and LCOM. CBO was excluded. Additionally, Li and Henry introduced Message Passing Coupling (MPC), Abstract Data Type (ADT), Data Abstraction Coupling (DAC), Number of Methods (NOM), SIZE1, and SIZE2. These were then evaluated against the maintenance effort in two commercial systems. They constructed the language-specific tools, but no information about a meta-model formalizing the metric definitions, the mapping from the programming language,

and implementation details were published. Therefore, the authors neither validate nor evaluate the metrics according to our definitions and criteria. In our understanding, neither is a validation possible, because no sufficient specification is given, nor is an evaluation carried out, because the described evaluation is not repeatable by third parties.

In 1994, Fernando Brito e Abreu and Rogério Carapuça published their set of Metrics for Object-Oriented Design (MOOD) [eAC94]. In their opinion, it was important for achieving widespread practical acceptance of object orientation to aid the management aspect of the software development process with quantitative data. This data can be used to allow for better planning, the assessment of improvements, the reduction of unpredictability, early identification of potential problem, and productivity evaluation [eAC94, p 1]. Their metric set contained eight metrics, which are: Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Coupling Factor (CF), Clustering Factor (ClF), Polymorphism Factor (PF), Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), and Reuse Factor (RF). The metric definitions are based on formally defined functions, set theory, and simple mathematics [eAC94]. Brito et al. present a set of seven criteria, which each metric should fulfill. They validate their metrics theoretically against this set. The authors provide suggestions on the interpretation of the metrics. Although the metrics appear well-defined, the lack of a public meta-model and language mapping (language binding) makes them appear not to be well-defined according to our criteria. Even though Mayer and Hall describe in [MH99b] that language bindings for at least C++ and Eiffel exist, and that other language bindings are in preparation, this does not help as long as this information is not public. The validation described by the authors is not a validation of the metrics according to our definition. In our understanding, no validation is possible, because no sufficient specification is given.

Bieman and Kang propose in their work "Cohesion and Reuse in an Object-Oriented System" [BK95] two new metrics of object-oriented class cohesion. They aim at improving shortcomings of the LCOM metric proposed by Chidamber and Kemerer [CK94]. The proposed metrics are: Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC). They use set theory and simple mathematics for the definition of the metrics. They further state that the measures operate in the language-specific context of the C++ programming language. Despite this information, no meta-model is presented, and the mapping from the programming language elements is not defined. The evaluation described by the authors is not an evaluation as we understand it, because is not repeatable by third parties.

The FAMOOS handbook of reengineering includes a survey of object-

oriented metrics in Chapter 20.3 [BBC+99, p 269-286]. In this chapter, Bär et al. provide an overview of the most present object-oriented metrics defined in the software metrics literature, which are mainly the metrics proposed by Chidamber and Kemerer [CK94], Wei and Li [LH93a], Brito e Abreu et al. [eAC94], and Bieman and Kang [BK95]. They classify the metrics into complexity metrics and metrics measuring the class-hierarchy layout and coupling and cohesion. They provide a definition for each metric together with its main characteristics and criticism found in the literature. As in the original definitions, neither is a meta-model presented, nor is a mapping for different programming languages provided. The authors conclude their overview with a survey of experimental result reports. They list experimental results from Chidamber and Kemerer [CK94], Basili, Briand and Melo [BBM95], and Bieman and Kang [BK95]. For each of the reports, a summary of the system description, the data collection, and the analysis of the results is presented. These reports do not yet contain a well-defined description of the metrics with a meta-model and a mapping for utilized programming languages. The authors neither validate nor evaluate the metrics according to our definitions and criteria.

In addition to their survey of object-oriented metrics, Bär et al. conduct an experimental study based on the metrics from the literature in Chapter 20.4 [BBC+99, p 286-299]. They select only a subset of the metrics presented earlier. The selection process, results, and conclusions of the study are described. No insight on how the metrics have been implemented is given. Therefore, the authors neither validate nor evaluate the metrics according to our definitions and criteria. In our understanding, neither is a validation possible, because no sufficient specification is given, nor is an evaluation carried out, because the described evaluation is not repeatable by third parties.

The authors of the FAMOOS handbook see various problems with the described and evaluated metrics and propose a set of new metrics which are more suitable for their purposes in Chapter 20.5 [BBC+99, p 299-313]. They begin by motivating their approach and presenting guidelines for the definition of metrics; then they introduce a concept of multi-layered systems of metrics, and finally they conclude by presenting the results of an experimental study. They present their opinion that there are three main causes why software metrics often do not help and why they feel confirmed by the literature. Citing Briand et al. [BMB96, BDW99], they point out as their first reason that there is a constant lack of following a rigorously specified process for defining software measures, even though Fenton [FP98] and Whitmire [Whi97] already proposed such a rigorous process for defining metrics to counter this particular problem. The authors suggest that the use of a standard terminology or formalism would eliminate the ambiguities found in most

metrics, thereby increasing trustworthiness and usefulness [FP98, BDW99]. As their second reason they point out that metrics are currently very scattered. There is no common order among metric definitions. That would be a prerequisite for comparing metrics and grouping similar metrics. At last, they draw conclusions from their surveys, stating that there is a lack of consistent experimental results for metrics. These results would indicate if it is really useful to apply a particular measure [BBC+99, p 300]. To improve on these issues, they declare their *"...[our] goal is to define in a rigorous manner a system of object-oriented measures, proving that it represents an adequate frame for defining further measurements that evaluate the different aspects of reuse by inheritance, and showing how this system can be applied to the re-engineering process."* [BBC+99, p 300]. In the following, they present definitions and notations for a multi-layered system of metrics consisting of three layers and four metrics, which are: Unitary RAM Metric (RAM_{unit}), Percentage RAM Metric (RAM_{perc}), CAR Metric (CAR), and TRA Metric (TRA). Despite the strict formalism used for the definition of the metrics, a meta-model and a mapping for used programming languages is not provided. The evaluation described by the authors is not an evaluation as we understand it, because the described evaluation is not repeatable by third parties.

Among the less popular metric suites, we find the ones from Kenneth Morris, who proposed one of the first metrics suites for Object-Oriented Software Development Environments in 1989 [Mor89]. With his thesis he aimed at providing a set of metrics, which can adequately support productivity measurement in an object-oriented development environment. The proposed metrics are: Methods per Object Class (average, maximum and minimum), Inheritance Tree Depth, Degree of Coupling Between Objects (average, maximum), Degree of Cohesion of Objects, Object Library Effectiveness (Existence of a Object Library, Percent of Library Objects in an Application, Average Number of Times a Library Object is Reused), Factoring Effectiveness, Degree of Reuse of Inheritable Methods (Percent of Potential Method Uses Actually Reused, Percent of Potential Method Uses Overridden), Average Method Complexity, and Application Granularity. Together with the metrics definitions, he describes so called impact variables and their effects (interpretation). This is similar to a quality model. He does not provide a meta-model or mapping for the programming languages, therefore the results are neither repeatable nor comparable. Morris does neither validate nor evaluate the metrics according to our definition and criteria.

In [WYF03], Washizaki et al. proposed one of the more recent metrics suites dedicated to component-based development. Their main goal was to measure the reusability of components in order to realize the reuse of com-

ponents effectively. For that purpose, they propose five metrics measuring the reusability of black-box components based on limited information obtainable from outside of components without considering any source code. They created a software quality model relating the metrics to re-usability, understandability, adaptability, and portability. They evaluate their metrics. Their metrics are: Existence of Meta-Information (EMI), Rate of Component Observability (RCO), Rate of Component Customizability (RCC), Self-Completeness of Component's Return Value (SCCr), and Self-Completeness of Component's Parameter (SCCp). For each metric, they provide a semi-formal definition, a confidence interval, their consideration, and example. They developed a tool supporting the validation and evaluation of the metrics, but no meta-model is provided. They use an inexplicit language mapping for the Java programming language. Although they provide extensible examples, there is still room for ambiguity and misinterpretation. Therefore, the authors neither validate nor evaluate the metrics according to our definitions and criteria. In our understanding, neither is a validation possible, because no sufficient specification is given, nor is an evaluation carried out, because the described evaluation is not repeatable by third parties.

In [Bal96], Balasubramanian introduces and validates three improved and new metrics for Object-Oriented Software. He selects metrics from the Chidamber and Kemerer metric suite [CK94], and discusses them with respect to the requirements stated by Weyuker [Wey88]. As a result, he proposes two improved metrics, namely Class Complexity (CC) and Cohesion Ratio (CR), and Weighted Method Send Out (WMSO) as a new metric. He describes the metrics semi-formally using natural language and simple mathematics. To evaluate the metrics he seems not to use tool support, but rather calculates WMC, LCOM, and RFC of Chidamber and Kemerer's suite, and his metrics in relation to CC, CR, and WMSO manually, according to his interpretation of the metrics. He does not provide a meta-model or language mapping information. Furthermore, the semi-formal specification leaves room for ambiguity and the human factor, making the results neither repeatable nor comparable. The evaluation described by the author is not an evaluation as we understand it, because the described evaluation is not repeatable by third parties.

In [LC00], Young Lee and Kai H. Chang propose a quality model for object-oriented software and a set of suitable metrics. Their goal is to be able to measure the quality of object-oriented programs in terms of maintainability and reusability. For that purpose, the authors perform an extensive survey of existing metric suits and approaches to measure software quality. They decide to focus on maintainability and reusability by defining their criteria and attributes, according to which they form a *metric model*. They propose the following metrics with their rationale: Internal Class Com-

plexity (ICC), External Class Complexity (ECC), Class Cohesion Complexity (CHC), and Class Coupling Complexity (CPC). The metrics are defined using natural language and simple mathematics, allowing ambiguity and misinterpretation. Lee and Chang do not provide a meta-model or language mapping information. The authors neither validate nor evaluate the metrics according to our definitions and criteria.

As we can see, software quality metrics have over a 30 year long history. The first metrics were published in the 70s. They targeted the procedural paradigm focusing mainly on complexity. Since the rise of the object-oriented paradigm there exists a large body of object-oriented software metrics, which have been developed and described in the literature from the beginning of the 90s. All of the metrics reviewed were defined using natural language or semi-formal descriptions, involving set theory and/or simple mathematics. Their implementations in tools were mainly language-dependent, and details about the implementations were not published. None of the described validations and evaluations were performed according to our definitions and criteria. Thus, none of the metrics discussed in this chapter satisfies our well-definedness and repeatability criterion, which will be summarized in Section 3.6.

3.2 Criticism and Improvement of Metrics and Definitions

As we can see, there exists a large body of software metrics, which aim at quantifying software quality. Many of these metrics have been discussed and validated, theoretically and in experiments, by the scientific community, and have been criticized. Some researchers could support existing metric definitions in their evaluations; others criticize these definitions and propose improvements in definition or completely new metrics.

The problem of ambiguity, which many metric definitions suffer from, becomes most obvious by looking at the correspondence between Churcher and Shepperd [CS95] and Chidamber and Kemerer, which was needed to clarify some of the metrics proposed in [CK94]. The authors point out that the *"... success has been limited by factors such as the lack of sound models, the difficulty of conducting controlled, repeatable experiments in commercial or military contexts, and the inability to compare data obtained by different researchers or from different systems."* [CS95, p 1] Furthermore, they remark that *"... development and acceptance of software metrics as a discipline has been hampered by the indifferent quality of some experimental work and a*

lack of standardization of techniques and terminology." [CS95, p 1] They further admit that their own work suffers from a lack of standard terminology and practices for comprehending and comparing existing work. They underline the importance of taking the varied nature of object-oriented languages into account and stress that composite or indirect metrics are compromised if there is room for interpretation in the definitions of the primitive counts they are based upon. For supporting their claims, the authors show exemplary different competing interpretations of applying the Number Of Methods (NOM) metric [CK94] in an example. They assume that the authors share without question their interpretation, which should be made public to improve the usefulness of these metrics.

In [HM96], Hitz and Montazeri agree with the concerns of Churcher and Shepperd [CS95]. They focus on the compliance of the Chidamber and Kemerer's metrics [CK94] with measurement theoretic principals. The authors demonstrate how the careful consideration of measurement theory allows indication of the strengths and weaknesses of some of the metrics proposed in [CK94]. In particular, they point out deficiencies regarding the CBO and LCOM metrics. To improve on the discussed issues, they propose a formal process for defining measures. Hitz and Montazeri conclude that specifying the mapping from a language-independent set of metrics to specific programming languages and sets of observations is important. The proposed process on its own does not satisfy our demands for a formal definition of the metrics upon a meta-model and language mappings. Therefore, the process still leaves room for ambiguity.

In [BK95], Bieman and Kang criticize Chidamber and Kemerer's Lack of Cohesion in Methods (LCOM) metric as not being effective at distinguishing between partially cohesive classes. They demand cohesion measures that are sensitive to small changes to be able to evaluate the relationship between cohesion and reuse. In their paper [BK95], they develop sensitive class cohesion measures, as described in more detail in Section 3.1.

In [HCN98b], Harrison et al. conduct an empirical experiment supporting the usefulness of Chidamber and Kemerer's set of object-oriented software metrics. They motivate the need for such an experiment with the fact that current evaluations do not use a variety of large scale systems from different application domains. To improve on that, they select three large scale software projects/systems from industry for which they have project metrics available. The authors used both automated tools and manual data capture techniques in order to collect the data and calculate the metrics. They selected a subset of the Chidamber and Kemerer metric set, namely WMC, DIT, NOC, and CBO, as product metrics, and certain other metrics as project metrics. For details refer to [HCN98b, p 8]. Harrison et al. conclude from

the empirical analysis that there are correlations between the used metrics and certain dependent variables, thereby supporting their hypothesis that the metrics are useful. Yet, the authors present no hints on how they interpreted the definitions of the metrics used. Since the metric definitions themselves lack a meta-model, a strict formal definition, and language mappings, we can only speculate about how they were implemented in the automated tools. Repeating the experiment or comparing it with other experiments is difficult or impossible. The authors neither validate nor evaluate the metrics according to our definitions and criteria. In our understanding, neither is a validation possible, because no sufficient specification is given, nor is an evaluation carried out, because the described evaluation is not repeatable by third parties. Therefore, it is difficult to generalize the results of the experiment. Consequently, conclusions generally supporting the Chidamber and Kemerer metric suite cannot be drawn.

In [BDW99], Briand et al. describe that despite the fact that there exists a rich body of work regarding object-oriented metrics, in particular about coupling, there is still little understanding of the motivation and empirical hypotheses behind many of the metrics. The relationship between the metrics and when they can be applied is not clear at all. This prevents practitioners and researchers from obtaining a clear picture of the state of the art. Therefore, it is difficult for them to select or define measures for their application. In particular, they criticize the fact that many metrics suffer from a lack of standardized terminology and a formalism for defining them in an unambiguous and fully operational form, relatively few of them are based on explicit empirical models, and an even smaller number of measures have been empirically validated. The authors address this issue by providing a standardized terminology and formalism for expressing metrics, thereby facilitating the expression of metrics in a consistent and operational manner. Furthermore, the authors review existing frameworks and metrics for coupling measurement and providing a structured synthesis. And they propose a unified framework classifying all considered existing measures, thereby improving on the issues discovered in their review. This allows Briand et al. to contribute to an increased understanding of the state of the art, allowing others to compare measures and their potential reuse, to integrate existing measures, and to facilitate more rigorous decision making when defining new metrics or selecting existing metrics for a specific purpose. Furthermore, the authors conduct an empirical validation study comparing works of Li and Henry [LH93c], Li et al. [LHKS95], Basili et al. [BBM96], and Briand et al. [BDM97]. The authors conclude that the metrics used are indeed useful predictors of fault-proneness or maintenance effort. Yet, they limit the validity of the results for relatively small and stable systems. The usefulness in

large systems and/or for stable system design information before the system is implemented remains to be tested. In our opinion, this empirical validation study is neither validation nor evaluation according to our definitions and criteria. It is rather a meta-study comparing the results presented by the other authors. The experiments or case studies are not repeated, which is anyway not possible since a meta-model and language mappings, as well as well-defined and formal metrics definitions, are lacking in all works. For that reason, it appears as if this study compares apples with pears, that is, the individual interpretation and implementation of the respective authors.

In [HM95], Hitz and Montazeri identify some inadequacies with respect to how certain approaches handle coupling and/or cohesion in object-oriented systems. The authors mainly criticize the fact that Chidamber and Kemerer's definition of coupling [CK91] has certain deficiencies, notably that it does not scale up to higher levels of modularization. Also, and more importantly, it is ambiguous and leaves room for interpretation or speculation, since their definition leaves certain aspects open, e.g., they point out that *"... it is not clear whether messages sent to a part of* self *(i.e., an instance variable of class type) contribute to CBO or not."* [HM95, p 3]. As an improvement they provide a framework for a better distinction between object and class level coupling, and a set of metrics improving existing metrics of Chidamber and Kemerer [CK94] and Li and Henry [LH93c], according to this framework. The proposed two new coupling metrics are: Change Dependency Between Classes (CDBC) and Locality of Data (LD). Also, as improvement of Chidamber and Kemerer's LCOM metric [CK94], the authors define Improvement of LCOM (ILCOM). They perform a theoretical validation of their metrics. The authors define their metrics using natural language, examples and simple mathematics. They do not provide a meta-model or language mappings. Therefore, their definitions of metrics leave room for interpretation making them ambiguous. The authors neither validate nor evaluate the metrics according to our definitions and criteria.

Subramanyam and Krishnan conducted an empirical experiment in order to analyze the Chidamber and Kemerer metrics for object-oriented design complexity. Their work is published in [SK03] and provides empirical evidence that object-oriented design complexity metrics are useful for determining software defects. In relation to previous studies, they conduct their analysis on a much larger system (over 700 classes), and they consider two popular programming languages (C++ and Java). In contrast to many existing studies, they take into account the size factor of software. The authors follow the original definitions of the metrics for the extraction of the metrics information. As we will argue in Section 5.1, we believe that the metrics are ambiguously defined, since they lack formalism, a meta-model, and language

mappings. Therefore, the empirical analysis reflects only an analysis of the Chidamber and Kemerer metric suit [CK94] according to the interpretation and implementation of the authors, but the results cannot be generalized.

In [MH99a], Mayer and Hall describe their findings in a critical analysis of Chidamber and Kemerer's suite of language-independent, object-oriented design metrics [CK94]. They highlight a number of problems arising from these metrics and question the applicability of Weyukers' axioms [Wey88] on object-oriented metrics, and thereby the theoretical validation of the metrics. Furthermore, they explain their general criticism of object-oriented metrics based on unclear definitions and a general lack of terminology, i.e., complexity, cohesion, and coupling, which result in a misconception about considering metrics wrongly as complexity metrics. Furthermore, the authors argue that the metrics fail to capture relevant OO-specific attributes, in particular data-hiding, polymorphism, and abstraction.

In addition to their analysis of Chidamber and Kemerer's suite, Mayer and Hall analyzed the MOOD metrics suite from Brito et al. [MH99b]. They performed a theoretical validation showing that the metrics proposed have severe flaws, which need to be improved before a serious empirical validation and evaluation can take place.

In [HCN98a], Harrison et al. conduct a theoretical and empirical validation of the MOOD metric set [eAC94]. In particular, they evaluate the metrics AHF, MHF, AIF, MIF, CF, and PF on three different releases of one and the same project, and one release of a different project. The projects are rather small, having less than 10 KLOC and not more than 13 classes. Their results suggest that the metrics can be useful for an overall assessment of a software system. However, the authors also issue a warning that their results cannot yet be generalized and that further empirical studies need to be conducted. In order to generalize the usefulness of metrics, comparative empirical studies are needed. Prerequisites for these are that the results are directly comparable and that the experiment settings can be replicated. This is still problematic, since the MOOD metrics are well-defined, but a meta-model and language mapping, making the experiments transparent, are missing. Thus, the discussed experiment shows only a validation of Harrison et al.'s interpretation and implementation of the MOOD metrics. The authors neither validate nor evaluate the metrics according to our definitions and criteria.

As we learned from the works discussed in this section, a number of authors have recognized shortcomings in the definition of metrics. They state that most of the existing metric definitions leave room for interpretation. To tackle this problem, they propose formal definitions or definition frameworks. Besides that, the need for empirical experiments showing the usefulness of ex-

isting metric definitions has been documented. Some of the authors perform comparative empirical studies. Yet, the results are problematic, since they are based on the interpretation and implementation of the metrics by the authors. No formal definition framework, including a sound meta-model and language mappings have been involved. None of the described validations and evaluations were performed according to our definitions and criteria. Therefore, the metrics used for validations were not well-defined, and the experiments are not repeatable by third parties. This holds for all empirical studies referenced by us and by Subramanyam and Krishnan (Li and Henry, 93 [LH93c]; Basili et al. 96 [BBM96]; Binkley and Schach, 98 [BS98]; Chidamber et al., 98 [CDK98]; Briand et al., 99 [BWIL99]; Tang et al. [TKC99], 99; Briand et al., 00 [BWDP00]; Cartwright and Shepperd, 00 [CS00]; El Emam et al., 01 [EBGR01]).

3.3 Meta-Models and Language Mappings

In several efforts, the need for language-independent metrics has been addressed by the scientific community, defining meta-models for the purpose of metrics evaluation and reverse engineering. Meta-models (cf. Definition 2.8) can be seen as a common consensus on how to create models, defining rules, elements, and relations. They are necessary to achieve interoperability and to allow information exchange. Only if two programs are based on the same meta-model can they exchange models and achieve equal metric definitions, cf. Section 2.6. A language-specific design of the meta-model also makes analysis inherently language-specific. Adapting them to support other languages requires changes to all parts of the system: the basic analysis for parsing source code, the design of the language-specific meta-model, as well as higher-level analyses using this information. In practice, such an adaptation is very expensive.

The FAMIX meta-model developed by Lanza and Ducasse in the context of the European Esprit Project FAMOOS [BBC+99] resulted in the creation of the Moose Re-engineering Environment, which is based on the language-independent FAMIX meta-model [LD02]. The authors restricted themselves to the field of object-oriented re-engineering and object-oriented metrics, suitable for metrics covering most of the relevant aspects of object-oriented software. In practice, this framework deals with information extracted from software written in C++, Java, Smalltalk, and Ada [DTS99], mapping the source code to a language-independent representation. The framework contains more than 50 different metrics of which approximately 30 are language-independent. The language-dependent metrics are explic-

itly marked as such. The computation of the language-independent metrics is based on the repository of software entities and is therefore by definition language-independent. As stated in [LD02], coupling and cohesion metrics are currently not supported by this framework, which is a result from the lack of consensus on how to define many of the metrics in this category. Even though the FAMIX meta-model is defined in detail, the mapping of the single language elements in form of front-end (or the context of FAMIX language plug-ins) is not explicitly provided, allowing ambiguity in the interpretation of the model elements.

Abounader and Lamb proposed a data model for object-oriented design metrics [LA97]. They performed an investigation of existing object-oriented metrics and classifications thereof. On this basis, they designed a data model for a database of design information from which most of the proposed object-oriented metrics can be computed. They provided a summary of entities and relationships, but left the information extraction and the mapping of the elements provided by the different tools/front-ends out of their consideration. Therefore, they introduce ambiguity. Extensibility was not explicitly part of their focus.

Briand et al. describe a unified framework for coupling [BDW99] and cohesion [BD97] measurement in object-oriented systems. Their framework allows for exact definition of metrics using relational algebra, but no language mapping is defined. Extensibility was not part of their focus, new frameworks must be developed for other kinds of metrics.

Wilkie and Harmer created tool support for measuring complexity in object-oriented systems [WH02]. The meta-model used by the tool is a relational database schema. Analyses (metrics) are described as SQL queries. The extensible meta-model can hold new language elements and new metrics can be added by writing appropriate SQL queries. No language mapping information is provided.

Reißing proposed a formal model for object-oriented design measurement [Rei01]. His model is based on a UML meta-model. Metrics are defined using natural language expressions, and the mappings from the front-ends to the formal model are not part of his work, thus the ambiguity problem is not solved.

El-Wakil et al. present their approach to formalize object-oriented design metrics in [EWEBRF05]. They do not have an explicit meta-model abstracting the information extracted by front-ends. Instead, they use XQuery as front-end and definition language for the metrics, thus calculating the metrics directly on the front-end model capturing the XMI[1] design documents.

[1] XML Metadata Interchange, `http://www.omg.org/technology/documents/formal/`

Their approach is therefore unambiguous, but limited to a single front-end.

In several papers, Baroni et al. describe a meta-model based on UML 1.3 [OMG99] and the use of OCL as metrics definition language [Bar02, BBeA02, BeA03b, BCPeA05]. A tool called FLAME (Formal Library for Aiding Metrics Extraction) implements their efforts and is described in [BeA03a]. Goulão et al. [GeA04] build on their approach with component based metrics and a meta-model based on UML 2.0 [OMG06] as foundation for their definitions. Their approach does not explicitly specify a language mapping.

[MP06] discusses the previous references [BDW99, EWEBRF05, BD97, WH02, Rei01, Bar02, BBeA02, BeA03b, BCPeA05, GeA04] in more detail.

With the Dagstuhl Middle Metamodel (DMM), Lethbridge et al. [LTP04] describe an extensible schema for a static model of software. It captures program level entities and their relationships rather than full abstract syntax graphs or architectural abstractions. Their focus is on providing a sound meta-model. They neither discuss the safe extension of that meta-model, nor on how program information is mapped from front-ends to the DMM. A discussion of further alternative meta-models is given in [SLLL06].

McQuillan et al. recognize the useful mechanisms for the precise definition of software metrics [Bar02, BBeA02, BeA03b, BCPeA05] and propose a generalization [MP06]. They extend Baroni's approach by decoupling the metric definitions from the meta-model. They extend the meta-model with a separate metrics package, containing metrics expressed as OCL queries. This approach allows them to easily extend their tool, which can in theory be applied to any language, and, furthermore, will allow easy re-use of metrics definitions. A prototype tool called DMML (Defining Metrics at the Meta Level) supports their approach. For illustration, formal definitions for the Chidamber and Kemerer metrics suite are presented. This approach is extensible and unambiguous on the meta-model level, but still introduces ambiguity by not specifying the mapping from the front-end meta-model to the Dagstuhl Middle Metamodel they use for capturing model information.

Complementing the efforts of formalizing metrics described in Section 2.6, authors recognized that the formal definition of metrics needs to be based on a meta-model. Different meta-models have been developed and implemented in tools. Yet, some of these efforts are still ambiguous, since they do not provide a language mapping. Therefore, they allow interpretation in the application of the meta-model to a specific language. Others allow a detailed and unambiguous definition of metrics, and with this satisfy our well-defined criterion, but they are language-specific. Therefore, they cannot be applied to systems written in other programming languages, and considerable effort

`xmi.htm`.

would be necessary to adapt them and the metric definitions. For these reasons, none of the presented approaches satisfies both our well-definedness and repeatability criteria.

3.4 Software Quality Standards and Processes

Standards exist for many domains. They make an enormous contribution to our everyday life and are so important that local and international organizations for defining and maintaining standards have been established. Examples are the IEEE, DIN, and ISO standards. As standards exist for most aspects of our lives, there also exist several standards for Software Engineering/Quality. The most important standards with respect to Software Quality currently are: ISO 9000, ISO/IEC 15504, Capability Maturity Model Integration (CMMI), ISO/IEC 12207 and ISO/IEC 9126.

ISO 9000

The ISO 9000 family, consisting of ISO standard 9000-9004, is by far the most popular standard in the area of quality improvement and management [ACN05]. The standard enables companies with high confidence to meet specific requirements. The fulfillment of these requirements aims at increasing customer satisfaction in an ordered and systematic way [Bea00]. Yet, ISO 9000 is not only designated to be used in Software Engineering, but in all kinds of organizations, regardless of type, size and product provided [Bea00, p 13]. For more details about this standard refer to [ISO05, ISO01a, ISO00, Bea00].

ISO/IEC 15504

ISO/IEC 15504 is a standard dedicated to software engineering. It is also an international standard that aims, in particular, at process assessment. This standard and the described process is also known as SPICE (Software Process Improvement and Capability dEtermination) [SPI07]. This standard was inspired by CMMI and ISO 9000, and its purpose is to bring a better integration of different standards like the CMMI, ISO 9000 and ISO 12207 [CMU07c]. The standard itself describes a framework for the assessment of software development processes, without defining the process models or assessment methods themselves. The process models (e.g., CMMI or ISO 12207) and assessment methods (SCAMPI or SPICE) have to be selected separately. The standard further defines reference models and requirements

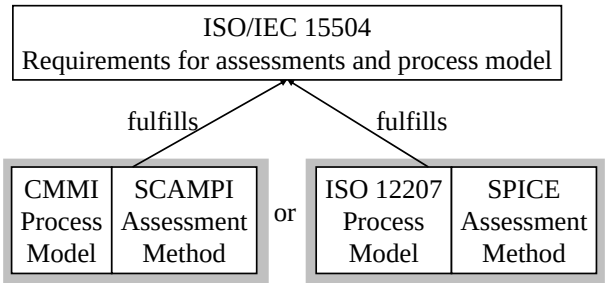

Figure 3.1: Connection between ISO/IEC 15504, CMMI and ISO 12207

for assessment methods. Its process models include the best practices being successfully applied in companies. The connection between the different standards defining process models and assessment methods can be seen in Figure 3.1.

Capability Maturity Model Integration

The Capability Maturity Model Integration (CMMI) has its own assessment method called SCAMPI. In [CMU07a], the editors of the CMMI describe their Capability Maturity Model (CMM) as *"... a reference model of mature practices in a specified discipline, used to improve and apprise a group's capability to perform that discipline"*. [CMU07a, p 13]. The CMMI extends the area of software engineering integrating process improvement for the four disciplines software engineering, systems engineering, integrated product, and process development and supplier sourcing. The model itself states its purpose as follows: *"... to provide guidance for improving your organization's process and your ability to manage the development, acquisition and maintenance of products and services. CMM Integration places proven approaches into a structure that helps your organization appraise its organizational maturity or process area capability, establish priorities for improvement, and implement these improvements."* [CMU07b, p 1]. Briefly explained, the CMMI covers 24 process areas that are relevant to process capability and improvement. They are described and organized in four categories, which are process management, project management, engineering, and support. Specific goals for each of these process areas describe the desirable state that shall be reached by the organization. In contrast, generic goals specify common features and are associated with capability or maturity levels. Practices in the CMMI describe how goals can be achieved. For each goal, up to seven practices recommend

ways to reach it [Som04].

ISO/IEC 12207

Besides CMMI, the ISO/IEC standard 12207 (Information Technology – Software life cycle processes) represents another process model. It is a standard accepted world-wide, which defines requirements for acquisition, supply, development, operation, and maintenance of software. It is applicable to internal and external software development [Fre07]. The standard describes processes with corresponding activities and tasks and their interrelation. Yet, it provides no concrete method on processing the development. For each project, a software development model has to be substantiated. Together with SPICE as assessment method, ISO/IEC 12207 fulfills the requirements of ISO/IEC 15504, which itself incorporates the process model of ISO/IEC 12207 [RFB04]. SPICE defines only one assessment type and is not as differentiated as SCAMPI.

ISO/IEC 9126

Unlike ISO/IEC 12207 and CMMI, which address software engineering processes, ISO/IEC 9126 (Software Engineering – Product quality) addresses software quality itself. It defines characteristics, which represent software quality and a model connecting them with each other. It allows us to provide the aforementioned standards with quantitative information about product quality in order to measure improvement. ISO/IEC 9126 consists of four parts:

- Part 1 (2001): Quality model – Classifies software quality in a structured set of factors. It is based on the Factor-Criteria-Metric (FCM) model [MRW77].
- Part 2 (2003): External metrics – Describes measurements during software execution.
- Part 3 (2003): Internal metrics – Describes static measures.
- Part 4 (2004): Quality in use metrics – Describes measurements (only) applicable to the final system used in real conditions.

In the first part of the standard, a two-part model for software product quality is defined [ISO01a]. It defines functional and non-functional customer and user requirements. Whereas the first part describes external and internal quality characteristics and their relationship in the form of a Quality Model,

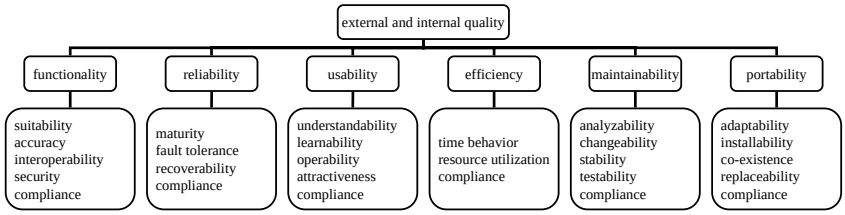

Figure 3.2: External and internal quality factors and criteria of ISO 9126 quality model. Factors are in the middle row, criteria in the bottom row.

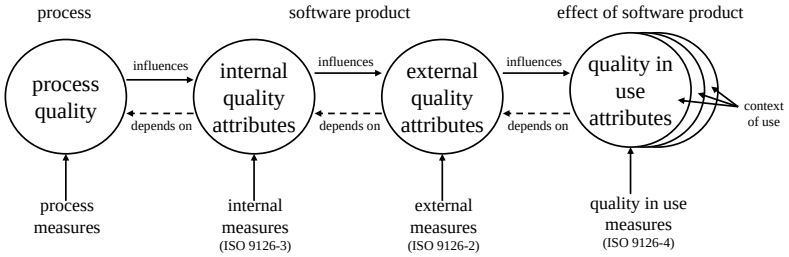

Figure 3.3: Quality in the life cycle as seen by ISO 9126 and ISO 9000

the second part presents such a Quality Model for quality in use. Figure 3.2 is taken from the standard description and illustrates this model.

ISO/IEC 14598 is a standard for Software Product Evaluation. It was part of earlier revisions of ISO/IEC 9126 (1991), but is now a standard on its own. It provides best practices for the software evaluation process, assists in the execution and describes requirements on the evaluation. It should be seen in combination with ISO/IEC 9126.

As already mentioned in Section 2.1, process management premises to fulfill the hypothesis of software engineering, which is that *process quality* and *internal quality* affects *external quality* and *quality in use*. Therefore, it is seen as natural that software development processes influence the software product quality directly, which is expressed by the software product life-cycle, which is illustrated in Figure 3.3. It shows the interrelation of process quality, internal and external quality, and quality in use. The process quality is defined by ISO/IEC 12207. Internal and external quality and quality in use can be assessed quantitatively with ISO 9126.

Therefore, ISO/IEC 9126-1 can be used together with ISO/IEC 15504, ISO/IEC 12207, and ISO 9001 to provide a framework for the qualitative and

quantitative evaluation of software product quality, and to support review, verification and validation activities [ISO01a, pp 1]. The combination of ISO/IEC 9126 with ISO 9001 allows to provide *"support for setting quality goals [and] support for design review, verification, and validation."* [ISO01a, p 2].

Summary

In this section, we presented a number of current standards for quality assurance and assessment. These include general standards like the ISO 9000, which are not only designated for use in software engineering and more specific standards, like the ISO/IEC 15504 which are designed for use in software engineering. Most of the standards presented are used to define requirements of the development process and other related processes, but only ISO 9126 defines a Software Quality Model, which satisfies our criterion of being standard-based.

3.5 Software Quality Models

With the advent of Software Quality Metrics, explicit and implicit Software Quality Models arrived, which tried to relate the values provided by metrics with more abstract quality properties like fault-proneness, maintainability, and reusability. Doing this, they tried to aid the interpretation and synthesis of metrics information, which would otherwise be difficult. Two well known methodologies for developing such quality models are the Factor-Criteria-Metric (FCM) approach by McCall [MRW77], and the Goal-Question-Metric approach by Basili and Rombach [BR88].

All metrics proposed in the last 30 years seek to infer or predict quality-related factors, criteria, and goals and questions from the attributes they measure. This shall then help to support management processes to focus development effort. While the early metrics tried to measure complexity, more recent metrics try to assess properties of the object-oriented paradigm.

While some metrics aim directly at providing information about a certain quality criterion, researchers and practitioners try to express their needs and expectations to software quality in so-called Software Quality Models (SQM). They relate existing and new metrics to a hierarchy of criteria and subcriteria, e.g., according to the FCM. These models clarify how the different factors and criteria contribute to the whole. They make it possible to see the quality of a product as a big picture.

One of the most popular Software Quality Models nowadays is the ISO 9126 Software Quality Model [ISO01b, ISO03], which is based on the FCM

approach from McCall [MRW77]. It distinguishes internal quality, external quality, and quality in use, providing a hierarchy of factors, criteria and metrics. The main issue with the proposed metrics is that they require manual interaction and do not consider software product metrics. This makes them unsuitable for automatic evaluation of software programs. Furthermore, the metrics are not well defined and open for interpretation.

To deal with this issue, certain researchers have provided mappings of existing metrics to software quality models like ISO 9126-3 [ISO03] or proposed new software quality models. Often they did not propose complete quality models, but investigated the usefulness of certain single metrics or whole suits of metrics for determining single quality attributes like fault-proneness, maintainability, and reusability. Some Quality Models are well established and even standardized software quality models, others are rather ad hoc.

Laguë and April decided to use the Software Quality Model defined in the ISO 9126 standard and map the software quality metrics of the DATRIX^{TM} software tool to it. They describe their experiences in [LA96]. DATRIX^{TM}is a commercial application used at Bell Canada and its metric definitions are not public, therefore nothing can be said about how well the metrics are defined or about the (general) validity of the metrics, alone or together with the quality model.

As we see, not too many authors tried to connect well-defined and automatically assessable software metrics with large-scale software quality models like the ISO 9126. Most authors limit themselves to investigating if a set of metrics is suitable for 1–2 quality factors or criteria.

In [LC00], Lee and Chang propose a software quality model for the quality factors maintainability and reusability, which are both dependent on the criteria Complexity and Modularity. Furthermore, they propose a set of metrics, which allow them to assess the criteria. The metrics themselves are semi-formally defined and no meta-model is presented. The factors, criteria, and metrics are not weighted. They implement the quality model and the metrics in their own tool (RAMOOS). A validation or evaluation is neither for the metrics nor the software quality model described.

In [LH93a], Li and Henry describe their findings on the usefulness of the Chidamber and Kemerer metrics and some of their own metrics with respect to maintenance. Their quality model has only one factor (maintainability), which is not connected to the metrics via criteria, but directly connected to the metrics. They evaluated their quality model in a case study with two commercial systems.

In [Kar95], Karlsson proposes a language-specific software quality model for maintainability and re-usability of C++ programs. For each factor, he proposes a set of characteristics and metrics related to the quality criteria.

He proposes normalization of the measures between zero and one. He uses Kiviat diagrams[2] for the presentation of the metrics.

Boehm et al. report on a conceptual framework and quantitative evaluation of software quality in [BBL76]. They describe a Software Quality Characteristics tree, relating Quality Factors to criteria and metrics. They evaluate quality metrics according to correlation with quality, potential benefit, if it is possible to develop automated metrics, and how complete the automated evaluation will be. The definition of the proposed metrics is in natural language and not well-defined.

In [CM78], Cavano and McCall describe their experiences with defining a framework for the measurement of software quality. Their motivation is the lack of a widely accepted definition of software quality, which leads to confusion in specifying quality goals for software, and the fact that determining the quality of software during operation is too late. To improve on that, they propose a process for the early measurement and representation of the qualities of a software system. In their paper, they point out the relative nature of quality and show that there are even competing and contradictory factors and criteria. Their approach for solving these problems is a framework allowing a clear identification and definition of quality factors and criteria, their relationship to measures, and a rating/combination of the factors, criteria, and metrics. The proposed framework facilitates automation and is the foundation of the ISO 9126 standard [ISO01b] software quality model.

In [Dro96], Dromey suggests a framework for the construction and use of quality models. He considers requirements, design and implementation specific criteria. Like the other models, product quality is defined by relating and combining sub-characteristics with characteristics and factors.

In contrast to the static quality models, which are based on human experience, intuition and limited analysis, Kececi and Abran propose a graphical dynamic quality assessment framework [KA01], which is optimized using a logical model for relating the metrics with the characteristics, sub-factors and factors. Their approach provides support for identification of testing objectives, interrelation between software and hardware and other sources of impact, trends in quality, and common measures applicable to more than one quality attribute.

The first Software Quality Models came up with the first software quality metrics and were described already in the 70s, starting with the Factor-Criteria-Metric (FCM) approach of McCall [MRW77]. To date, several software quality models have been proposed. Yet, only few relate their factor and

[2] Also known as spider, radar or star chart. The value of three or more quantitative variables is shown on axes, which originate in the center of the diagram. For an example, we refer to: `http://en.wikipedia.org/wiki/Radar_chart`

criteria to well-defined and automatically assessable software quality metrics. They lack, either completely or partially, evaluation, or they are not described in such detail that the evaluation is repeatable. In some cases they are not public, and only company-internally reference implementations exist. Often, the mechanisms for combining metric values and criteria are not clear. There is only one model among them which is an international standard. It is the ISO 9126 Software Quality Model, which only provides manually assessable metrics, which meet neither our well-definedness nor our repeatability criteria.

3.6 Discussion

This section discusses the results of our investigation of related work. First, we summarize the previous sections, then we draw conclusions.

Summary

Quality Assurance and Quality Management processes and practices, e.g., ISO 9000 family [ISO05] and CMMI [SEI06], are used in industry in order to actively improve the quality of processes and resulting products. Most of them depend on the collection of qualitative and quantitative data used for identifying room for improvement and for controlling the success of applied measures.

Metrics for measuring software attributes and quality models organizing them in a meaningful way have been researched. These efforts resulted in various metrics suites, among others Chidamber and Kemerer's Metrics Suite for object-oriented Design [CK91, CK94] and "The MOOD Metrics Set" [eAC94] and in industrial standards like ISO 9126 [ISO01b] describing a Software Quality Model.

Furthermore, a large set of tools has been developed for facilitating the validation (Def. 2.10) and evaluation (Def. 2.11) of these metrics on several different programming languages, including Ada, C/C++, Java, and other high-level representations, like UML. Yet, the results provided by the different tools are difficult to compare, since there is, so far, no *unambiguous* description framework for metrics. Unambiguous means that given a metric definition and a source program or specification, the result of that measurement is uniquely defined. This prevents the implementation of metrics "by the book" and the discussion of variants, let alone scientifically comparative and repetitive studies. Hence, common validation efforts, e.g., the FAMOOS project [BBC+99], appear to be of limited significance.

An early reason for ambiguities was that data models (meta-models) – a basis for the metrics definitions – were implicit and not included in the definitions [BDW99]. This problem has been noted among others by Churcher and Shepperd who identified ambiguities in the suite of metrics proposed by Chidamber and Kemerer [CK94]. They even needed a dialog with the authors of the metrics to interpret them in a "correct" way [CS95]. The scientific community has addressed this source of ambiguity by proposing common and formally defined meta-models. Metrics were precisely described on these models according to the individual interpretations. Examples are the FAMIX meta-model [LD02], the Data Model for Object-Oriented Design Metrics [LA97], and the Dagstuhl Middle Metamodel [LTP04]. Refer to Section 3.3 for more details.

Furthermore, among all the discussed software quality standards and processes discussed in Section 3.4 and all the software quality models discussed in Section 3.5, the ISO 9126 software quality model [ISO01b] is the only standard-based software quality model of international acceptance. It summarizes the state of the art in software quality models, even though there is some criticism, and other more advanced models exist.

Our findings are summarized in Table 3.1. The first row (Metric) provides an overview of the metrics defined in the literature we reviewed. The second row states whether the metric definitions are well-defined (WD) according to our criterion. The third row states if the metric has been defined upon a meta-model (MM), which is explicitly provided. The fourth row states whether a language mapping (LM) for the meta-model a metric is based on exists, and has been explicitly provided. The fifth row (V) states whether or not the metric has been validated and evaluated according to our definitions (Def. 2.10 and 2.11). The last row (References) provides references to sources in which the metric has been defined, validated, evaluated, discussed or referenced.

Since none of the metrics satisfied our criteria to the full extent, all cells below the column heading WD, MM, LM and V would contain a *no*. To differentiate between the reasons for rejecting a metric according to our criteria, we refer to the reason leading to rejection. The reasons are listed below; their numbers correspond to the numbers in the table:

1. Our well-definedness criterion is not met, since the metric is described semi-formally or using natural language. Therefore, it is open to interpretation. Thus, it is not possible for different research groups to implement their tools in such a way that different tools measuring one and the same software system calculate the same metric values.

2. Our well-definedness criterion is not met. A formal definition is used,

which is an improvement over semi-formal or natural language descriptions. Yet, it still allows for interpretation, since no (public) meta-model and language mapping is defined. Therefore, it is open to interpretation. Thus, it is not possible for different research groups to implement their tools in such a way that different tools measuring one and the same software system calculate the same metric values.

3. No meta-model is defined at all, or it is not public. Therefore, our reusability criterion is not fulfilled. This means that it is not possible to define metrics once only, and to apply them to other software systems written in other languages.

4. No explicit language mapping for a meta-model is defined at all, or it is not public. Therefore, our well-definedness and reusability criteria are not fulfilled. This means that it is not possible to define metrics only once, and to apply them to other software systems written in other languages.

5. Only a theoretical inspection or reasoning was performed.

6. The authors validate or evaluate the metrics according to our definitions and criteria. A validation is possible, because a sufficient specification is given. An evaluation can be carried out, because the described evaluation is repeatable by third parties.

Conclusion

Our investigation of related work shows that none of the existing solutions satisfies all of our criteria. Some are close but not complete, as can be seen in our summary and Table 3.1.

Metric	WD	MM	LM	V	References
WMC	1	3	4	6	[CK91, CK94, LH93a, BK95, BBC+99, Bal96, CS95, HM95, HM96, HCN98b, BDW99, SK03, MH99a]
DIT					
NOC					
CBO					
RFC					
LCOM					
MPC	1	3	4	6	[LH93a, BBC+99, BDW99, HM95]
ADT					
DAC					
NOM					

Metric	WD	MM	LM	V	References
SIZE1	1	3	4	6	[LH93a, BBC+99, BDW99, HM95]
SIZE2					
MIF	2	3	4	5,6	[eAC94, BBC+99, MH99a, HCN98a]
AIF					
CF					
CLF					
PF					
MHF					
AHF					
RF					
TCC	1	3	4	6	[BK95, BBC+99]
LCC					
RAM_unit	1	3	4	6	[BBC+99]
RAM_perc					
CAR					
TRA					
Methods per Object Class	1	3	4	6	[Mor89]
Inheritance Tree Depth					
Degree of Coupling Between Objects					
Degree of Cohesion of Objects					
Object Library Effectiveness					
Factoring Effectiveness					
Degree of Reuse of Inheritable Methods					
Average Method Complexity					
Application Granularity					
EMI	1	3	4	6	[WYF03]
RCO					
RCC					

Metric	WD	MM	LM	V	References
SCCr SCCp	1	3	4	6	[WYF03]
CC CR WMSO	1	3	4	6	[Bal96]
ICC ECC CHC CPC	1	3	4	6	[LC00]
ILCOM CDBC LD	1	3	4	6	[HM95, HM96, BBC+99]

Table 3.1: Summary on literature on metrics

Therefore, we still see the following limitations of the existing approaches:

1. Metrics are specified in either an informal or semi-formal way, which makes them ambiguous. This has been addressed by some researchers, providing meta-models and formal descriptions of metrics, based on these meta-models. Yet, no complete well-defined specification upon a well-defined meta-model has been published and is freely available for other researchers to reuse and to discuss.

2. For the existing meta-models the following limitations hold, in particular:

   (a) The models are ambiguous; because, it is not specified how information from a software system is mapped to the meta-model. Due to this missing language binding, two metric tools – for the same source language, using the same meta-model and metric definition – may measure different things in the same software system.[3]

   (b) Even though meta-models are extensible, so far none of them allows us to control the effects of extensions of metrics definitions. It should be easy to determine if they deliver the same results after a meta-model extension or, otherwise, if a re-definition becomes necessary.

[3]For the language-transparent Middleware Corba, the language binding is part of the standard to avoid this kind of incompatibilities.

3. Several software quality models exist, even including one international standard software quality model. Yet, the metrics proposed with this quality model do not fulfill our criteria for well-definedness and repeatability.

Some additional remarks to address:

- We observed that the way natural language and semi-formal specifications are used in the literature for the specification of metrics bears the danger of ambiguity. We leave it open if natural language and semi-formal specifications are per se ambiguous. Anyhow, we want to point out that in our experience the exclusive use of natural language in specifications makes them usually too long and therefore difficult to comprehend. Unambiguous specifications would require the definition of underlying models, which are often awkward to express using only natural language. This is possibly the reason why specifications in natural language often lack important details, which makes them underspecified in the end, which in turn allows interpretation and therefore leads to ambiguity.
- We use natural language in our metric specifications to make them more readable. We further refer to well-known mathematical principles to avoid over formalization.
- For the sake of completeness, we want to point out that models may be ambiguous not only because of a missing language mapping, but also because of meta-models being ambiguous themselves. This has not yet been observed in our review of literature, because in many cases the meta-models are not provided in an extend that allows to decide whether they are ambiguous or not.
- The way the terms *validation* and *evaluation* are used in literature is inconsistent. This made it difficult to compare the contributions of different researchers. Therefore, we provide our own definitions of validation and evaluation in Sections 2.8 and 2.9.
- None of the papers referenced in this chapter ever discuss the scope of metrics. To limit the scope of metrics on a software system is basis for a successful evaluation.
- None of the empirical validations were performed on bigger systems. With Randis (see Section 8.2) we have a software project consisting of more than 600 classes and 180 KLOC, which is about 10 times larger than the size of the systems the referenced studies have investigated.

- The criticism about missing language mappings is not aimed at the people defining metrics, but at the people doing validations and evaluations, since they apply inexplicit language mappings, which need to be documented for others to enable them to compare and repeat these validations and evaluations.
- Just a few of the quality models proposed so far provide any description on how to combine metrics values to criteria and factors, thereby allowing us to infer, rank, and quantify the quality of a whole system, sub-system, class or other part, in one number. Some approaches are mentioned, but they are far from being exhaustively discussed.

An integrated approach including solutions to all these problems is needed.

Our proposal for overcoming these limitations, most of all ambiguity, is a description framework including a meta-model, language mappings, and language-independent metric definitions. The idea behind this proposal together with an overview of the current status will be discussed in Chapter 4. The core of our description framework is a meta-model, which is based on the Dagstuhl Middle Metamodel[4] [LTP04] as its initial meta-model. We will discuss our meta-model together with the description schema for the metrics, language mappings, and the definition of metrics in Chapter 5. In Chapter 6, we will discuss how we apply the extension method for meta-models that we proposed in [SLLL06] to add controlled extensibility and, thereby, flexibility. In Chapter 7, we discuss the implementation of our own tool support for evaluating and validating the described metrics and software quality model upon public information. We assume that other researchers can come to the same results when creating their tools based on our descriptions. The tool support also allows our experiments to be repeated, since it automates the data extraction and evaluation process. The evaluation of our efforts and tools will be discussed in Chapter 8. We publish our formalized metrics definitions in a compendium of software quality standards and metrics [LL05], which will be presented in Part II of this thesis. Additionally, we publish a software quality model based on ISO 9126-3 [ISO03] and the metrics defined in the compendium. We want to discuss it with the research community, allowing others to use the same definitions for conducting and repeating studies and experiments.

[4]We preferred to use the DMM, rather than defining our own meta-model, since this model is rather complete and already accepted.

Chapter 4

Compendium

Although there exist formal frameworks for metrics definitions, we concluded in Section 3.6 that many of the metrics published are described in an informal or semi-formal way, which creates ambiguity with respect to implementation, comparison, validation, and repetition. In many cases, the implementation is the only formal definition of the metrics. The few attempts that have the potential to present a well-defined set of metrics have published neither the complete specification of the metrics, nor the implementation. Examples include McQuillian et al. [MP06], who defined metrics using, e.g., the Object Constraint Language (OCL)[1] as the executable description language on top of a well-defined meta-model.

From scientific point of view it is "regrettable" that few authors are willing to publish their tools including source code, even though this would allow others to understand and compare the metrics definitions, to use the tool for conducting new experiments and to repeat existing experiments. From a business point of view, it is understandable. Therefore, we suggest that it should be enough to publish the specification of the metrics, meta-model, and language mapping in an open and well-defined form. This would allow everybody to implement or adapt their own tools in accordance with this shared specification. Thereby, it would be possible to achieve comparable and repeatable results from different research groups. The only threat to validity is programming errors in the implementation of the specification, not a specification that is ambiguous in itself.

4.1 Idea

Our idea and proposal for achieving a shareable and unambiguous specification of software metrics is a public *Compendium of Software Quality Standards and Metrics* [LL05]. This compendium is published and publicly available. It shall serve as a basis for creating a list of specifications of existing

[1] OCL is a formal language for describing expressions on UML models, `http://www.omg.org/technology/documents/formal/ocl.htm`

and new metrics, and their possible variants. The research community can refer to this compendium when discussing the current body of metrics, and base validation and evaluation experiments on it. Basing all these efforts on the same specification, the results will be directly comparable, and can be published together with the metrics. Thereby, the compendium will not only provide a well-defined specification of the metrics, but also serve as a guide for interpreting metric values, allowing us to find suitable metrics on a case to case basis, and allowing us to learn from similar situations.

Furthermore, this compendium shall describe a Software Quality Model, organizing the metrics into factors and criteria, allowing us to assess which metrics are suitable for evaluating a certain quality need, or, the other way around, what quality criteria a metric says something about. This Quality Model should be based on a standard in order to gain a high initial trust level.

To get started, we chose metrics that have been discussed, accepted, validated (to a certain degree) in case studies, and commented on, e.g., in the FAMOOS project [BBC+99]. The quality properties and metrics are linked to each other by a double index, allowing us to determine the relevance between the metrics and criteria from each point of view.

However, this compendium is meant to be a "living document" going beyond the normal experience sharing in conferences and workshops. We wish to create a compendium in the spirit of "A compendium of NP optimization problems" edited by Pierluigi Crescenzi and Viggo Kann [CKKW06]. The difference is that we propose a double index. The community is welcome to contribute with new metrics or comments, corrections, and add-ons to already defined ones. References to validating experiments or industrial experiences are especially appreciated.

The contributions proposed by the community in the form of web forms or emails will be edited, added to the compendium, and used to create new references in the double index, or to change/remove references shown to be invalid.

The requirements for such a compendium are:

Public – It needs to be publicly accessible for everybody to view and use.

Maintainable – It needs to be an extensible document. The public needs to contribute to it. The contributions need to be edited in order to maintain a high quality and uniform appearance.

Quality Model – It needs to contain a quality model.

Metrics – It needs to contain metrics, with a description scheme and references.

Mapping – It needs to contain a mapping between the metrics and the criteria of the quality model.

Index – It needs to provide an index for finding metrics related to quality factors and criteria, and vice versa.

4.2 Quality Model

As discussed in Section 3.6, many software quality models have been published. Some implicitly describe the direct relation between metrics and factors like maintainability or error-proneness. Others are more advanced and use a particular methodology for defining the relationship between metrics, quality attributes, and quality factors or goals. The best known methodologies are the Factor-Criteria-Metric approach of McCall [MRW77] and the Goal-Question-Metric approach of Basili and Rombach [BR88]. A number of other improved quality models have been developed, but the only internationally standardized software quality model is the ISO 9126, which has been discussed in Section 3.4. Since one of our goals is to be standard-based, thereby expecting a higher trust level than developing our own software quality model from scratch, we chose to base the quality model used in the compendium on this standard.

The ISO/IEC 9126 standard describes a quality model, which categorizes software quality into six characteristics that are divided into sub-characteristics. Figure 4.1 shows an adapted version of this quality model, which has been introduced in Section 3.4, Figure 3.2. The characteristics are manifested externally when the software is used as a consequence of internal software attributes. The internal software attributes are measured by means of internal metrics (e.g., monitoring of software development before delivery). Examples of internal metrics are given in ISO 9126-3. The quality characteristics are measured externally by means of external metrics (e.g., evaluation of software products to be delivered). Examples of external metrics are given in ISO 9126-2.

ISO 9126 has four parts, each being concerned with different aspects of the software quality life cycle, Figure 3.3 in Section 3.4. Our intention is to measure software quality using internal product attributes, and to draw conclusions from them to external quality. Therefore, we have selected ISO 9126-3:2003 as the basis for our software quality model.

We describe the factors and criteria of the quality model in the compendium and adapt them when necessary. We establish links between the criteria and the metrics and state which metrics are related to a specific

criterion, and, the other way round, which criteria are related to a certain metric. This double index is reflected in the compendium and distinguishes six levels of relevance:

not evaluated means that a connection between the metric and the quality criterion has not been investigated. Possible relations are not known.

no relation means that experiments and case studies show that there is no significant connection between the metrics and the quality criterion.

directly related means that experiments and case studies show that there is a significant relation between the metrics value and the quality criterion; direct means that high values for the metric are desired; low values are not desired.

highly directly related means that experiments and case studies show that there is a *strong* significant relation between the metrics value and the quality criterion; direct means that high values for the metric are desired.

inversely related means that experiments and case studies show that there is a significant relation between the metrics value and the quality criterion; inversely means that low values for the metric are desired, high values are not desired.

highly inversely related means that experiments and case studies show that there is a *strong* significant relation between the metrics value and the quality criterion; inversely means that low values for the metric are desired; high values are not desired.

The initial quality model describing the connection between the metrics and the quality factors and criteria is based on intuition and experience. It is intended as a starting point to be improved with an increasing number of experiments and reviews. We expect that it is possible to synthesize some general quality model by looking at different experiments and case studies. This quality model is then supposed to be tailored to specific projects. Experience from different projects and case studies are expected to provide categories of quality models, suiting better to the one or other project. Using the provided knowledge base shall help to find a good quality model as a starting point, easing its adaptation for project specific requirements.

The following three subsections provide an overview of the factors and criteria structuring the Software Quality Model, the metrics included, and the mapping between the criteria and metrics, defining the relationship between the metrics and criteria, and vice versa.

Criteria

The quality factors and criteria considered in the compendium are based on ISO 9126-3:2003, with the difference that usability is interpreted as reusability. This seems to us more appropriate considering internal quality from a developers point of view. Figure 4.1 provides a short overview of the quality model used in the compendium. It displays 6 quality factors and the related 29 quality criteria. For full details we refer to Part II.

Metrics

The software quality metrics used in the compendium so far are a collection of the most popular metrics discussed in literature. They are taken from different well know metrics suites like Chidamber and Kemerer [CK91, CK94], Li and Henry [LH93c, LHKS95], Bieman and Kang [BK95], and Hitz and Montazeri [HM95, HM96], and have already been discussed in Sections 3.1 and 3.2. The metrics are categorized into metrics assessing complexity, with the sub-categories size, interface complexity, and structural complexity, metrics assessing architecture and structure, with the sub-categories inheritance, coupling and cohesion, and metrics assessing design guidelines and coding conventions, with the sub-categories documentation and guidelines. So far, 23 metrics are described in the compendium.

Currently, our single size metric is Lines Of Code (LOC). Interface complexity metrics are Number of Attributes and Methods (SIZE2) and Number Of local Methods (NOM). Structural complexity metrics are McCabe Cyclomatic Complexity (CC), Weighted Method Count (WMC), and Response For a Class (RFC). Inheritance metrics are Depth of Inheritance Tree (DIT) and Number Of Children (NOC). Coupling metrics are Coupling Between Objects (CBO), Data Abstraction Coupling (DAC), Package Data Abstraction Coupling (PDAC), Message Passing Coupling (MPC), Change Dependency Between Classes (CDBC), Change Dependency Of Classes (CDOC), Locality of Data (LD), Coupling Factor (CF), Afferent Coupling (Ca), Efferent Coupling (Ce), and Instability (I). Cohesion metrics are Lack of Cohesion in Methods (LCOM), Improvement of LCOM (ILCOM), and Tight Class Cohesion (TCC). Currently, our single documentation metric is Lack Of Documentation (LOD). Guideline metrics are not yet defined. For exact metric definitions we refer to Part II.

Mapping

The mapping between the criteria of the software quality model and the metrics is described in three ways. First, associated with each quality criterion is

a list of related and highly related metrics. Second, associated with each metric is a set of related and highly related software quality properties (criteria). Third, an overview is provided in a matrix, listing the metrics and quality properties, showing their connection. This matrix is shown in Figure 4.2, for details on the relations between metrics and criteria, and vice versa, we refer to Part II.

The columns in Figure 4.2 represent the quality factors and criteria, the rows the metrics, grouped according to their particular category and subcategory. The colored cells mark the relationship between a metric and a criterion. In accordance with the six levels we distinguish, the options are:

not evaluated white (blank).

no relation gray (blank).

directly related + (plus sign).

highly directly related ++ (double plus sign).

inversely related − (minus sign).

highly inversely related −− (double minus sign).

Assessment of Quality Model

Our approach for assessing the Software Quality Model and unifying the different metrics values is to assign weights to the relations between factors and criteria and metrics and criteria. Each relationship receives a number specifying its weight. Currently, we associate the following weights with the mapping levels: *not evaluated/no relation* has weight 0, *directly/inversely related* has weight 1, and *highly directly/indirectly related* has weight 2.

Directly and inversely related have the same number, since the aggregation takes care of positive or negative effects of metrics.

The metric values are normalized over the system. They are mapped on the interval [0.0..1.0], by setting the maximum absolute metric value to 1.0 and the minimum absolute metric value to 0.0. All values in between are scaled linearly to [0.0..1.0], cf. Example 4.1.

Example 4.1. For instance, the Number of Children (NOC) metric is calculated for all classes C_s in the system, with $s = 1 \ldots m$ and m is the number of classes in the system. $NOC(c)$ is a metric that calculates the NOC value for the given class c, cf. Metric NOC in Section 12.2. The absolute value is stored in the attribute NOC^{abs} of each class C_s.

$$C_s.NOC^{abs} = NOC(C_s). \tag{4.1}$$

Further the absolute minimum min^{abs}_{NOC} and maximum max^{abs}_{NOC} values over all classes C_s are calculated. For each class C_s, the absolute value $C_s.NOC^{abs}$ is normalized and stored as $C_s.NOC$ as following:

$$C_s.NOC = \frac{C_s.NOC^{abs} - min^{abs}_{NOC}}{max^{abs}_{NOC} - min^{abs}_{NOC}}. \tag{4.2}$$

Further the absolute minimum and maximum values are normalized and stored in min_{NOC} and max_{NOC} respectively, to prepare the aggregation of metric values in different ranges and scales, cf. Section 1.4.

They are then aggregated within their scopes according to the defined outlier intervals. A scope can be the whole software system or parts of it, e.g., packages or subsystems. The aggregation can happen based on absolute or relative values. Having absolute values available might be an exception, since this requires already pretty specific knowledge about the software system under evaluation. Using relative values allows us to focus on outliers. The following examples show how to calculate the aggregated (cf. Example 4.2) and relative (cf. Example 4.3) values for a scope being *package*.[2]

Example 4.2. Continuing from Example 4.1, we assume the scope to be package, having an absolute outlier interval of < 0.1 and > 0.15 defined. P_i are all packages in the system, with $i = 1 \dots p$ and p is the number of packages in the system. C^j_i are all classes in a package i, with $j = 1 \dots c_i$ and c_i is the number of classes in P_i. $C^{<0.1}$ is the number of all classes C^j_i for which it holds that $C^j_i.NOC < 0.1$ and $C^{>0.15}$ is the number of all classes C^j_i for which it holds that $C^j_i.NOC > 0.15$. The outliers over all NOC values $C^j_i.NOC$ of classes C^j_i of all packages P_i are calculated as:

$$P_i.NOC^{abs} = \frac{C^{<0.1} + C^{>0.15}}{c_i}. \tag{4.3}$$

Example 4.3. Contrary to Example 4.2, we now use a relative confidence interval with $< 15\%$ and $> 85\%$ defined. For the enclosing scope (package) the minimum min^j_{NOC} and maximum max^j_{NOC} values over all classes C^j_i are calculated. Therefore, $C^{<15\%}$ is the number of all classes C^j_i for which holds that $C^j_i.NOC$ is within the lower 15% of all normalized NOC values in the scope (< 0.15, since normalized). That is:

$$C^{<15\%} = \left| \{ C^j_i | C^j_i.NOC < 0.15 \} \right|.$$

[2]Package as in the context of the Java programming language, where classes are organized in packages.

$C^{>85\%}$ is the number of all classes C_i^j for which holds that $C_i^j.NOC$ is within the upper 15% of all normalized NOM values in the scope (> 0.85, since normalized). That is:

$$C^{>85\%} = \left|\{C_i^j | C_i^j.NOC > 0.85\}\right|.$$

The outliers over all NOC values $C_i^j.NOC$ of classes C_i^j of all packages P_i are calculated as:

$$P_i.NOC^{rel} = \frac{C^{<15\%} + C^{>85\%}}{c_i}. \tag{4.4}$$

This works similarly for other metrics or other scopes, e.g., methods, classes, packages, or the whole system.

A similar procedure is used for the characteristics and sub-characteristics. We calculate a weighted aggregation. We consider high values as bad if they are in the upper 10% ($> 90\%$), and we consider low values as bad if they are in the lower 10% ($<10\%$). *$> 90\%$* and *$< 10\%$* corresponds to the $-$, $--$, and *$> 10\%$* and *$< 90\%$* corresponds to the $+$, $++$. A single $+$ or $-$ relates to a weight of 1, a double $+$ or $-$ ($++/--$) relates to a weight of 2.

Example 4.4. The sub-characteristic *Analyzability* is influenced by two metrics, *NOC* and *LOC*. *LOC* has a higher influence than NOC, therefore it is weighted with 2 ($--$), and *NOC* is weighted with 1 ($-$), cf. Figure 4.2. We calculate the Analyzability of all classes C_i^j in all packages P_i as:

$$P_i.Analyzability = \frac{2 * P_i.LOC + 1 * P_i.NOC}{2 + 1}. \tag{4.5}$$

Getting a value close to 0 means good analyzability of a package, since few classes have neither too high nor too low values for *LOC* and *NOC*, which brings them outside the outlier interval. Getting a value close to 1 means bad analyzability of a package, since many classes have too high or too low values for *LOC* and *NOC*, which brings them inside the outlier interval. Following this principle, sub-characteristics can be aggregated in the same way as characteristics.

4.3 How to use the Compendium

Currently, we see two main uses of the compendium, depending on the intentions of the users. The first group of users consists mainly of practitioners who want to inform themselves about the state-of-the-art in metrics, and who want to find a suitable set of metrics quantifying their quality factors

and criteria. The second group of users consists mainly of researchers who want to validate and evaluate the metrics. They also consult in the compendium, but they report back their findings and propose corrections to the compendium, extending, correcting and restructuring it.

I want to consult the Compendium

The compendium has two main indices "Software Quality ISO Standards" and "Software Quality Metrics", cf. Part II. The first index lists quality characteristics (factors) and sub-characteristics (criteria). Each characteristics and sub-characteristics is first defined and then related to a number of metrics allowing its assessment. The second index lists quality metrics. Each metric is defined and then related to a number of quality characteristics and sub-characteristics for which it supports the assessment.

I want to contribute to the Compendium

All contributions are edited before publication. Please submit contributions to `rudiger.lincke@arisa.se`.

If you want to describe a new metric, please use the description schema for metrics, cf. Section 5.1. If you want to comment on, correct, or add to an already defined metric, please refer to the metric by its handle and indicate the items in the description schema for metrics your contribution applies to, cf. Section 5.1. All other kinds of contributions are welcome as well.

4.4 Overview and Current Status

The compendium consists of two parts: the first defines the Software Quality ISO Standards, and the second Software Quality Metrics. The goal of the compendium is to provide an information resource precisely defining our interpretation of the software quality standards and the software metrics and their variants. Moreover, we propose connections among them. These connections are the hypotheses we want to validate in our long term research, as discussed in Section 1.2.

Our *Compendium of Software Quality Standards and Metrics* has been published in its first version in August 2005, cf. [LL05]. It is available online as HTML pages,[3] and as a printer friendly version for download.[4] At the time of publishing it contained:

[3] `http://www.arisa.se/compendium`

[4] `http://www.arisa.se/compendium/quality-metrics-compendium.pdf` for the compendium `http://www.arisa.se/compendium/quality-metrics-matrix.pdf` gives an overview of the relations between quality properties and metrics

- 37 software quality properties (factors, criteria), and
- 8 software quality metrics.

The current version, as of December 2006, which will be published online with the successful defense of this thesis, will contain:

- 37 software quality properties (factors, criteria),
- 23 software quality metrics.

The 37 quality properties are taken from the ISO 9126-3:2003 standard. For a description, we refer to the ISO standard [ISO03]. The 23 software quality metrics are taken from different well know metrics suites like Chidamber and Kemerer [CK91, CK94], Li and Henry [LH93c, LHKS95], Bieman and Kang [BK95], or Hitz and Montazeri [HM95, HM96] and contain among others Weighted Method Count (WMC), Tight Class Cohesion (TCC), Lack of Cohesion in Methods (LCOM), McCabe Cyclomatic Complexity (CC), Lines Of Code (LOC), Number Of Children (NOC), Depth of Inheritance Tree (DIT), Data Abstraction Coupling (DAC), and Change Dependency Between Classes (CDBC), etc. See Section 4.2 for a complete overview, and Part II for details. In addition to these metrics, we include a few of our own metrics. The non-object-oriented metrics are mainly size and complexity metrics. The object-oriented metrics focus on cohesion, coupling, and inheritance structure.

4.5 Summary

In this chapter, we discussed our idea for defining metrics using a description framework and their publication in a so called *Compendium of Software Quality Standards and Metrics*. In this compendium, we define a standard software quality model relating metrics to quality criteria and vice versa. We invite researchers and practitioners to contribute to this compendium.

Despite providing an overview of the compendium, its quality model, the metrics involved, and how to use the compendium, we still have two open points to discuss:

1. We need to describe exactly how to define metrics in an unambiguous way.
2. We need to define how we want to maintain the compendium and the metrics definitions.

These two questions will be answered in the following two chapters.

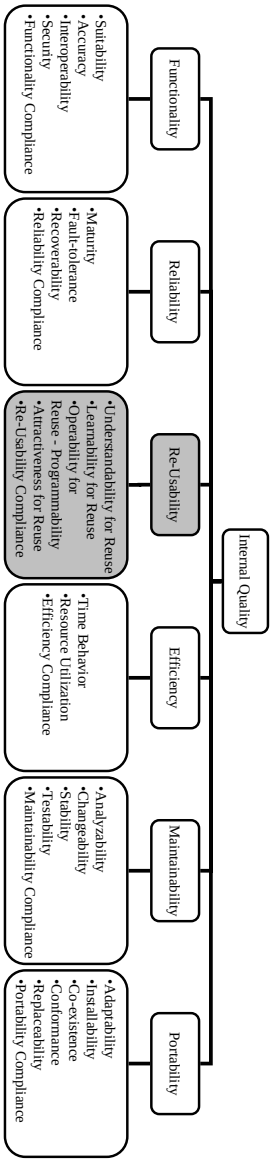

Figure 4.1: Quality factors and criteria of compendium quality model. Modified ISO 9126-3 Software Quality Model. Usability factor (cf. Figure 4.1) is now Re-Usability factor (gray). The same holds for the criteria.

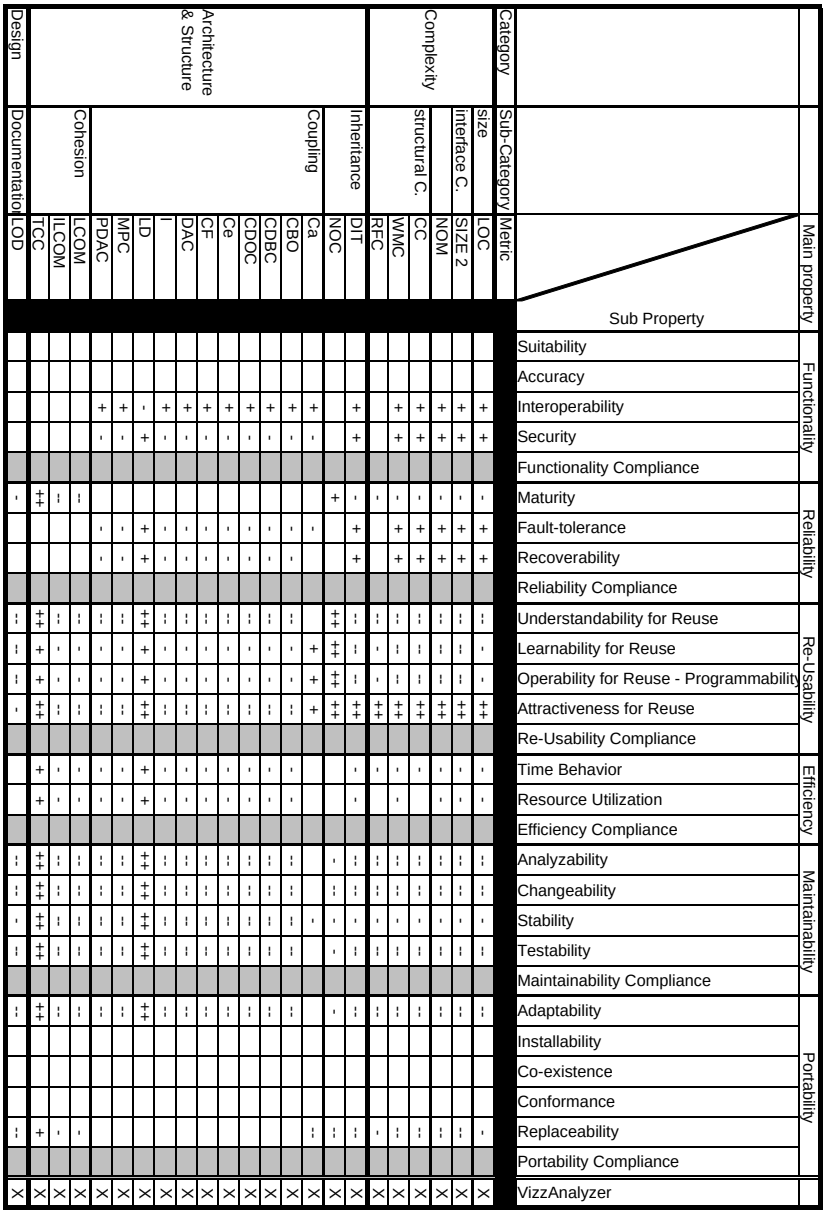

Figure 4.2: Software Quality Matrix, showing quality factors, criteria and metrics and their relationship.

Chapter 5

Metrics Definitions

We still need to describe how to define metrics exactly. We concluded in Chapter 3 that existing approaches are limited, since they are either not formally defined, or not completely available to the public. Even more importantly, a sound meta-model upon which the metrics can be formally defined is required, and language mappings need to be provided in order to make the meta-model itself unambiguous and language-independent.

We are going to eliminate limitations of existing approaches by defining the description schema used in the compendium for metrics in Section 5.1, and by defining our meta-model and language mappings in Section 5.2.

5.1 Description Schema

To give our compendium an intuitive structure, and to ease the classification, identification and description of metrics, we provide a *description schema* for the metrics described in the compendium. Metrics are classified as supporting mainly statements on software *Complexity*, software *Architecture and Structure*, and software *Design and Coding*. See Section 4.2 for an overview, and Part II for details. Each metric is described using the schema described below to facilitate the identification of metrics based on their classification, name, handle, definition, or other properties, and to facilitate the search for related quality criteria or related literature.

We use the following description schema for each metric. It defines:

Handle A short name or symbol uniquely identifying the metric. This is the way it is referenced in this document and literature. If there are more than one variation of a metric, additional symbols will be added to distinguish the different variants.

Description A textual description of the metric and what it measures. This description is a summary of how the metric is described in literature. Since it is described in natural language, it is not suitable for an unambiguous definition of the metric.

Scope Identifies the program element the metric applies to. This could be *system* for the whole software system, *package/directory* for entities of a package or a directory, *class/module/file* for classes, modules and files, respectively, and *procedure/function/method* for procedures, functions and methods, respectively. Only the core scope is named.

Definition A formal definition of the metric using set theory and mathematics. The definition is based on the meta-model specified in the compendium. Usually a view is defined as well, see below.

View Definition of the *view*, describing its relevant elements, semantic relations, and the tree grammar.

Scale Defines the scale type of the metric. It is one of *Absolute*, *Rational*, *Interval*, *Ordinal*, and *Nominal*.

Domain Defines the domain of metric values.

Highly Related Software Quality Properties Discusses factors and criteria as listed in the ISO 9126-3:2003 standard that are assessable with the metric. See Section 4.2 for an overview, and Part II for details.

Related Software Quality Properties Discusses factors and criteria as listed in the ISO 9126-3:2003 standard that are somewhat assessable with the metric. See Section 4.2 for an overview, and Part II for details.

References Gives references to literature, projects, repositories, and programs using, implementing, or describing the metric. Moreover, it points to validating and evaluating experiments investigating its appropriateness.

Since Since which version of the compendium is the metric contained.

The definition of the metrics is based on the common meta-model as defined in Section 5.2. For each metric, we define a *view* further abstracting from the common meta-model to provide exactly the information required by that metric. The view is used for the actual metric definition. This approach makes the metric definitions independent from changes of the common meta-model (this is discussed in more detail in Chapter 6). Formally, views of a metric analysis A are defined as pairs $V^A = \left(G^A, R^A\right)$ and are bound to the common model with a mapping specification α^A. Again, G^A is a tree grammar specifying the set of view entities and their structural containment required by A. R^A is a set of semantic relations over view entities required by A.

The construction of concrete view models follows the same principles as the abstractions from front-end specific to common models: we ignore some common meta-model entity types, which leads to a filtering of the corresponding nodes. We propagate relevant descendants of filtered nodes to their relevant ancestors by adding them as direct children. Moreover, we ignore some relation types, and attach remaining relations defined over filtered nodes to the relevant ancestors of those nodes, as described in detail in [SLLL06].

To simplify metric definitions, we define some utility operation(s):

$succ(e, r)$ $[succ^*(e, r), succ^+(e, r)]$ – denotes the set of direct [transitive, transitive excluding e] successors of a node or set of nodes e over edges (relations) of type r.

$|S|$ – denotes the number of elements in a set S.

$pred(e, r)$ $[pred^*(e, r), pred^+(e, r)]$ – denotes the set of direct [transitive, transitive excluding e] predecessors of a node or set of nodes e over edges (relations) of type r.

$dist(e, e', r)$ – denotes the distance or set of distances (number of relations) between a node e and a node or set of nodes e'. It is 0, if it e equals e', and ∞, if there is no path between e and e'.

$max(S)$ – denotes the maximum in a set S of values.

$min(S)$ – denotes the minimum in a set S of values.

$scc(e, r)$ – denotes the set of strongly connected components in a set e of elements over edges (relations) of type r.

$outdegree(e, r)$ $[outdegree^+(e, r)]$ – denotes the number of outgoing relations of type r from a node e including [excluding] relations to e itself.

For additional simplification, we define some frequently used sets:

$M(c) = pred(c, \text{methodof})$ – denotes the set of all methods locally declared in a class c, ignoring visibility and other attributes.

$F(c) = pred(c, \text{fieldof})$ – denotes the set of all fields locally declared in a class c, ignoring visibility and other attributes.

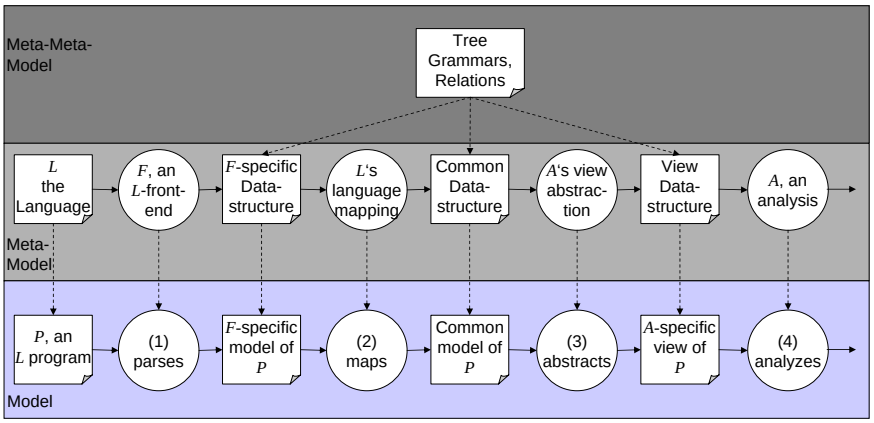

Figure 5.1: Information extraction, mapping to common representation, view abstraction, and analysis – displayed on model- and meta-model-level. Based on [SLLL06].

5.2 Meta-Model – Theory

This Section discusses the theory of the meta-model used for defining the software quality metrics. The practice is discussed in Section 5.3.

Our architecture for constructing, capturing, and accessing a model of a software system consists of four major components, cf. circles in Figure 5.1:

1. Different concrete information-extracting front-ends for programming languages or other program representations. They capture information about a program in a *front-end specific model*.[1]

2. Converters mapping this front-end specific to a language-independent *common model* capturing program information relevant for later analysis.[2,3]

3. Abstractions computing *views* on the common model specific for a subset of analyses.

[1] We are aware of the fact that if a front-end deliberately does not provide elements, which are defined in the front-end specific meta-model, it is not correctly implemented.

[2] We are aware of the fact that if a language mapping is well-defined but not compatible with the common meta-model, metrics will be front-end dependent.

[3] We are aware of the fact that if a language mapping deliberately leaves out elements which are contained in a front-end specific meta-model, it is not correctly implemented.

4. Different, concrete *analyses* accessing their respective views.[4]

A number of *front-end specific model*s relate to one *common model*, which, in turn, can have a number of different *views*. Each view may be accessed by a number of *analyses*.

In what follows, we clearly distinguish the *model* from the *meta-model* level. The former captures more or less abstract information of a concrete program. In that sense, it contains *model*s of that program. The latter describes all *possible* models of programs. It can be understood as the type of models or data structure capturing them. We distinguish front-end specific, common, and view models. When we define mappings between models at different levels of abstraction, we do that on the respective meta-model-level.

Finally, there is a common formalism that we will use for defining the front-end specific, the common, and the view meta-models. This common formalism, i.e., the meta-meta-model, will be *tree grammars* and *relational algebra*. See Figure 5.1 for an overview.

Front-end Specific Meta-Model

Each *front-end* understands a specific program representation, e.g., a programming or specification language. It provides a front-end specific model of that program representation. In general, such a model consists of entities representing concrete program entities and relations between them representing syntactic and semantic program relations.

We cannot influence the front-end F specific representations, i.e., meta-models $\mathcal{M}^F$, but we may safely assume that they can be described in the following way: a front-end F specific meta-model is denoted by

$$\mathcal{M}^F = (\mathcal{G}^F, \mathcal{R}^F),$$

where $\mathcal{G}^F$ is a tree grammar specifying the set of model entities and their structural containment. $\mathcal{R}^F$ is a set of semantic relations over model entities. Formally,

$$\mathcal{G}^F = (T^F, P^F, prog^F)$$

with T^F the set of model entities (node types), P^F a set of BNF-productions defining structural containment tree structures, and $prog^F \in T^F$ the root type of the structural containment trees. BNF-productions $p \in P^F$ have the form:

$$t ::= expr$$

[4] We are aware of the fact that if a metric requires elements or relations which cannot be provided by a particular front-end, the front-end is not suitable for this metric.

where $t \in T^F$, and $expr$ is an expression over $T \subseteq T^F$. Expression are either sequences $(t_1 \dots t_k)$, iterations (t^*), or alternatives $(t_1| \dots |t_k)$ with the obvious semantics.

$\mathcal{R}^F$ denotes a set of semantic relations over model entities:

$$\mathcal{R}^F = \{R_1^F, \dots, R_n^F\}$$

and each $R_i^F, 1 \leq i \leq n$ is defined over subsets of entities T^F, i.e., $T_1 \times \dots \times T_k$, $T_j \subseteq T^F, 1 \leq j \leq k$.

Example 5.1. Let $\mathcal{M}^{\mathtt{J}} = (\mathcal{G}^{\mathtt{J}}, \mathcal{R}^{\mathtt{J}})$ be the meta-model of a `Java` front-end `J`, which we will use as a running example. $\mathcal{G}^{\mathtt{J}} = (T^{\mathtt{J}}, P^{\mathtt{J}}, Program^{\mathtt{J}})$ defines `Java`'s Abstract Syntax Trees (ASTs). It contains, among others, entities (node types) for class $ClassD^{\mathtt{J}}$, interface $IntD^{\mathtt{J}}$, method declarations $MethodD^{\mathtt{J}}$, and call expressions $CallExpr^{\mathtt{J}}$. Productions $P^{\mathtt{J}}$ define the structural containment in ASTs. $\mathcal{R}^{\mathtt{J}}$ contains "extends" relations on classes and interfaces: $ext_c : ClassD^{\mathtt{J}} \times ClassD^{\mathtt{J}}$ and $ext_i : IntD^{\mathtt{J}} \times IntD^{\mathtt{J}}$. Also, $\mathcal{R}^{\mathtt{J}}$ contains "implements" relations $imp : ClassD^{\mathtt{J}} \times IntD^{\mathtt{J}}$ and "call" relations $call^{\mathtt{J}} : CallExp^{\mathtt{J}} \times MethodD^{\mathtt{J}}$.

Common Meta-Model

The *common meta-model* $\mathcal{M}$ abstracts from front-end specific details. As argued before, it is not static but evolves with the introduction of new analyses, and front-ends, cf. Section 6.2. However, at any point in time, it can be defined as a pair of a tree grammar for entities and their structural containment, and a set of semantic relations. Hence, we may use the same describing formalism, i.e., meta-meta-model, for defining the common meta-model at any point in its evolution. We denote the common meta-model by

$$\mathcal{M} = (\mathcal{G}, \mathcal{R}),$$

i.e., we skip the index for the specific front-end. Apart from that, a common meta-model and a frontend-specific meta-model look exactly the same.

Example 5.2. Let $\mathcal{M} = (\mathcal{G}, \mathcal{R})$ be our common meta-model at a certain point in evolution and let $\mathcal{G} = (T, P, Program)$. Assume that T only defines node types for the whole program, and type and method declarations, i.e., $Program, TypeD, MethodD \in T$. The containment structure is defined by P:

$$\begin{aligned} Program &::= TypeD^* \\ TypeD &::= MethodD^* \end{aligned}$$

$\mathcal{R}$ defines an inheritance relation $inh : TypeD \times TypeD$ and a call relation $call : MethodD \times MethodD$.

As mentioned before, the common meta-model is an abstraction of several front-end specific meta-models. For each front-end F, this abstraction is called the *front-end mapping* α^F. It is defined by mapping front-end specific grammars $\mathcal{G}^F$ to the common meta-model grammar $\mathcal{G}$ and front-end specific relations $\mathcal{R}^F$ to the common meta-model relations $\mathcal{R}$.

For the grammars, the front-end mapping α^F is defined by mapping front-end specific to common model entities:

$$\alpha^F : T^F \to T.$$

The front-end specific program node types $prog^F$ are always mapped to the common program node type $prog$, i.e., $\alpha^F(prog^F) = prog$. For the front-end mapping of relations, we define the mapping of individual relations:

$$\alpha^F : \mathcal{R}^F \to \mathcal{R}.$$

In general, we require α^F to be neither:

- *surjective*, i.e., some common meta-model types and relations do not correspond to front-end specific meta-models types and relations, nor
- *total*, i.e., some language-specific meta-model types and relations may be ignored.

Surjectiveness would imply that every front-end must at least provide the information which the common meta-model is able to capture. This is unnecessarily restrictive. If an analysis needs information that a particular front-end cannot provide (but others can), this front-end is not applicable (while with some others, the analysis works fine).

Completeness would imply that there is basically no abstraction from the front-end specific to the common meta-model (just renaming of types and relations). This would lead to unnecessary efforts in plugging in very rich and detailed front-ends, even if this detailed information is never used in analysis.

Example 5.3. Our front-end mapping α^{J} maps node types and relations of $\mathcal{M}^{\mathrm{J}}$ sketched in Example 5.1 to the common meta-model $\mathcal{M}$ sketched in Example 5.2. For the node types:

$$\begin{aligned} \alpha^{\mathrm{J}}(Program^{\mathrm{J}}) &= Program \\ \alpha^{\mathrm{J}}(ClassD^{\mathrm{J}}) &= TypeD \\ \alpha^{\mathrm{J}}(IntD^{\mathrm{J}}) &= TypeD \\ \alpha^{\mathrm{J}}(MethodD^{\mathrm{J}}) &= MethodD \end{aligned}$$

And for the relations:

$$\begin{aligned} \alpha^{\mathrm{J}}(ext_c^{\mathrm{J}}) &= inh \\ \alpha^{\mathrm{J}}(ext_i^{\mathrm{J}}) &= inh \\ \alpha^{\mathrm{J}}(imp^{\mathrm{J}}) &= inh \\ \alpha^{\mathrm{J}}(call^{\mathrm{J}}) &= call \end{aligned}$$

For other types and relations of $\mathcal{M}^{\mathrm{J}}$, a front-end mapping is not defined.

The front-end mapping is specified on the meta-model level and implies a mapping for concrete model instances in the following way: first, the front-end specific structural containment tree is mapped to the corresponding common structure. Then the mapped semantic relations are attached to the common structure.

The mapping of the containment trees is defined recursively: starting with the root, we traverse the front-end specific containment tree in depth-first-search order. We create new common model nodes of types with a mapping defined in α^F – we call those nodes *relevant*. The other, *irrelevant* nodes are ignored.

Algorithm 1 *generateTreeEvents*($n =< id, t^F >$)

```
call startNode(n)
for all c_i, ∈ childrenOf(n) do
  call generateTreeEvents(c_i)
end for
call finishNode(n)
```

Algorithm 2 *startNode*($n =< id, t^F >$)

```
if α^F(t^F) is defined then
  create new node n' :=< newId, α^F(t^F) >
  map(n) := n'   // define n's mapping
  relAncTmp := n   // set n to the current relevant ancestor
  childrenStack.top.append(n')   // top contains current childrenList
  childrenStack.push(create new childrenList)
end if
relAnc(n) := relAncTmp   // define n's relevant ancestor
```

A generic event-based interface between front-end specific and common meta-models and an abstract algorithm for mapping the actual model instances is given in Algorithms 1– 3: A *tree-walker*, cf. Algorithm 1, initially

Algorithm 3 *finishNode*($n =< id, t^F >$)

```
if α^F(t) is defined then
  if children := childrenStack.pop() is not empty then
    add children to map(n)   // map(n) defined in Program 2
  end if
  relAncTmp := n   // reset the current relevant ancestor
end if
```

called with the root node of the front-end specific model, traverses the containment tree in depth-first-search order and generates *startNode*-events on traversal downwards, and *finishNode*-events on traversal upwards, respectively. Nodes of the structural containment tree are pairs $< id, t >$ with id and $t \in T$ the nodes's key identifier and type, respectively.

The common model data structure is created by the corresponding event-listener, *startNode*, cf. Algorithm 2, and *finishNode*, cf. Algorithm 3. They preserve the tree structure, but filter out irrelevant nodes.

A front-end specific relation is a set of tuples $R^F(n_1, \ldots, n_k)$ over containment tree nodes. For constructing the common model, we ignore the relations that are not mapped by α^F; we only consider the relations for which such a mapping is defined. Let $R^F : T_1^F \times \ldots \times T_k^F \in \mathcal{R}^F$ be such a relation with front-end mapping $\alpha^F(R^F) = R$. Assume that each type in each T_i^F was mapped by α^F as well. Then each node in $R^F(n_1, \ldots, n_k)$ would have a correspondence in the common model; R could simply be defined over those nodes. However, if α^F was not defined for a type of a node n_i in $R^F(n_1, \ldots, n_k)$, we would "attach" the relation to n_i's closest relevant ancestor. That is the closest transitive parent of n_i, denoted by $relAnc(n_i)$, which is relevant. For relevant nodes n, by definition $relAnc(n) = n$ holds.

Algorithm 4 *generateRelationEvents*($\mathcal{R}^F$)

```
for all R_i^F ∈ R^F do
  for all (n_1, ..., n_k) ∈ R_i^F do
    call newRelationTuple(R^F(n_1, ..., n_k))
  end for
end for
```

The mapping of a front-end specific relation to a common relation is done in a second phase using the event generator *generateRelationEvents*, cf. Algorithm 4, and the corresponding event listener *newRelationTuple*, cf. Algorithm 5.

Note that the *abstract event generation* (algorithm schema) and the event

Algorithm 5 *newRelationTuple*$(R^F(< id_1, t_1^F >, \ldots, < id_k, t_k^F >))$

if $\alpha^F(R^F)$ is defined **then**
  for all $n_i =< id_i, t_i^F > 1 \leq i \leq k$ **do**
    $r_i :=$ map$(relAnc(n_i))$ // map and *relAnc* defined in Program 2
  end for
  add tuple $(r_1, \ldots, r_k)$ to relation $\alpha^F(R^F)$
end if

handlers work independently of different concrete front-ends, languages, the front-end mappings, and the current common meta-model. The abstract event generation and the event handling do not change when any of these components changes. However, a *concrete implementation* of the abstract event generation side, i.e., the implementation of Algorithms 1 and 4, is front-end specific and must obey the front-end specific meta-model APIs.

View Meta-Models and Analysis

The common model is the data repository for program analysis. Analyses might directly traverse that model, extract required information, and perform computations. However, we introduce analysis-specific *views* on the common model, further abstracting from the common model and providing exactly the information required by that analysis. Several analyses could share a view; hence, a view factoring out this information is useful to have.

Views are further abstractions of the common model. Formally, a meta-model specific view for an analysis A is described as

$$\mathcal{V}^A = (\mathcal{G}^A, \mathcal{R}^A).$$

$\mathcal{G}^A$ is a tree grammar specifying the set of view entities and their structural containment required by A. $\mathcal{R}^A$ is a set of semantic relations over view entities required by A. Again, we use the same description framework for defining a view meta-model as before: the front-end specific, the common, and the view meta-model are all defined with the same meta-meta-models: tree grammar and relational algebra.

View model construction follows the same principles as the abstractions from front-end specific to common models: we ignore some entity types, which leads to filtering of the corresponding nodes. We propagate relevant descendants of filtered nodes to their relevant ancestors by adding them as direct children. Moreover, we ignore some relation types and attach remaining relations defined over filtered nodes to the relevant ancestors of those nodes.

As in our mapping from front-end specific to common models, construction of a view is defined using a mapping specification α^A. In contrast to

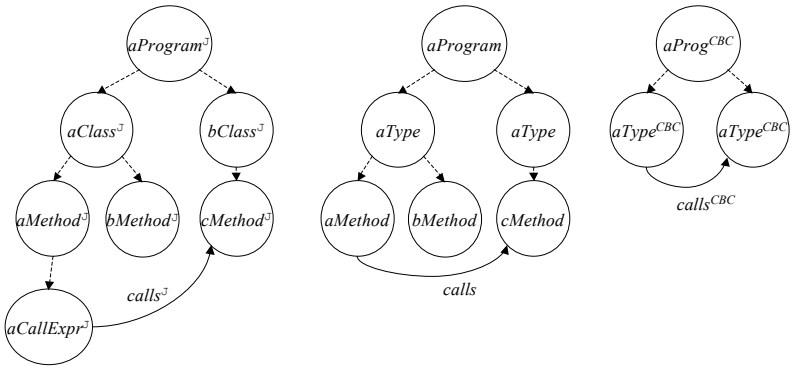

Figure 5.2: A front-end specific, a common, and a view model corresponding to examples 5.1–5.4. Taken from [SLLL06].

the mapping from front-end specific to common models, α^A is not front-end specific, but specific for a set of analyses A. However, the same definitions for the actual model to view mapping apply. Even for computing a specific view on a common model instance, we reuse the event-based architecture (and implementation). Handlers for tree- and relation-events are responsible for constructing the view's tree-structure and for adding relevant relations, respectively. The event-generator consists of the same phases as before: tree walker and relation generator. Since the event source, i.e., the common model data-structure, is part of our system (as opposed to the front-end specific model data-structure), we can even define the event-generator side *implementation* (as opposed to the only abstract algorithm schema generating events for the front-end specific models).

Finally, analysis accesses the corresponding view and performs computations. We deliberately skip a discussion on how to store and display analysis results and refer to [LP05a] instead.

Example 5.4. Let CBC be a metric analysis computing the relative coupling of a class via method calls:

$$CBC(c) :=| \, calls(c, c') \mid / \mid calls(c, \_) \mid \;\; where \; c \neq c'.$$

An appropriate view meta-model $\mathcal{V}^{CBC}$ would define the entity types for type declarations $TypeD^{CBC}$ and a call relation $call^{CBC}$. As opposed to the call relation of the common meta-model, cf. Example 5.2, this $call^{CBC}$ is defined over class declaration nodes $call^{CBC} : TypeD^{CBC} \times TypeD^{CBC}$.

Our mapping α^{CBC} defines a view meta-model for the common meta-model $\mathcal{M}$ sketched in Example 5.2, i.e., for the node types

$$\begin{aligned} \alpha^{CBC}(Program) &= Prog^{CBC} \\ \alpha^{CBC}(TypeD) &= TypeD^{CBC} \end{aligned}$$

and for the relations

$$\alpha^{CBC}(calls) = calls^{CBC}.$$

Hence, nodes of the method declaration type and inheritance relations are filtered. Figure 5.2 shows an example mapping from a front-end specific to a common model, and, further, to a CBC view model.

5.3 Meta-Model – Practice

This Section discusses the practice of the meta-model used for defining the software quality metrics. The theory was discussed in Section 5.2.

Dagstuhl Middle Meta-Model

We use our notation of tree-grammars and relations to describe the Dagstuhl Middle Metamodel version 0.007. Further, we sketch the meta-models for two front-ends and their mapping to the DMM. Additionally, we describe views on the meta-model that we will need for three (example) metric definitions.

Originally, the Dagstuhl Middle Metamodel 0.007 (DMM) was defined in a UML notation, cf. [LTP04]. We use our model description $M = (G, R)$ instead, since this is closer to the notation used for programming language specifications. This simplifies the definition of the language binding later on.

The root node type of G is *Model*. The productions P of G describe a structural containment relation, denoted by ::= in the productions, and a specialization hierarchy on the meta-model entities T, denoted by $\succ$. The DMM is shown in Table 5.1, 5.2, and 5.3. The special formatting of some text elements in the tables and paragraphs can be ignored for now. Its meaning will be explained in the next subsection. The structural containment relation is added to the original DMM. In contrast to the DMM, we separate abstract and concrete meta-model elements: abstract model elements, denoted by "A<*entity name*>", are never instantiated in a concrete model.

Attributes of DMM entities (unary relations) and binary relations are defined in R as unary and binary relations respectively. Unary relations include: isSubclassable(Class), size(CollectionType), name(MacroArgument |

Model	::=	ASourceObject* AModelObject*
ASourceObject	≻	ASourcePart | ASourceUnit
ASourcePart	≻	SourcePart | MacroExpansion | MacroArgument | ADefinition | AResolvable | Comment
ADefinition	≻	MacroDefinition | Definition
AResolvable	≻	Declaration | Reference | Resolvable
ASourceUnit	≻	SourceFile | SourceUnit
AModelObject	≻	ModelObject | AModelElement
AModelElement	≻	ModelElement | Package | AStructuralElement | ABehaviouralElement | CompilationUnit
AStructuralElement	≻	AType | AValue
AType	≻	Type | AStructuredType | CollectionType | EnumeratedType
AStructuredType	≻	StructuredType | Class
Class	::=	Field* Method* Initializer*
Class	≻	AbstractClass | AnonymousClass
AValue	≻	Value | EnumerationLiteral | AVariable
AVariable	≻	Variable | Field | FormalParameter
EnumerationType	::=	EnumerationLiteral*
ABehaviouralElement	≻	ExecutableValue | Method | Routine | Constructor
Method	::=	FormalParameter*
ExecutableValue	::=	FormalParameter*
Routine	::=	FormalParameter*

Table 5.1: Dagstuhl Middle Metamodel, productions P and specialization hierarchy

MacroDefinition | ModelObject | AResolvable | ASourceUnit), visibility(Field | Method | ModelElement), path(SourceFile), position(FormalParameter), isConstructor(Method), isDestructor(Method), isAbstract(Method), isDynamicallyBound(Method), isOverrideable(Method), visibility(Field | Method | ModelElement), path(SourceFile), startLine(SourcePart), start(SourcePart), endLine(SourcePart), and endChar(SourcePart). Binary relations and their specialization *prec* are listed in Tables 5.2 and 5.3, respectively.

Common Meta-Model 1.0

We base our Common Meta-Model 1.0 (CMM10) on the Dagstuhl Middle Meta-Model 0.007 (DMM). We describe it using a notation of tree-grammars and relations. (See the preceding subsection, where we display the full DMM in Tables 5.1, 5.2, and 5.3.)

Since the DMM provides more elements and relations than is currently needed by our Java and UML front-ends, we removed unnecessary elements and relations, in order to increase clarity. These are marked with wavy underline text (text) in Tables 5.1, 5.2, and 5.3. Additional elements and relationships needed to contain information provided by the front-ends have been added. They are marked with underlined text (text). Details about the extension of the original meta-model, necessary to resemble CMM10, are

Accesses	:	ABehaviouralElement × AStructuralElement
Contains	:	SourceObject × SourcePart, Package × ModelElement
Declares	:	SourceObject × ModelObject
Defines	:	SourceObject × ModelObject
Describes	:	SourceObject × Comment
HasValue	:	Variable × Value
Imports	:	Class × Package
Includes	:	SourceFile × SourceFile
InheritsFrom	:	Class × Class
Invokes	:	ABehaviouralElement × ABehaviouralElement
IsActualParameterOf	:	ModelElement × Invokes
IsConstructorOf	:	Class × Constructor
IsDefinedInTermsOf	:	Type × Type
IsEnumerationLiteralOf	:	EnumerationLiteral × EnumeratedType
IsExpansionOf	:	MacroDefinition × MacroExpansion
IsFieldOf	:	Field × StructuredType
IsMethodOf	:	Method × Class
IsOfType	:	Value × Type
IsParameterOf	:	FormalParameter × BehaviouralElement
IsReturnTypeOf	:	Type × BehavioralElement
IsSubpackageOf	:	Package × Package

Table 5.2: Dagstuhl Middle Metamodel, binary semantic relations in R

contains	$\succ$	containsField, containsMethod, containsConstructor
Relationship	$\succ$	ModelRelationship, SourceRelationship, SourceModelRelationship
ModelRelationship	$\succ$	InheritsFrom, IsPartOf, Invokes, IsOfType, Accessess, IsDefinedInTermsOf, IsPartOfSignatureOf, IsActualParameterOf, HasValue, IsSubpackageOf
InheritsFrom	$\succ$	Extends, Implements
Invokes	$\succ$	InvokesConstructor, InvokesSuperConstructor, InvokesSuper
IsPartOf	$\succ$	IsEnumerationLiteralOf, IsMethodOf, IsFieldOf, IsConstructorOf
IsPartOfSignatureOf	$\succ$	IsParameterOf, IsReturnTypeOf

Table 5.3: Dagstuhl Middle Metamodel, specialization hierarchy $\prec$ of semantic relations

discussed in Section 6.3, cf. Tables 6.4 and 6.5.

UML Language Mapping

This subsection describes a language-specific (simple) UML class diagram meta-model $M^{\mathtt{U}} = (G^{\mathtt{U}}, R^{\mathtt{U}})$ and its language binding. The model entities $T^{\mathtt{U}}$ and the productions $P^{\mathtt{U}}$ describing the containment structure over the model entities are defined in Table 5.4. The root node type is model$^{\mathtt{U}}$. There is no particular specialization hierarchy $\prec^{\mathtt{U}}$ defined over these types. The (binary) semantic relations $R^{\mathtt{U}}$ are shown in Table 5.6. The specialization hierarchy is defined in Table 5.5.

The mapping functions for mapping the front-end model specific types $T^{\mathtt{U}}$

$\text{model}^{\mathtt{U}}$	::=	$\text{cluster}^{\mathtt{U}}*$
$\text{cluster}^{\mathtt{U}}$	::=	$\text{module}^{\mathtt{U}}*$ $\text{class}^{\mathtt{U}}*$
$\text{module}^{\mathtt{U}}$	::=	$\text{class}^{\mathtt{U}}*$
$\text{class}^{\mathtt{U}}$	::=	$\text{attribute}^{\mathtt{U}}*$ $\text{operation}^{\mathtt{U}}*$
$\text{operation}^{\mathtt{U}}$	::=	$\text{parameter}^{\mathtt{U}}*$

Table 5.4: UML-specific productions $P^{\mathtt{U}}$.

$\text{dependency}^{\mathtt{U}}$	$\succ^{\mathtt{U}}$	$\text{association}^{\mathtt{U}}$
	$\succ^{\mathtt{U}}$	$\text{generalization}^{\mathtt{U}}$
	$\succ^{\mathtt{U}}$	$\text{message}^{\mathtt{U}}$
	$\succ^{\mathtt{U}}$	$\text{binding}^{\mathtt{U}}$
	$\succ^{\mathtt{U}}$	$\text{uninterpretedaction}^{\mathtt{U}}$
$\text{association}^{\mathtt{U}}$	$\succ^{\mathtt{U}}$	$\text{type\_ref}^{\mathtt{U}}$

Table 5.5: UML-specific specialization hierarchy $\prec^{\mathtt{U}}$ on relations.

and relations $R^{\mathtt{U}}$ to the common meta-model are given in Table 5.7.

We follow the *lazy approach*, implementing elements as needed. Therefore, our UML meta-model does not cover all parts of the OMG meta-model definition for UML. We focus only on model information related to Class and Sequence diagrams, which are relevant for representing the static structure of the software system included in our mapping. Layout-related diagram information is ignored. Other diagram types, like Use-Case Diagrams, are currently not relevant and, therefore, they are not included.

5.4 Definition of Metrics

In this section, we define two metrics using the description framework above: Number Of Children [CK94] and Coupling Factor [eAC94]. These metrics are unambiguously defined, because documentation exists for: (1) a meta-model used for describing metrics, (2) a mapping from the front-end to the meta-model, (3) views onto the meta-model filtering unnecessary information, and (4) unambiguous description of the elements analyzed using relational algebra. Therefore, it is perfectly clear how metrics values are calculated, preparing the ground for comparable studies and the validation and standardization of metrics. For more examples, see Part II.

Number Of Children (NOC)

This metric works with all instances of a common meta-model, regardless if they were produced with the Java or the UML front-end. The respective *extends* (Java) or *generalization* (UML) relations expressing the inheritance

$\text{generalization}^{\mathtt{U}}$	:	$\text{class}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
$\text{dependency}^{\mathtt{U}}$	:	$\text{class}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
	:	$\text{class}^{\mathtt{U}} \times \text{operation}^{\mathtt{U}}$
	:	$\text{attribute}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
	:	$\text{module}^{\mathtt{U}} \times \text{module}^{\mathtt{U}}$
	:	$\text{cluster}^{\mathtt{U}} \times \text{module}^{\mathtt{U}}$
	:	$\text{module}^{\mathtt{U}} \times \text{cluster}^{\mathtt{U}}$
	:	$\text{operation}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
$\text{association}^{\mathtt{U}}$	:	$\text{class}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
$\text{message}^{\mathtt{U}}$	:	$\text{class}^{\mathtt{U}} \times \text{operation}^{\mathtt{U}}$
$\text{type\_ref}^{\mathtt{U}}$	:	$\text{attribute}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
	:	$\text{parameter}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
	:	$\text{operation}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
uninterpretedaction	:	class × class
$\text{binding}^{\mathtt{U}}$	:	$\text{class}^{\mathtt{U}} \times \text{class}^{\mathtt{U}}$
	:	$\text{module}^{\mathtt{U}} \times \text{module}^{\mathtt{U}}$
	:	$\text{cluster}^{\mathtt{U}} \times \text{cluster}^{\mathtt{U}}$

Table 5.6: Semantic relations $R^{\mathtt{U}}$ of the common meta-model.

$\alpha^{\mathtt{U}}(\text{attribute}^{\mathtt{U}})$	$\mapsto$	Field
$\alpha^{\mathtt{U}}(\text{class}^{\mathtt{U}})$	$\mapsto$	Class
$\alpha^{\mathtt{U}}(\text{model}^{\mathtt{U}})$	$\mapsto$	Model
$\alpha^{\mathtt{U}}(\text{operation}^{\mathtt{U}})$	$\mapsto$	Method
$\alpha^{\mathtt{U}}(\text{cluster})$	$\mapsto$	Package
$\alpha^{\mathtt{U}}(\text{module})$	$\mapsto$	Package
$\alpha^{\mathtt{U}}(\text{parameter}^{\mathtt{U}})$	$\mapsto$	FormalParameter
$\alpha^{\mathtt{U}}(\text{association}^{\mathtt{U}})$	$\mapsto$	FieldAccess
$\alpha^{\mathtt{U}}(\text{dependency}^{\mathtt{U}})$	$\mapsto$	ModelRelationship
$\alpha^{\mathtt{U}}(\text{generalization}^{\mathtt{U}})$	$\mapsto$	Inheritance
$\alpha^{\mathtt{U}}(\text{message}^{\mathtt{U}})$	$\mapsto$	Invokes

Table 5.7: Mapping $\alpha^{\mathtt{U}}$ to the common meta-model right).

between two classes are mapped to relations of type *inheritance* in the common meta-model (and the NOC specific view).

Description NOC is the number of immediate subclasses (children) subordinated to a class (parent) in the class hierarchy. NOC measures how many classes inherit directly methods or fields from a super-class. NOC is only applicable to object-oriented systems.

Scope Class

View $V^{NOC} = (G^{NOC}, R^{NOC})$

- Grammar $G^{NOC} = (\{\text{class}^{NOC}\}, \emptyset, \text{class}^{NOC})$
- Relations $R^{NOC} : \{\text{inheritance}^{NOC} : \text{class}^{NOC} \times \text{class}^{NOC}\}$

- Mapping α^{NOC}:

$$\begin{aligned} \alpha^{NOC}(\text{Class}) &\mapsto \text{class}^{NOC} \\ \alpha^{NOC}(\text{Inheritance}) &\mapsto \text{inheritance}^{NOC} \end{aligned}$$

Definition The NOC value of a class $c \in class^{NOC}$ is defined as:

$$NOC(c) := \left| succ(c, \text{inheritance}^{NOC}) \right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Coupling Factor (CF)

This metric also works with all instances of a common meta-model, regardless if they were produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model, and, further, to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Coupling Factor (CF) measures the coupling between classes excluding coupling due to inheritance. It is the ratio between the number of actually coupled pairs of classes in a scope (e.g., package) and the possible number of coupled pairs of classes. CF is primarily applicable to object-oriented systems.

Scope Package

View $V^{CF} = (G^{CF}, R^{CF})$

- Grammar $G^{CF} = (\{\text{package}^{CF}, \text{class}^{CF}\}, P^{CF}, \text{package}^{CF})$
- Productions $P^{CF} = \{\text{package}^{CF} ::= \text{class}^{CF\ *}\}$
- Relations $R^{CF} : \{\text{coupling}^{CF}\}$[5]

[5]The structural contains^{CF} relation is implicitly defined by the productions P^{CF}.

- Mapping α^{CF}:

$$\begin{aligned}
\alpha^{CF}(\text{Class}) &\mapsto \text{class}^{CF} \\
\alpha^{CF}(\text{Package}) &\mapsto \text{package}^{CF} \\
\alpha^{CF}(\text{Invokes}) &\mapsto \text{coupling}^{CF} \\
\alpha^{CF}(\text{Accesses}) &\mapsto \text{coupling}^{CF} \\
\alpha^{CF}(\text{IsOfType}) &\mapsto \text{coupling}^{CF}
\end{aligned}$$

Definition The CF value of a package $p \in \text{package}^{CF}$ is defined:

$$\begin{aligned}
Classes(p) &= succ^*(p, \text{contains}^{CF}) \cap \text{class}^{\texttt{CF}} \\
&\quad \textit{-- set of classes contained in } p \\
Coupled(p,c) &= succ(c, \text{coupling}^{CF}) \cap Classes(p) \\
&\quad \textit{-- set of classes contained in } p\textit{, which } c \textit{ is} \\
&\quad \textit{-- coupled to} \\
CF(p) &= \frac{\sum_{c \in Classes(p)} \mid Coupled(p,c) \mid}{\mid Classes(p) \mid^2 - \mid Classes(p) \mid}
\end{aligned}$$

Scale Absolute.

Domain Integers in $0..\infty$.

5.5 Summary

In this chapter, we discussed our solution for the unambiguous definition of software quality metrics. We started by presenting a description schema allowing a consistent description of the metrics, and easy lookup according to our double-index. Furthermore, we presented the theory and practice of our description framework, connecting front-end specific meta-models and common meta-models through language mappings. The concept of views was introduced, allowing us to describe metrics and analysis. We provided a language mapping from UML to our common meta-model instance, which is based on the Dagstuhl Middle Meta-Model. Finally, we provided two examples showing how to define software metrics in an unambiguous way.

The only thing left unclear is how to maintain the compendium and the definition of software quality metrics, in case that a new language is added together with its language mapping, and the common meta-model needs to be extended with new elements. This will be discussed in Chapter 6.

Chapter 6

Maintenance of Compendium, Metrics and Meta-Model

In the previous two chapters, we defined a description framework for metric definitions and meta-models in form of a *Compendium of Software Quality Standards and Metrics*. As indicated already in Section 5.2, we assume that our efforts of creating the knowledge base, capturing the meta-models, and defining the metrics do not deliver static and final results. We even expect contributions from the research community to extend and improve the compendium, which is a living document, cf. Section 4.1. Therefore, one of our goal criteria for reaching repeatability is maintainability. Thus, we need to be able to maintain the compendium, the common meta-model, and the metric definitions, in case the described knowledge base evolves with the introduction of new analyses, and front-ends, cf. Section 6.2.

In this chapter, we discuss how to solve this problem: first by explaining how to extend the compendium, second by providing the theory behind meta-model evolution, and then giving a real example by defining a Java front-end meta-model, its mapping to the common meta-model, and the extension of the common meta-model with new elements and relationships, which were not initially present in the version discussed in Section 5.2.

6.1 Compendium Evolution

Our *Compendium of Software Quality Standards and Metrics* is a "living document". It is supposed to be extended by us and other research groups, enriching it with new knowledge about metric definitions and their relationships to quality factors and criteria. We briefly explained in Section 4.1 how to contribute to the compendium. In this section, we describe how we maintain the Software Quality Model (SQM) and the mapping of the metrics to the SQM, and how the results of validation experiments are integrated into the compendium.

Quality Model and Metrics Mapping

The Software Quality Model (SQM) can be extended by adding new factors or criteria to it. These new elements of the SQM need to be described and put into relation with the existing elements. The evaluation of the quality model needs to be updated by reconsidering the assigned mapping levels, cf. Section 4.2. The existing metrics need to be mapped to the new criteria. The double index needs to be consistent in both directions; that is, all metrics referenced by a criterion need to reference this criterion as (highly) related.

As described in Section 4.2, these tasks are performed by an editor in order to guarantee consistency in the compendium.

Integration of Validation Experiments

The results of validation experiments supporting or rejecting a relationship between a metric and a criterion will be added to the compendium in the form of references and short summaries. Contradictions might be possible as a result of different contexts. E.g., results from one projects might suggest a high value for a certain metric, while results from another project (from a different domain) suggest low values. Anyway, this information, together with a description of the context, will be valuable.

Also, the integration of validation experiments will be performed by an editor to guarantee consistency in style, layout, expressions and quality. This is necessary, since different authors might introduce different styles, which might make it confusing to use the compendium.

6.2 Meta-Model Evolution – Theory

The initial common and view meta-models are usually designed to be suitable for a set of information sources and analyses. However, when a new information source or a new analysis is added, the meta-models could change as well. Now we discuss which kind of changes remain local, and which changes may have global effects.

Trivially, new analyses rely only on information already provided by an existing view do not trigger changes in the meta-models.

Assume analysis A cannot be applied on any of the existing view meta-models, but the required information is already encoded in the common meta-model. Then a new view meta-model $\mathcal{V}^A$ and a new view mapping α^A from the common to the view meta-model are to be specified. There is no additional implementation effort since event-generators and -handlers for creating the actual views are generated automatically, cf. Algorithms 1–5, Section 5.2.

In general, a new analysis also requires an extension of the common meta-model, which, in turn, implies that the common model creation is affected as well. Either the front-end(s) are able to provide this new piece of information, it has just not been considered relevant so far, or a new front-end needs to be integrated. In both cases, we need to extend the common meta-model $\mathcal{M}$ and the front-end F specific mapping α^F. Given that the tree- and relation event generators work according to our schemata in Algorithms 1–5, no additional programming is needed when reusing an existing front-end. Then we just specify the missing entities and relations as relevant in α^F, and the common model creation is generated automatically. New front-ends obviously require specific implementations of the Algorithms 1–5 schemata.

However, by changing the common meta-model and thereby relevant types and relations, we could even run into a reuse problem: if formerly irrelevant nodes become relevant, mappings may create relations that are not well-typed any longer. Practically, this would mean that a relation that used to be attached to one node type is now attached to a descendant of that type. This, in turn, could lead to situations where analyses cannot work as before, i.e., cannot be reused without adaptation. Fortunately, the effect of changes in the common meta-model is often not visible in the existing view-model, and, hence, many analyses can be applied without changes.

Example 6.1. Given is the common meta-model of Example 5.2 with its type and method declarations, inheritance relations on type declarations, and call relations on method declarations. Additionally, we assume the view meta-model of Example 5.4 for computing the coupling between classes CBC. Now we are to add a new (complexity) analysis that counts *statements*. This leads to the following changes: block and statement nodes are introduced in the common meta-model; its grammar productions change accordingly:

$$\begin{aligned} Program &::= TypeD^* \\ TypeD &::= MethodD^* \\ MethodD &::= Block \\ Block &::= Statement^* \\ Statement &\succ Assign|If|Loop \\ If &::= Block_{then}\ Block_{else} \\ Loop &::= Block_{body} \end{aligned}$$

The original call relation $call : MethodD \times MethodD$ changes to $call : Statement \times MethodD$ since statements are the relevant ancestors of call expressions in the meta-model now. However, when applying α^{CBC} from

Example 5.4 to this new meta-model, $\mathcal{V}^{CBC}$ remains unchanged, i.e., the view grammar is still

$$Prog^{CBC} \quad ::= \quad TypeD^{CBC\ *},$$

and the call relation is $call^{CBC} : TypeD^{CBC} \times TypeD^{CBC}$, as before. This is because type declarations are the relevant ancestors of statements in the view mapping. Hence, the old CBC analysis is applicable without any change.

What we learned from the above example is that many effects of changing the common meta-model are removed from subsequent view abstractions. This actually comes as no surprise, since the view mapping is defined by explicitly declaring relevant types; newly introduced types would not be declared relevant in existing view mapping specifications. As long as changes just extend the common model trees, view mappings would compensate and produce the original views for the existing analyses.

An exception to this involves extensions to alternative productions:

$$t \succ t_1 | \ldots | t_k | t_{new}.$$

When adding a new alternative type t_{new} to a production in the common meta-model, this type could end up in the view as well. This is because a view mapping α^A could declare the super type t as relevant and t_{new} nodes would be introduced without explicitly mapping t_{new}.

Example 6.2. To illustrate this, let us modify the common meta-model of Example 6.1 a bit; we simply add expressions:

$$\begin{aligned} If &::= Expr_{if}\ Block_{then}\ Block_{else} \\ Loop &::= Expr_{cond}\ Block_{body} \\ Assign &::= Expr \\ Expr &\succ CallExpr | ArithmeticExpr \end{aligned}$$

The call relation is now $call : CallExpr \times MethodD$ since call expressions are directly relevant in the common meta-model. Let us also change the CBC view mapping α^{CBC} a bit:

$$\begin{aligned} \alpha^{CBC}(TypeD) &= TypeD^{CBC} \\ \alpha^{CBC}(Expr) &= Expr^{CBC} \end{aligned}$$

While assuming that the original CBC analysis required type nodes with expressions as direct children of which some (the call expressions) refer to type nodes (callee).

Now we extend the common meta-model and add conditional expressions – because a new analysis requires this.

$$\begin{aligned} Expr &\succ CallExpr|ArithmeticExpr|IfExpr \\ IfExpr &::= Expr_{if}\ Expr_{then}\ Expr_{else} \end{aligned}$$

When applying α^{CBC}, the CBC views do change: type nodes have expressions as direct children of which some (the call expressions) refer to type nodes (callee) as before. However, other expressions (the newly introduced *if* expressions) have call expressions *children*. The original CBC analysis would not find and count the latter kind of calls.

In general, a changed common meta-model could change a view meta-model, which would affect existing analyses. However, there are *safe* changes to the common meta-model guaranteed not to affect an analysis A. The safe changes are defined as four Theorems 6.1, 6.2, 6.3, and 6.4 and are shown below, together with a corresponding sketch for their proof:

Theorem 6.1. *Adding a* new *type t to a sequence expression on the right-hand side of a production.*

Proof. Theorem 6.1 guarantees a safe change to the meta-model, because adding a new type to the right-hand side of a production extends the meta-model with a type, which cannot be known in existing analysis. Therefore, this type is filtered out by the views the analysis is based on. Thus, the analysis results remain unaffected in any case. □

Theorem 6.2. *Adding an* existing *type t to a sequence if no other type relevant for A can transitively be derived from t.*

Proof. Theorem 6.2 guarantees a safe change to the meta-model, because adding an existing type to a sequence extends the meta-model with that type and all transient derived types. The types will be filtered if the type and its transient derived types are not known in existing analysis, and therefore, the analysis results remain unaffected. □

Remark. If the type or one of its transient derived types is known to an analysis, the results will change, and the theorem is offended.

Theorem 6.3. *Introducing a* new *production $t ::= \ldots$ if no type relevant for A can transitively be derived from t.*

Proof. Theorem 6.3 guarantees a safe change to the meta-model, because adding a new production $t ::= \ldots$ extends the meta-model with that production and all transient derived types. The types will be filtered if the production and its transient derived types are not known to existing analysis and, therefore, the analysis results remain unaffected. □

Remark. If the production or one of its transient derived types is known to an analysis, the results will change, and the theorem is offended.

Theorem 6.4. *Adding a* new *relation.*

Proof. Theorem 6.4 guarantees a safe change to the meta-model, because adding a new relation extends the meta-model with that relation, which cannot be known in existing analysis. Therefore, they filter it out and the analysis results remain unaffected in any case. □

In all these cases, the nodes newly introduced to a common model will be filtered by existing view mappings and the relations will be attached to the original node types. Conversely, if a meta-model change is not safe for an analysis A, we should check and potentially adapt A.

Obviously, we have to produce a first common meta-model in the series, computing the essential structural and relational information. This initial meta-model is carefully designed with a first set of analyses in mind. This meta-model evolves on demand for new analyses.

As one builds more dialects into the front-ends, and has to carry more and more nodes through the intermediate representations, there is a threat that the front-ends might finally end up as kitchen sinks with everything in them. This again cannot be avoided automatically, but requires careful design when extending with new front-ends and analyses. New front-ends added should reuse existing program elements and relation types as often as possible in order not to end up with many constructs implementing the same or similar concepts.

6.3 Meta-Model Evolution – Practice

In this section, we demonstrate how extension of the common meta-model works by defining a front-end meta-model for the Java language and its mapping to the common meta-model. Then, we want to define a new analysis (metric), requiring elements in the common meta-model not present so far. The common meta-model is extended according to our *safe* changes.

program^R	::=	compilation_unit^R *
compilation_unit^R	::=	type^R +
type^R	≻	class^R | interface^R
interface^R	::=	method^R *
class^R	::=	constructor^R * method^R * field^R * initialization_block^R?
method^R	::=	statement^R *
statement^R	≻	assign^R | call_expr^R | create_expr^R | do^R | for^R | if^R | switch^R | while^R
initialization_block^R	::=	statement^R *
constructor^R	::=	statement^R *
do^R	::=	statement^R *
for^R	::=	statement^R *
if^R	::=	statement^R *
switch^R	::=	statement^R *
while^R	::=	statement^R *
assign^R	::=	expression^R expression^R *-- left-hand side '=' right-hand side*
expression^R	≻	call_expr^R | create_expr^R | read_expr^R | write_expr^R | ...
call_expr^R	::=	expression^R * *-- first designator then actual parameters*
create_expr^R	::=	expression^R * *-- actual parameters*

Table 6.1: Recoder front-end, productions P^R

call^R	:	call_expr^R × method^R
create^R	:	create_expr^R × method^R
extends^R	:	interface^R × interface^R
extends^R	:	class^R × class^R
implements^R	:	class^R × interface^R
field_access^R	:	read_expr^R × field^R
field_access^R	:	write_expr^R × field^R
type_ref^R	:	expression^R × type^R

Table 6.2: Recoder front-end, semantic relations R^R

Java Specific Model (Recoder Front-end)

This section describes a language-specific representation of the programs that the metrics are applied to. Therefore, we use a meta-model of the compiler front-end Recoder [Uni06] denoted by $M^R = (G^R, R^R)$.[1]

The model entities T^R of the grammar G^R are implicitly defined by the productions P^R, which in turn define the containment structure of the model entities and their specialization hierarchy, cf. Table 6.1. The root node type of G^R is *program*. The list of unary semantic relations (intrinsic node attributes like names, positions, and visibility) is straight forward and therefore omitted. Binary semantic relations R^R are shown in Table 6.2. There is no

[1] Actually, one should base the language mapping on the grammar and the semantic relations of the Java language-specification [GJSB00]. For brevity of presentation, we choose to start from the meta-model of a Java specific front-end instead. This does not compromise our goal of unambiguous language bindings since compiler verification is well understood for front-ends [GGZ04].

$\alpha^{\mathtt{R}}(\text{program}^{\mathtt{R}})$	$\mapsto$	Model
$\alpha^{\mathtt{R}}(\text{compilation\_unit}^{\mathtt{R}})$	$\mapsto$	SourceUnit
$\alpha^{\mathtt{R}}(\text{class}^{\mathtt{R}})$	$\mapsto$	Class
$\alpha^{\mathtt{R}}(\text{interface}^{\mathtt{R}})$	$\mapsto$	Class
$\alpha^{\mathtt{R}}(\text{constructor}^{\mathtt{R}})$	$\mapsto$	Method, Method.isConstructor = true
$\alpha^{\mathtt{R}}(\text{method}^{\mathtt{R}})$	$\mapsto$	Method, Method.isConstructor = false
$\alpha^{\mathtt{R}}(\text{field}^{\mathtt{R}})$	$\mapsto$	Field
$\alpha^{\mathtt{R}}(\text{initialization\_block}^{\mathtt{R}})$	$\mapsto$	Method, Method.isConstructor = true
$\alpha^{\mathtt{R}}(\text{call}^{\mathtt{R}})$	$\mapsto$	Invokes
$\alpha^{\mathtt{R}}(\text{create}^{\mathtt{R}})$	$\mapsto$	Invokes
$\alpha^{\mathtt{R}}(\text{extends}^{\mathtt{R}})$	$\mapsto$	InheritsFrom, InhertisFrom.inheritanceType = Extends
$\alpha^{\mathtt{R}}(\text{implements}^{\mathtt{R}})$	$\mapsto$	InheritsFrom, InhertisFrom.inheritanceType = Implements
$\alpha^{\mathtt{R}}(\text{field\_access}^{\mathtt{R}})$	$\mapsto$	Accesses
$\alpha^{\mathtt{R}}(\text{type\_ref}^{\mathtt{R}})$	$\mapsto$	IsOfType

Table 6.3: Mapping functions $\alpha^{\mathtt{R}}$

particular type hierarchy $\prec^{\mathtt{R}}$ for relations defined.

The mapping functions for mapping the language-specific types $T^{\mathtt{R}}$ and relations $R^{\mathtt{R}}$ to the common meta-model are given in Table 6.3. The mapping of actual models filters nodes of types $t^{\mathtt{R}}$ with $\alpha(t^{\mathtt{R}})$ not defined. Otherwise it keeps the containment structure of the language-specific model in the common model, but some transitive children in the language-specific model become direct children in the common model. Tuples of the language-specific $rel^{\mathtt{R}} : t_1^{\mathtt{R}} \times t_2^{\mathtt{R}}$ are mapped to the corresponding tuple of $\alpha^{\mathtt{R}}(rel^{\mathtt{R}}) : t_1 \times t_2$. If nodes of a source tuple are filtered, the target tuple nodes become the next suiting parent in the target containment structure. For example, $\alpha^{\mathtt{R}}(call^{\mathtt{R}} : call\_expr^{\mathtt{R}} \times method^{\mathtt{R}}) \mapsto Invokes : ABehaviouralElement \times ABehaviouralElement$, and $call^{\mathtt{R}}$ nodes are filtered.

Hence, a source tuple $\text{call}^{\mathtt{R}}(aCallExpr, aMethod)$ with $aCallExpr$ contained in a method $anotherMethod$ of the language-specific model is mapped to a tuple $Invokes(anotherMethod', aMethod')$ with $anotherMethod'$ and $aMethod'$ the target of $anotherMethod$ and $aMethod$, respectively, in the common model.

Currently, our Java meta-model does not cover all parts of the Java Language Specification. We include in our mapping only model information required for the metric-relevant parts of the static structure of a software system. This is a *lazy approach*; we can easily extend the model.

Extension of Java Specific Model (Recoder Front-end)

McCabe's Cyclomatic Complexity metric (CC) cannot be calculated on the original meta-model since CC requires control flow statements. Therefore,

Method	::=	FormalParameter* Statement*
Statement	≻	Loop | Condition
Loop	≻	Do | For | While
Condition	≻	If | Switch
Do	::=	Statement*
For	::=	Statement*
If	::=	Statement*
Switch	::=	Statement*
While	::=	Statement*
For	≻	EnhancedFor
EnhancedFor	::=	Statement*

Table 6.4: Extended Common Meta-Model.

$\alpha^{\mathtt{R}}(\mathrm{do}^{\mathtt{R}})$	$\mapsto$	Loop
$\alpha^{\mathtt{R}}(\mathrm{for}^{\mathtt{R}})$	$\mapsto$	Loop
$\alpha^{\mathtt{R}}(\mathrm{while}^{\mathtt{R}})$	$\mapsto$	Loop
$\alpha^{\mathtt{R}}(\mathrm{if}^{\mathtt{R}})$	$\mapsto$	Condition
$\alpha^{\mathtt{R}}(\mathrm{switch}^{\mathtt{R}})$	$\mapsto$	Condition

Table 6.5: New mapping functions $\alpha^{\mathtt{R}}$ for Java.

the common model first has to be extended, cf. Table 6.4. In particular, we introduce the following modifications of the model $M = (G, R)$ defined in Section 5.3. According to extension Theorem 6.1 (cf. Section 6.2) we introduce a new entity type *Statement* to the production: $Method ::= FormalParameter*$. According to extension Theorem 6.3 (cf. Section 6.2), we add a *Statement* including new entity types for loops and branches: *Loop* and *Switch*. No new semantic relations are added.

The mapping functions α^R for the Java front-end is extended as well, cf. Table 6.5, to map the corresponding Java entities, which already existed in the language-specific meta-model.

After extension, CC can be computed on instances of a common meta-model, as long as the required types are provided by the front-end. A UML front-end, e.g., would not construct nodes of the required type. Still, this front-end works with the new meta-model and when the metric is unambiguously defined.

Description CC is a measure of the control structure complexity of software. It is the number of linearly independent paths and, therefore, the minimum number of independent paths when executing the software.

Scope Method

View $V^{CC} = (G^{CC}, \emptyset)$.

- Grammar $G^{CC} = (T^{CC}, P^{CC}, \mathrm{method}^{CC})$

- Entities $T^{CC} = \{\text{method}^{CC}, \text{ctrl\_stmt}^{CC}\}$
- Productions P^{CC}:

$$\begin{aligned} \text{method}^{CC} &= \text{ctrl\_stmt}^{CC\,*} \\ \text{ctrl\_stmt}^{CC} &= \text{ctrl\_stmt}^{CC\,*} \end{aligned}$$

- Mapping α^{CC}:

$$\begin{aligned} \alpha^{CC}(\text{Method}) &\mapsto \mathit{method}^{CC} \\ \alpha^{CC}(\text{Switch}) &\mapsto \text{ctrl\_stmt}^{CC} \\ \alpha^{CC}(\text{Loop}) &\mapsto \text{ctrl\_stmt}^{CC} \end{aligned}$$

Definition The CC value of a method $m \in \mathit{method}^{CC}$ is defined as:

$$CC(m) := \left| \mathit{succ}^{+}\left(m, \text{contains}^{CC}\right) \right|$$

Scale Absolute.

Domain Integers in $1..\infty$.

See Part II for more details about this metric and its relationship to the software quality model.

6.4 Summary

In this chapter, we discussed our solution for maintaining the *Compendium of Software Quality Standards and Metrics*, its included meta-models, language mappings, and metric definitions. We provided the theory for an extensible meta-model being based on tree grammars, thereby providing a set of theorems which need to be observed when extending the common meta-model. With this extension and together with the abstracting views already discussed in Section 5.2, our maintainability criterion are realized.

Chapter 7

VizzAnalyzer

To reach our goal of repeatability, we need to satisfy the automation criterion. This is considered fulfilled if we provide appropriate tool support implementing the proposed metric definition process, including the front-ends, meta-models, and metrics. Since we already had a suitable tool at hand, which provides the foundations for most of the needed tasks, we could reuse and extend this tool, without needing to develop such a tool from scratch. This tool is the VizzAnalyzer, which will be presented in this chapter.

Besides giving a short overview of the VizzAnalyzer and its architecture, we will present examples of extending the VizzAnalyzer with new metrics and new front-ends, as well as some benchmark results. Furthermore, we are going to describe the technical side of the extensions made to the VizzAnalyzer for two case studies. These case studies and their evaluation will be described in Chapter 8.

7.1 VizzAnalyzer

The VizzAnalyzer tool[1] is an instantiation of the VizzAnalyzer reverse engineering framework [LP05a], which has been developed at Växjö University. It can be seen in action in Figure 7.1. Its flexible architecture allows us to integrate tools for information extraction, analysis, and visualization.

Architecture

Figure 7.2 shows the architecture of the VizzAnalyzer. Its core architecture is an implementation of the architecture for constructing, capturing, and accessing a model of a software system presented in Figure 5.1, Section 5.2. There, we required four components (cf. circles in Figure 5.1), which have been realized by the VizzAnalyzer architecture. We included the relevant part of that Figure 5.1 in this Section as Figure 7.3 for easy reference.

These components are:

[1] `http://www.arisa.se`

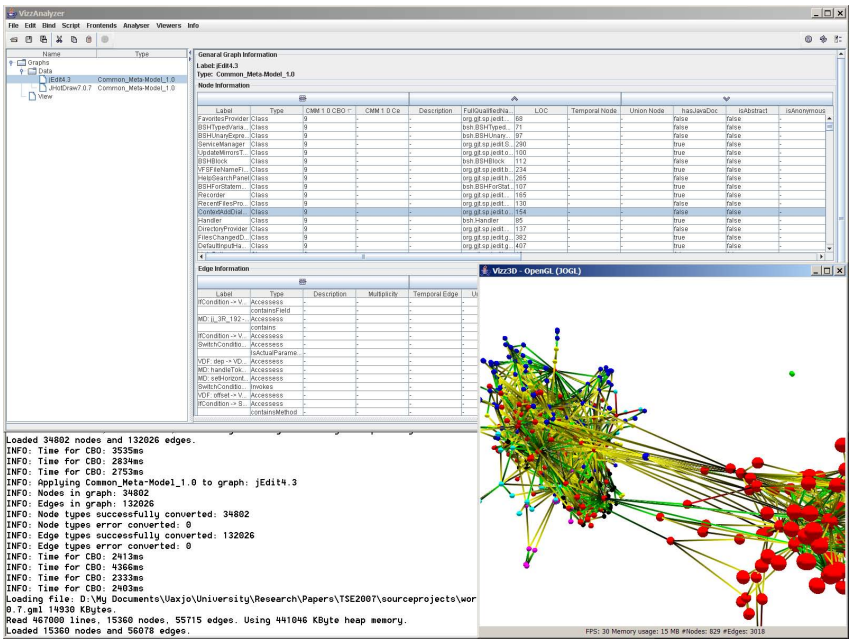

Figure 7.1: Example: Metrics analysis and visualization of jEdit and JHotDraw. The VizzAnalyzer (top) shows nodes and edges of jEdit including same basic information and metrics values (CBO and LOC). Vizz3D (bottom right) shows a 3D visualization of the call and access structure between classes of jEdit using a force-based clustering algorithm. Coloring of nodes is according to packages. The figure is taken from [SLLL06].

1. Different concrete information-extracting front-ends for programming languages or other program representations, which capture information about a program in a *front-end specific model*. They are implemented as source plug-ins, represented by *Recoder* in Figure 7.2, and as *parses*, in Figure 7.3.

2. Converters mapping this front-end specific to a language-independent *common model* capturing program information relevant for later analysis. They are implemented within the core, represented by *GML and GXL* in Figure 7.2, and as *parses* in Figure 7.3.

3. Abstractions computing *views* on the common model specific for a subset of analyses. They are implemented as analyses plug-ins, represented

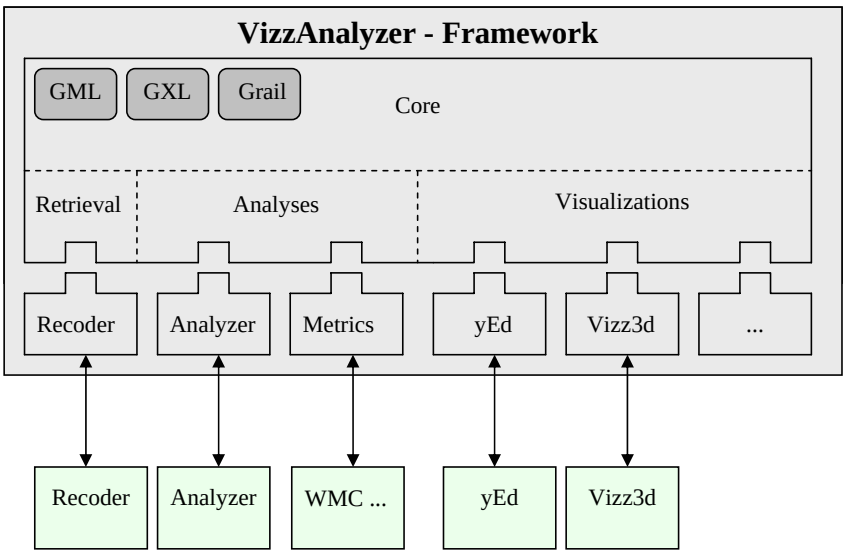

Figure 7.2: Architecture of the VizzAnalyzer Framework. It displays core components and connectors to retrieval, analyses, and visualization plug-ins.

by *Analyzer* in Figure 7.2, and as *abstracts* in Figure 7.3.

4. Different, concrete *analyses* accessing their respective views. They are implemented as Metrics plug-ins, represented by *WMC* in Figure 7.2, and as *Analyzes* in Figure 7.3.

More information on the VizzAnalyzer Framework, including details about the architecture, design and usage of the VizzAnalyzer, is available from the latest publications [PLL04, PLL05, LP05b, PS05, PLLL05].

Relevant Parts

It is important to know that the flexible architecture of the VizzAnalyzer allows us to plug in new components, and to define meta-models based on the internal graph-based data structure. The important parts for our work are the retrieval and analysis plug-ins, as well as the data store (GRAIL) as seen in Figure 7.2. The Metrics discussed in the previous chapters are implemented as *Analyses* plug-ins. The UML (cf. Section 5.3) and the Java Front-ends (cf. Section 6.3) are implemented as *Retrieval* plug-ins.

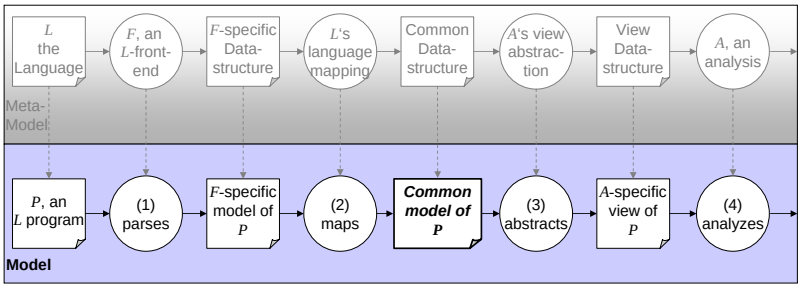

Figure 7.3: Process of information extraction, mapping to language-independent representation, abstraction and analysis on model. The figure is taken from [SLLL06].

The architecture for the retrieval plug-ins is event-based. That is, a specialized tool provides access to the native data structure of the UML or Java program representation, thereby providing access to it according to the front-end meta-model. A so-called walker is traversing this data-structure, sending events to listeners. This corresponds to the mapping between the front-end meta-model and the common meta-model. The listeners receive the events and accordingly create the elements and relations using the common meta-model elements. This architecture and implementation is based on the algorithms presented in Section 5.2, Algorithms 1–5.

VizzFrontEnd

This subsection provides an overview of the architecture used for the VizzAnalyzer retrieval (front-end) plug-ins. It is an implementation of the Algorithms 1–5 we presented in Section 5.2. We give a brief overview of the process as a whole, and present an example in the following sections.

Design Goals are as follows:

- To separate the language/tool specific "source code reader" from the VizzAnalyzer front-end. That is, it should be possible to completely remove the source code reader from the memory once graph construction is completed.

- To provide a simple/flexible interface where new source code readers can be added without too many complications.

```
 1 public interface VizzFrontEnd {
 2   public void addLeaf(TypeValues nodeType,
 3                       Object nodeKey,
 4                       String label,
 5                       Attribute[] attributes);
 6
 7   public void enterNode(TypeValues nodeType,
 8                         Object nodeKey,
 9                         String label);
10
11   public void exitNode(TypeValues nodeType,
12                        Object nodeKey,
13                        Attribute[] attributes);
14
15   public void addCrossReferenceEdge(TypeValues edgeType,
16                                     Object srcKey,
17                                     Object tgtKey,
18                                     Attribute[] attributes);
19
20   public void completeGraphs();
21 }
```

Figure 7.4: VizzFrontEnd interface.

VizzFrontEnd is the only access point for source code readers. It is designed to be simple, flexible, and intuitive. It contains tree methods related to the containment tree (CT) construction: one method to add cross-references, and a method to inform the front-end when the reading is done.

The graph construction has tree phases:

- Construct the tree using the methods `addLeaf` (lines 2–5), `enterNode` (lines 7–9), and `exitNode` (lines 11–13). The VizzAnalyzer front-end uses this information to construct a CT for each one of the requested graphs (defined by graph specifications). The actual information stored in these trees depends on their graph specification. However, the source code reader knows nothing about the constructed graphs, and constructs a complete CT. (The content in a complete CT depends on the specific source code reader). This corresponds to Algorithms 1, 2, and 3, cf. Section 5.2.
- Add cross-reference edges calling `addCrossReferenceEdges` (lines 15–

18) repeatedly. The two keys (srcKey, tgtKey) in the call must have been added during the first phase. This corresponds to Algorithm 4, cf. Section 5.2.

- Use the method `completeGraphs` (line 20) to ask the front-end to complete the graphs. This informs the VizzAnalyzer front-end that no more source code events will be generated. The front-end will clean up the graphs and release all source code references. This corresponds to Algorithm 5, cf. Section 5.2.

Noteworthy is, first, that the keys used by the source code reader are not used in the constructed graphs. The front-end maintains a key to key mapping which is released after the graph construction is completed. Thus, it releases any references related to the source code reader. Second, the properties (e.g., size and modifiers) related to the different node types are represented as attributes. An attribute is a pair `(GraphProperties property, Object value)` that should be interpreted as `property = value`. A `null` value is interpreted as `property = true`. The set of properties used, as well as the set of node types used, is determined by the source code reader.

7.2 Extending the VizzAnalyzer

Adding a New Analysis

The initial meta-model was suitable for a number of object-oriented metrics. Adding McCabe's Cyclomatic Complexity (CC) metric [WM96] was not possible in the initial meta-model. CC is a measure of the control-flow complexity of a method, defined as the number of linearly independent paths.

The control structure of a method in object-oriented languages is encoded with `if`, `for`, `do`, `while`, and `switch` statements. For the purpose of simplicity, any exceptions, tertiary operators, and boolean expressions in the control statements are neglected. CC of a method can be computed by counting control statements contained in a method (adding 1 since methods without branches implement a single path).

The initial meta-model did not contain control statement entities. A method declaration was a sequence of call expressions:

$$MethodD \quad ::= \quad CallExpr^*$$

In order to calculate CC, we extend the meta-model with a new type *CStmt*

	Lines of code	Model build time	Analysis time	Model size
`JEdit`	145508	91s	3s	100MB
`JHotDraw`	57020	42s	<1s	54MB

Table 7.1: CPU and memory requirements of the model

and the following productions:

$$\begin{aligned} MethodD &::= (CallExpr|CStmt)^* \\ CStmt &::= (CallExpr|CStmt)^* \end{aligned}$$

The front-end mapping – initially there was only one `Java` front-end – was extended to map all `Java` control statements to *CStmt*:

$$\begin{aligned} \alpha^{\mathrm{J}}(If^{\mathrm{J}}) &= CStmt \\ \alpha^{\mathrm{J}}(For^{\mathrm{J}}) &= CStmt \\ &\ldots \end{aligned}$$

A new view mapping α^{CC} trivially defines:

$$\begin{aligned} \alpha^{CC}(Program) &= Program^{CC} \\ \alpha^{CC}(MethodD) &= MethodD^{CC} \\ \alpha^{CC}(CStmt) &= CStmt^{CC} \end{aligned}$$

On these views, Cyclomatic Complexity of a method m could simply be computed by counting the control statement descendants under m.

Altogether, we needed to add less than 100 lines of specification and `Java` code. To put it more precisely, the walker traversing the AST in the front-end was extended to generate events for `if`, `for`, `do`, `while`, and `switch` statements. The common meta-model specification was extended by the *CStmt* type and the corresponding grammar productions. The extensions of the walker implementation and the meta-model specification generated the extended mapping of front-end specific to common models. The *CC* view and the mapping specification from the common meta-model to the view were defined as described above. Together, the view specific meta-model and the mapping specification generated the new view abstraction. Finally, the *CC* metric was implemented.

Adding a New Front-End

The initial, RECODER[2]-based front-end could only handle `Java` source code. This means that the influence of external libraries provided in byte code

[2] `http://recoder.sourceforge.net`

was neglected. It is well-known that all more-precise call graph construction algorithms require some sort of data-flow analysis, which, in turn, requires a whole program representation [TP00, GDDC97]. To add these kinds of analyses, we integrate another front-end that is based on the byte code reader of the SOOT Framework.[3] It constructs a program representation where each method is represented by a basic block graph, and each basic block contains a sequence of statements. The edges in the basic block graph represent the control-flow.

The transition from the RECODER- to the SOOT-based front-end only involved adding two new node types (*BasicBlock* and *Alloc*), and two new relations (*controlFlow* and *allocates*) to the common meta-model.

All coupling and cohesion metrics previously developed for the common model could immediately be reused. The Cyclomatic Complexity metric could, however, not be reused since it requires the control statement node type – a node type that the SOOT front-end never generates.

Benchmarks

To show the real-world fitness of VizzAnalyzer we show how it performs on some real-world projects. The results are shown in Table 7.1. System configuration was: Windows XP SP2, Java 1.5, Pentium M 1.7GHz, 1 GB RAM. All measured times are overall times, measured between invocation of the commands and retrieval of the results, including parsing, building of the representation and writing of the result to a file. The values measured are:

- Model-building time; that is, the time needed to build the full model from the source code. This includes parsing time and model creation time.
- Analysis time; that is, the time needed to perform a particular analysis on the model. In this case it was the *CBC* metric described in Section 5.2, Example 5.4.
- Model size; that is, the amount of memory in RAM used by the model after it has been completely created.

7.3 Using Common Meta-Model 1.0

Common Meta-Model 1.0, which has been discussed theoretically in Sections 5.2 and 5.3, has been implemented in the VizzAnalyzer. Its implementation consists of:

[3] `http://www.sable.mcgill.ca/soot`

- A profile for the GRAIL component, defining the types of the elements, relations, and their type hierarchy, together with the possible attributes. This represents the common meta-model, cf. Common Datastructure, Figure 5.1 and 7.3. The elements, relations, and attributes constituting a certain stable version, are numbered to distinguish them from future modified versions. The current version as with this thesis, is *Common Meta-Model 1.0*. It is based on the Dagstuhl Middle Metamodel, has been introduced in Section 5.3, and extended upon in Section 6.3.

- A mapping document describing the mapping from the front-end specific meta-model elements, relations, attributes, and types to the common meta-model. This information has been provided for Java Recoder and UML, as discussed in Sections 5.3 and 6.3, cf. Language Mapping, Figure 5.1 and 7.3.

- Retrieval plug-ins implementing the source code reader, extracting the information from the source format, converting it according to the mapping functions into a common meta-model, cf. Front-end/Parses, Figure 5.1 and 7.3.

- A set of analyses calculating the views on the common meta-model and implementing the metrics according to the definitions in the compendium, cf. View Abstraction/Abstracts, Figure 5.1 and 7.3.

Eclipse Front-End

The Eclipse Front-end is a new plug-in for the VizzAnalyzer, which has been used to analyze the project RANDIS, see Chapter 8. It is, like the Recoder front-end (cf. Section 6.3), a Java parser, but it is based on the Eclipse AST API for Java. It has the advantage over Recoder, because it also provides detailed information about JavaDoc, allowing us to access single JavaDoc comments, their text, tags, values and return values, as well as references like links. It is also Java 5 and Java 6 source code compatible.

The front-end implements the architecture described in Section 5.2, implementing the `VizzFrontEnd` interface (cf. Figure 7.4) as `DefaultBackEnd` class and creating graphs of the common meta-model 1.0. The walker traversing the Eclipse AST is implemented by class `EclipseJavaFrontEnd`. It accesses the Eclipse internal representation of the Java source code, which corresponds to the front-end specific meta-model that has been discussed in Section 5.2. It traverses the AST, and for each element found it creates events

```
  1 public class CommonMetaModel10 {
  2
    ...
 30
 31    public static final TypeValues AValue = getTypeInstance("Class",
 32                          TypeValues.TYPE_NODE, AStructuralElement);
    ...
143    public static final TypeValues Method = getTypeInstance("Method",
144                          TypeValues.TYPE_NODE, ABehaviouralElement);
    ...
365    public static final TypeValues IsMethodOf = getTypeInstance(
366                          "containsMethod", TypeValues.TYPE_EDGE, contains);
    ...
500    validContainsEdges.put("" + Class + Method, IsMethodOf);
    ...
600
601 }
```

Figure 7.5: Implementation of the Common Meta-Model 1.0 GRAIL profile, defining node and edge types and general attributes defining the elements and relations of the meta-model.

the `DefaultBackEnd` is listening for. The events and the related transformation from the front-end specific meta-model to the common meta-model is the mapping discussed in Section 5.2.

Common Meta-Model 1.0

Figure 7.5 shows how the common meta-model is implemented using the VizzAnalyzer Framework. It defines a set of `TypeValues`, which are used to generate a profile for a grail graph. Grail graphs can usually represent anything, but by assigning the nodes and edges types according to a certain type system, they can be used for special purposes. Line 31–32 in Figure 7.5 shows how a class is defined in the meta-model. The type of that class has the super type `AStructuralElement`. Line 143–144 shows how a method is defined in the meta-model. The type of that method has the super type `ABehaviouralElement`. Line 500 defines that the class and method are connected by a relation (defined in line 365–366) of type `IsMethodOf`. This implementation corresponds to the *Common Meta-Model 1.0* presented in Section 5.3.

Metrics

Figure 7.6 shows how a metric is implemented using the VizzAnalyzer Framework. This corresponds to Analyses, which is seen in Figure 7.3, Section 7.1. Extending from the `MetricsAnalysis` class, the metric hooks into the framework as an analysis plug-in (line 0). In `initProperties()`, the view model

```
 0 public class CMM_1_0_NOM_NumberOflocalMethods extends MetricsAnalysis {
 1  private void initProperties() {
 2     Model model = new Model();
 3
 4     model.addNodeType(CommonMetaModel10.Class);
 5     model.addNodeType(CommonMetaModel10.Method);
 6     model.addEdgeType(CommonMetaModel10.IsMethodOf);
 7
 8     view = new ModelView((DirectedGraphInterface) newGraph, model);
 9  }
10  public void startAnalysis() {
11     initProperties();
12
13     NodeIterator classNodeIterator = view.nodes();
14     DirectedNodeInterface classNode = null;
15     TypeValues nodeType = null;
16     int nom = 0;
17
18     while (classNodeIterator.hasNext()) {
19        classNodeIterator.next();
20        classNode = (DirectedNodeInterface) classNodeIterator.getNode();
21        nodeType = (TypeValues) classNode.getProperty(GraphProperties.TYPE);
22        if (nodeType != null && (nodeType.equals(CommonMetaModel10.Class))) {
23           nom = getNOMperclass(classNode);
24           classNode.setProperty(nomValue, new Integer(nom));
25        }
26     }
27  }
28  private int getNOMperclass(DirectedNodeInterface classNode) {
29     int nom = 0;
30
31     nom = view.outDegree(classNode);
32
33     return nom;
34  }
35 }
```

Figure 7.6: Implementation of the Number Of local Methods metric (NOM), using Common Meta-Model 1.0. A view is defined containing only classes and methods.

of a graph representing a software system according to the common meta-model 1.0 is created. The graph is available as `newGraph` (line 8). The metric analysis itself is implemented in lines 10–27. The analysis iterates over all classes in the view, and counts the attached methods, simply by querying the out-degree. Since the view is a forest consisting of class nodes, having their method nodes attached over edges, no other nodes need to be considered.

7.4 Summary

We introduced the VizzAnalyzer reverse engineering framework as an architecture for the composition of reverse engineering and measurement tools.

The VizzAnalyzer tool implements this architecture. It has been reused and extended to implement an architecture for constructing, capturing, and accessing a model of a software system presented in Figure 5.1, Section 5.2. The required four components (cf. circles in Figure 5.1) have been realized by the common meta-model as the central data store, while different retrieval plug-ins provide access to Java source code and UML through front-end specific meta-models, and mappings from these specific meta-models have been defined. The five algorithms described in Section 5.2 were realized in an event-based architecture performing these mappings. New analyses have been written for implementing the view and metric definitions provided in the compendium. They compute metric values upon the common meta-model. They are safe from side effects of extensions to the meta-model, since the extension of the Common Meta-Model 1.0 is protected by four theorems, introduced in Section 6.2, and by the abstractions of views. We provided code samples of the metric implementation, common meta-model definition, and front-end implementation, together with other examples and benchmarks, to demonstrate the feasibility and flexibility of our solution.

Chapter 8

Case Studies

In this chapter, we present results from two case studies we used our tool in. We briefly describe the backgrounds of the case studies and their goals. We discuss how we used the VizzAnalyzer to conduct the studies and which modifications were necessary. We reflect on the studies and the lessons learned.

8.1 Overall Target Architecture Activity

The EUROCONTROL Overall ATM/CNS Target Architecture (OATA) develops a high level design of an integrated Air Traffic Management (ATM) "system of systems" across all ECAC States.[1] It will supersede the current collection of individual national systems [EUR07].

So far, the lack of an overall, high level design has led to a currently insufficient level of integration and interoperability. EUROCONTROL aims at changing this so that the introduction of Operational Improvements – as envisaged in the European ATM 2000+ Strategy – can proceed at the pace required by the market [EUR07].

In the context of this project, the VizzAnalyzer tool was used to conduct a metrics based analysis of the developed architecture. This project was a so called *pilot validation*, which prepared the real validation. The goal was to calibrate the VizzAnalyzer for the current project and to resolve existing problems in order to start monitoring all the architecture from future iterations on. The pilot validation focused only on a subsystem of the complete architecture, which consisted of 8 modules and 70 classes. The participating companies defined a set of candidate metrics, which ought to quantify the architecture represented as a UML model (class and sequence diagrams), with respect to quality goals.

We implemented a new VizzAnalyzer front-end for reading in the UML model information. Furthermore, the proposed metrics were analyzed and

[1] European Civil Aviation Conference; an intergovernmental organization with over 40 European states, active since 1955 in promoting the co-ordination, better utilization and orderly development of European civil aviation in the economic, technical, security and safety fields.

implemented. The 16 metrics were taken from well known metric suites, similar to the metrics defined in the compendium. They included: Number Of Methods (NOM), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Coupling Between Classes (CBC), Coupling between object classes (CBO), Coupling Factor (CF), Afferent Coupling (Ca), Efferent Coupling (Ce), Instability (I), Cyclic Dependencies (CYC), Response For a Class (RFC), Encapsulation Principle (EP), Length of Names (LEN), Name Uniqueness Ratio (UNIQ), and Comment Density (DC).

The definitions of the metrics were taken from standard literature, and we found now in practice what we discussed in theory in the previous sections of this thesis: the metric definitions were too ambiguous to be implemented, even too ambiguous to be discussed, while being sure that other participants in a conversation have the same understanding. We needed to create a mini-compendium describing the metrics in the variant that should be used for the project. Since the metrics should be used for UML and no other use was currently planned, they were defined upon a UML and project specific version of the common meta-model. The UML font-end meta-model and mapping has been presented in Section 5.3.

The pilot validation was successfully conducted, and metric values were computed. They were analyzed to find self-related outliers pointing on issues in the architecture. The results were presented to the project partners.

The following lessons could be learned:

1. Several issues with the metric definitions existed.
   (a) Unclear and inexact definitions, as well as using the same terms in different context, leads to confusion and misunderstanding.
   (b) Different variants of the same metric were not distinguished by name, thereby making it difficult to know which one was meant.
   (c) Well known metrics from the literature were reused without proper definition, with wrong intentions, and partially changing the purpose of the metrics, thereby causing confusion.
2. Inconsistent and incomplete UML Model
   (a) Since the UML model is an abstract model and not a detailed implementation, the model is neither consistent nor complete. This causes problems for the front-end reading in the data, and for the analyst interpreting the results. A question is whether the values are so good/bad because it is bad design or because it is not complete yet.

   (b) Robustness of the front-end is needed, since inconsistencies exist.
   (c) Different interpretations for metric values are needed, as for values calculated for systems provided as source code.

3. Deviations from metrics calculated with other metric tools.
   (a) The OATA project from the beginning used a different metric tool called Ntools. Even though our tool and Ntools use the same informal metrics definitions, the results were not comparable. As discussed in the previous chapters of this thesis, this could be expected, since no shared meta-model and language mappings were used. Each tool implemented the metric definitions according to its own interpretation. This was still ambiguous, even though the metrics have been discussed several times in meetings.
   (b) Additionally, we found that the scope of the metrics calculation was not (sufficiently) defined. This made it even more difficult to get the same results using both metric tools. This aspect of defining metrics has never been considered in the literature so far. A simple metric counting the classes in a software system might deliver different values for different tools: which classes shall be counted at all? Does all include test cases, deprecated classes, dead code, 3rd party and runtime classes? Where does the software system start and where does it end? Shall calls to the API or a special library count?
4. We created a Metrics Definition Document for communicating the metric information, but it contained no meta-model or language mapping, and the metrics were described using natural language and semi-formal approaches using simple mathematics, but not formal approaches, involving set theory upon a well defined meta-model.

Most of these issues were addressed in the next iteration. The scope of the application of metrics was included in the definition, the metrics definitions were improved, but still no meta-model or language mapping is formally documented. The tool implementation represents the specification. We hope to improve most of these issues in successive iterations, then repeat the old measures and re-evaluate them.

8.2 *First Contact Analysis of* RANDIS

In the context of the research project of this thesis, the software system provided by Combitech AB as project partner has been analyzed using the

VizzAnalyzer tool. The context of the research project is presented in the following subsection. Then follows an overview of the project analyzed, modifications to the VizzAnalyzer, and a summary of the results.

Context

Validation of metrics-based quality control is a project founded by the KK-foundation.[2] Within this project, we develop a method for quality assessment integrating industrial standards and automated software measurement (metrics). The research goal is the validation of this method in industry projects. The goal for the partner companies is an improved quality management. In selected development projects, partner companies assess quality characteristics suggested in ISO/IEC 9126 using automated and manual measures. We try to validate statistically the significance of the automated approach. The expected result is a service assessing industrially standardized software qualities in an effective (since significant) and efficient (since automated) way.

Project Randis

As described in Section 1.2, one of the project partners is Combitech AB. Combitech is a provider of services in system development, integration, information, and system security. They participate in the project Randis, which is an avionic system used in Norway. It runs on a JBoss application server under Linux. It is implemented in Java 1.4.x, JSP and connected to a MySQL database. The software is component based, containing Java Applets and HTML-code. We analyzed this project applying a First Contact Analysis,[3] and documented our findings in a report.

What was analyzed?

The goal of our analysis was to get a first impression of the software system by analyzing its architecture and structure, design, and complexity. We calculated some global statistics to get a feel for the nature of the project. This included information about number of packages, classes, interfaces, methods, fields, and lines of code. The analysis of architecture and structure consisted of an evaluation of the system architecture at large. For that purpose, the

[2] KK-stiftelsen: Project 2005/0218; Validation of metrics-based quality control. `http://www.arisa.se/index_projects.html`

[3] ARiSA First Contact Analysis™ is an assessment method for software systems. It is based on a number of analyses successfully applied to a variety of systems. It gives a graphical overview of a system's architecture and structure, design, and complexity and uncovers potential quality issues in the software system. Cf. `http://www.arisa.se/Q2.html` for more details.

system was visualized using 2D and 3D visualization tools and clustering techniques for finding the natural components of the system through visualization. This was based on interaction graphs on the class and packages level. Furthermore, the inheritance structure was investigated. Several object-oriented design metrics were used to find outliers with respect to cohesion, coupling, and complexity on the class and package levels. The results were visualized using 2D diagrams. The findings were summarized and conclusions drawn.

Lessons Learned

Our assessment as part of the First Contact Analysis included a general analysis of the system architecture. A comparison between intended and actual architecture still needs to be done. For doing this, the designers/maintainers of RANDIS need to be involved. An analysis of the inheritance structure has been completed. Some detailed analysis of the subsystems is required after the current results have been discussed with the RANDIS maintenance team. Analyses of coupling and cohesion on class level, and analyses of the understandability and maintainability of the software are completed. Some classes as starting points for discussions have been identified.

We separated analyses from interpretation. All analyses results are factual. However, we need to emphasize that their interpretation should be taken with care. It can only point to suspicious spots in the system. Developers and designers experienced with the system should manually check each interpretation.

None of the results revealed a severe problem, but still, the actual and the intended design need to be compared. It is now the task of the RANDIS maintenance team to evaluate the results documented by our written report using their expert knowledge. Together we will draw conclusions if the current design and implementation of the software provides a solid foundation for future development cycles and how the used analysis and metrics should be adjusted to measure the quality factors that are important for the maintenance.

VizzAnalyzer extension

The VizzAnalyzer tool needed to be extended with a new retrieval plug-in, allowing us to deal with certain code constructs the applied Recoder front-end could not deal with (Java 1.5, JavaDoc). This retrieval plug-in was the Eclipse Front-end, which has already been discussed in Section 7.3. Furthermore, we defined a new language mapping, which maps the front-end specific elements and relationships to the Common Meta-Model 1.0 (CMM10). The

CMM10 and all the previously defined metrics could be reused without any modification.

8.3 Summary

In this chapter, we presented an evaluation of the feasibility of our approach for defining software quality metrics in an unambiguous way. We conducted two case studies, one within the OATA project, and the other within the RANDIS project. These showed that our implemented tool support could realize the architecture we introduced in Sections 5.2 and 7.1, thus saving development effort when extending the meta-model, set of metrics, and front-ends, cf. Section 6.2. Additionally, we could confirm the correctness and relevance of the well-definedness and repeatability goal criteria described in Section 1.3.

The first case study (OATA) was discussed in Section 8.1. It gave us insight into the definition process of software metrics and confirmed our theoretical assumptions that informal definitions of metrics lead to problems. We could, furthermore, gain some insight in related and so far not discussed problems, regarding the definition of scope when measuring metrics. Furthermore, we could see that two tools fail to provide the same results, even though they are based on the same (informal) definition. These experiences confirmed our first goal of *Well-definedness*, as described in Section 1.3. Additionally, we found some problems arising from the incomplete and inconsistent nature of UML models, which might be a potential threat to applying software metrics which target on source code.

The second case study (RANDIS) was described in Section 8.2. Here, we were not confronted with new challenges. Instead, the case study showed that our current approach seems to work fine for source code based analysis. We found that the extension of the VizzAnalyzer and the knowledge base behind it works well with the provided architecture and theoretical foundations. We could reuse metrics defined on a meta-model, and use them when creating a model from a new front-end. This confirmed our third goal of *Repeatability*, and in particular the sub criteria *Reusability* and *Maintainability* as described in Section 1.3.

In both case studies we could confirm the relevance of our *Automation* criterion, since we needed to extract data repeatedly during test and validation. Having a fully automated data extraction, analysis, and report process significantly saved time.

During these two case studies, we learned a few lessons. As we have seen in the case of the OATA project, a sound definition of software quality met-

rics is essential. It is not sufficient to define the metrics in natural language or in semi-formal way, because even after deep discussions there remain misunderstandings. Therefore, a formal definition upon a meta-model, including language mappings from the front-end specific to the shared meta-models, is essential. This confirms our first goal of *Well-definedness*, as described in Section 1.3. In addition to the definition of metrics, it is of high importance to describe the scope of the metrics applied.

Since there exists a large number of metrics in literature, different variants of metrics need to be handled by different names. Otherwise, misunderstandings are inevitable. This confirms our idea of creating a compendium of software metrics, cf. Section 4.1, in which all these definitions can be documented.

In comparing our tool results with the second tool (Ntools) in the OATA project, we saw our assumptions confirmed that ambiguous metric definitions lead to incomparable results among different tools.

Furthermore, we noted some issues which are likely to apply in general to projects performing analysis on a non-source code basis. UML models are abstract and not necessarily consistent. This can be intended, depending on the development phase, or simply mistakes. Unlike with source code, it is difficult to find out if something is wrong, since there is no "compiler" available.

In our case, the validation team was separated from the design team. We received the model for evaluation as is, and we were not in the position to perform any corrections on the model prior to analysis. Because of this and inconsistencies in the existing model, we experienced over 1000 warnings exporting the OATA model from *Rational Rose*[4] to XMI. The warnings informed about attributes having no type assigned, missing association ends, parameters without type, etc. All these inconsistencies visible as warnings affect the measures performed. We conducted our analysis and recommend for the future to reduce the number of model inconsistencies.

The model inconsistencies affected also our meta-model, since we received warnings from the UML front-end (cf. Section 5.3) reading in the exported XMI files. Problems included: attributes associated to associations, parameters not associated to operations, associations with less than two ends, undefined stereotypes, etc. We handled these errors, but we also need feedback from the model designers if our error handling is in their spirit.

Calculating some of the metrics also leads to problems, since some metrics could not be calculated because of missing elements in the model.

[4] A UML modeling tool, `http://www-306.ibm.com/software/awdtools/developer/rose/`.

Regarding the evaluation process itself, we noted that the Metric Definition Document (MDD) we created for defining and discussing metrics was suitable in specifying the metrics itself, but it lacked some detail on specifying the scope on which to apply the metrics. This caused problems in comparing the results of the two tools. Further, it caused significant extra work in adjusting the scope of the VizzAnalyzer in the attempt to cover the same scope as NTools.

Our tool architecture proved very flexible, allowing the easy extension and reuse of metrics. We saw this in particular in the RANDIS project, as we implemented a new front-end and mapping, and we could still reuse the existing metric implementations without any problems. Therefore, we could save days of development effort. This confirms our third goal, *repeatability*, cf. Section 1.3.

Chapter 9

Summary

This last chapter of the thesis summarizes our work, draws conclusions, and shows directions for our future work.

9.1 Conclusion

Summarizing our long term research goal (cf. Section 1.3), our intention is to validate a standard- and metric-based software quality model. Yet, before conducting this research, we need to solve a number of related problems. We found that ambiguous metric definitions lead to incomparable implementation variants in tools, and make large scale empirical experiments and generalization of results impossible. Furthermore, for relating metrics and quality attributes, a large number of explicit and implicit software quality models, processes, and methodologies have been proposed (cf. Section 3.4). Additionally, as for all experiments, the performed measures need to be repeatable in exactly the same way in all compared projects.

In Section 1.3, we defined the problem to be solved by this thesis as follows:

> *The problem tackled by this thesis is the unambiguous definition of Software Quality Metrics, the definition of a standard-based Software Quality Model (SQM) that relates the well-defined metrics to the criteria of the SQM, and the development of appropriate tool support in order to facilitate the conduction of our research experiment in a repeatable and efficient way.*

We formalized three goals and appropriate criteria for solving this problem. They have been addressed individually throughout the thesis in the following way:

Well-definedness This goal is reached if the complete data extraction and metrics computation process is described in such detail that it is possible for different research groups to implement their tools in such a

way that different tools measuring one and the same software system calculate the same metric values.

Even though we did not provide any empirical proof that our metric definitions implemented by different research groups lead to the same results, we believe that our proposals should resolve ambiguity, thereby satisfying *this criterion* through the shared and unambiguous specification of software metrics in the form of a public *Compendium of Software Quality Standards and Metrics*, as discussed in Chapter 4. This compendium alone is not enough; we need a description schema for the metrics (cf. Section 5.1), a meta-model for formally defining metrics using set theory and simple mathematics (cf. Section 5.2), front-end specific models, and language mappings for unambiguity of the meta-model itself (cf. Section 5.2).

Standard-based This goal is reached if the quality model used is based on an existing standard summarizing the state of the art/best practices of the last decades, thereby gaining a high trust level. The metrics are selected from well known and discussed metrics in current literature.

This criterion has been satisfied by selecting the ISO 9126-3:2003 software quality model, which de facto represents the state-of-the-art/best-practices. Since it does not provide metrics suitable for the automatic assessment of software quality, we incorporated this international standard quality model in our compendium, selecting well known and automatically computable software quality metrics from literature, creating a double index referencing relevant metrics from the factors and criteria of the quality model, and referencing relevant criteria from the metrics (cf. Section 4).

Repeatability In general, this goal is reached if the complete data extraction, metrics computation and measurement process can be repeated in an efficient way and at will, leading to the same results for first and third parties. The criteria for this goal can be split into three sub-criteria: Automation, Reusability, and Maintainability.

Automation It is possible to extract data and compute metrics in an automated way. Thus, once defined sequences of data extractions and metrics computations can be applied in random order and frequency on the same or different software systems, leading to the expected results. The time spent on the initial definition is saved in consecutive uses, and user interaction is reduced to a minimum or not needed at all. It is possible to integrate the measurement

process as a fully automated part into the software development process.

This criterion has been fulfilled by implementing the metrics and the other parts of the description framework into the VizzAnalyzer tool (cf. Chapter 7). Therefore, it is possible to apply the defined metrics in random order and frequency on the same or different software systems, leading to the expected results. Through the scripting feature of the VizzAnalyzer (Section 7.1), the user interaction is either reduced to a minimum or not needed at all. The VizzAnalyzer can be integrated into a process and automatically executed, e.g., on each build of a software system.

Reuseability Metrics can be defined and implemented once in a generic way, supported by a meta-model. Briefly, a meta-model defines the rules and elements according to which models can be constructed, see Section 2.6. This makes the metrics applicable (and thereby reusable) to different software systems written in different programming languages. This means that, e.g., one and the same *Number Of Methods* metric (cf. Section 12.1), can be applied regardless if it is the number of methods in a class provided in Java Source Code or a class in a UML model. The metrics are defined once only, and can be applied to software systems represented in any existing or future software systems and languages, under the condition that the metric makes sense in that context at all.

This criterion has been fulfilled by defining metrics independently of language in a common meta-model (Chapter 5). The different languages are described by front-end specific meta-models (Section 5.2), which are mapped by language mappings (Section 5.2) to the common meta-model. Thus, once defined metrics can be reused for different languages, as long as they make sense. We showed this as example by defining a Java and a UML front-end (Section 5.2 and Section 6.3), and language-independent metric definitions (Section 12).

Maintainability Mechanisms are provided, which allow metrics to be defined and implemented once in a generic way, so that they keep up with changes and extensions in the meta-model without becoming invalidated. This means that metrics are so robust and flexible that they do not break if they encounter new, unknown elements and relations from new programming languages, and that these elements and relations are considered by the metrics if appropriated. E.g., a metric is defined upon a meta-model (Section 2.6)

containing classes, it is then extended to distinguish classes and interfaces. The metric just expecting classes must not break, if it finds an interface.

This criterion has been fulfilled by defining our architecture using front-end specific, common- and view-meta-models, hiding modifications to underlying meta-models through abstractions (Section 5.2), and by providing maintenance mechanisms which allow only safe changes to meta-models according to well-defined theorems (cf. Chapter 6).

9.2 Future work

This thesis has presented only some prerequisites for the solution to a complex problem. It has developed a definition framework for software quality metrics, and a *Compendium of Software Quality Standards and Metrics*. Yet, this presents only the necessary prerequisites including tool support for conducting the experiments outlined in the introduction (cf. Section 1.2). The detailed planning, conduction, and evaluation of the experiments still lie ahead, being the biggest part of the future work. This includes extraction of data from the different software systems, the software engineering processes, and other external quality indicators. We need to build an evaluation model and perform the evaluation. The progress shall be published in consecutive research papers and will finally lead to the dissertation.

Other future work includes publishing and maintaining the described compendium, thereby trying to make it more popular in the research community. We hope that we can soon provide some critical mass of valuable information to make the compendium interesting for other researchers, so that they start to contribute to it.

We are currently preparing for the analysis of the other projects of our project partners. We can already cover the Java based projects, but we also have projects having a code base written in C# and C/C++. We have been developing a C/C++ front-end utilizing the ROSE compiler infrastructure,[1] which we are about to integrate in our meta-model and tool. Since this front-end is currently only available for Linux, we plan to integrate a front-end based on the Eclipse C/C++ Development Tooling (CDT), which provides access to the C/C++ AST on other platforms than Linux.[2] Moreover, we

[1] ROSE is a compiler infrastructure which offers mechanisms for analyzing C/C++ source code, `http://www.llnl.gov/CASC/rose/`.

[2] The CDT is an industrial strength C/C++ IDE, `http://www.eclipse.org/cdt/`.

develop a C# front-end based on the *X-develop parser*.[3]

In parallel to these activities, we want to define and implement (our interpretation of) some more software quality metrics discussed in literature, thereby increasing the quantitative volume of the compendium. On the qualitative side, we intend to apply more advanced categorization over the software metrics, and we want to evaluate the current software quality model with some more advanced approaches, e.g., as proposed by Ortega et al. in [OPR03].

We also want to verify our conviction that independent implementations of the same metrics in different tools, with the current quality of metric definition, lead to incomparable results. Thus, we want to strengthen the indications we found within the OATA project, described in Section 8.1, where we could not compare the VizzAnalyzer and Ntools results. We want to select one or two open source software projects and a number of freely available metric tools, calculate the same metrics, and check if they get similar and comparable results.

Furthermore, we want to add a third dimension or level to the quality model, organizing the connection between the metrics and the criteria over quality attributes like cohesion, coupling, and complexity, which are then assessed by one or more metrics.

Some upcoming milestones are: Implement a C# front-end, evaluate C# projects, get process and external quality information, define an evaluation model, prepare and conduct the experiments.

[3]The *X-develop parser* is part of the *X-develop* multi-language cross-platform IDE offered by Omnicore, `http://www.omnicore.com/en/`. It allows to parse C#, Java, JSP, J#, VB.Net, ASP.Net, JavaScript and HTML through one API.

Part II

Compendium of Software Quality Standards and Metrics

Chapter 10

Introduction

Goal of the *Compendium of Software Quality Standards and Metrics* is to provide an information resource connecting software quality standards with well-known software metrics. Currently, the compendium describes

- 37 software quality attributes (factors, criteria), and
- 23 software quality metrics.

Please refer to the "How to use the Compendium", Section 10.1, to learn how to efficiently consult the compendium.

The `quality-metrics-compendium.pdf` is a printer-friendly version of the compendium. The `quality-metrics-matrix.pdf` gives an overview of the relations between quality properties and metrics.

This compendium is meant to be a "living document". Feel free to contribute with new metrics and comments, corrections, and add-ons to already defined ones. References to validating experiments or industrial experiences are especially welcome. Please refer to the *"How to use the Compendium"*, Section 10.1, to learn how to submit contributions.

10.1 How to use the Compendium

I want to consult the Compendium

The compendium has two main indices: *"Software Quality ISO Standards"*, (cf. Section 11) and *"Software Quality Metrics"* (cf. Section 12). The first index lists quality attributes and sub-attributes. Each attribute (factor) and sub-attribute (criterion) is first defined and then related to a number of metrics allowing its assessment. The second index lists quality metrics. Each metric is defined and then related to a number of quality attributes (factors) and sub-attributes (criteria) that it supports to assess.

I want to contribute to the Compendium

All contributions will be edited before publication. Please submit contributions to Rüdiger Lincke (`rudiger.lincke@arisa.se`).

If you want to describe a new metric, please use the description schema for metrics, cf. Section 12. If you want to comment, correct, or add to an already defined metric, please refer to the metric by its handle and indicate the items in the description schema for metrics, cf. Section 12, your contribution applies to. All other kinds of contributions are welcome as well.

10.2 History of the Compendium

2007-03 Rüdiger Lincke prepares the current version of the compendium for publication.

2007-03 Rüdiger Lincke edits and updates several sections of the compendium related to the Software Quality Model. Connections between metrics and criteria are updated.

2007-02 Rüdiger Lincke edits and updates several sections of the compendium related to metrics. The layout of metric sections is improved. Connections between new metrics and criteria in SQM are updated.

2006-04 through 2006-12 Rüdiger Lincke adapts existing metric definitions to new description schema, based on CMM10, adding metrics Ca 12.2, Ce 12.2, I 12.2, CF 12.2.

2005-09-27 through 2005-11-22 Rüdiger Lincke adds Number Of Children (12.2), Coupling Between Objects (12.2), Data Abstraction Coupling (12.2), Package Data Abstraction Coupling (12.2), Message Passing Coupling (12.2), Number Of local Methods (12.1), Change Dependency Between Classes (12.2), Locality of Data (12.2), Lack of COhesion in Methods (12.2), Improvement of LCOM (12.2), Tight Class Cohesion (12.2).

2005-09-26 Rüdiger Lincke adds Response For a Class 12.1

2005-08-18 Welf Löwe adds sub-classes for metrics and Lack of Documentation 12.3 metric.

2005-08-12 Welf Löwe adds Lines of Code 12.1, Number of Attributes and Methods 12.1, Depth of Inheritance Tree 12.2 metrics; also adds an alternative definition of McCabe Cyclomatic Complexity 12.1.

2005-08-03 Welf Löwe changes the initial classification of metrics. The structure of the compendium is adapted accordingly. The compendium moves to Latex and HTML and is published online.

2005-08-01 Rüdiger Lincke produces the first version of the compendium as Word document containing definitions of the Software Quality ISO Standards, cf. Section 11, McCabe Cyclomatic Complexity 12.1 and Weighted Method Count 12.1 as initial metrics.

2005-06-16 Rüdiger Lincke and Welf Löwe sketch goals and structure of the compendium.

Chapter 11

Software Quality ISO Standards

The ISO/IEC 9126 standard describes a software quality model which categorizes software quality into six characteristics (factors) which are sub-divided into sub-characteristics (criteria). The characteristics are manifested externally when the software is used as a consequence of internal software attributes.

The internal software attributes are measured by means of internal metrics (e.g., monitoring of software development before delivery). Examples of internal metrics are given in ISO 9126-3. The quality characteristics are measured externally by means of external metrics (e.g., evaluation of software products to be delivered). Examples of external metrics are given in ISO 9126-2.

Our work focuses on the assessment of the internal quality of a software product as it can be assessed upon the source code. In the following we describe the general quality model as defined in ISO 9126-1 with its properties and sub-properties. Details of the internal aspect of this model are given as well.

11.1 Functionality

The *functionality* characteristic allows to draw conclusions about how well software provides desired functions. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies functional requirements.

Suitability

The *suitability* sub-characteristic allows to draw conclusions about how suitable software is for a particular purpose. It correlates with metrics which measure attributes of software that allow to conclude the presence and appropriateness of a set of functions for specified tasks.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

Accuracy

The *accuracy* sub-characteristic allows to draw conclusions about how well software achieves correct or agreeable results. It correlates with metrics which measure attributes of software that allow to conclude about its provision of correct or agreeable results.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

Interoperability

The *interoperability* sub-characteristic allows to draw conclusions about how well software interacts with designated systems. It correlates with metrics which measure attributes of software that allow to conclude about its ability to interact with specified systems.

Highly Related Metrics

- none related, or not evaluated

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,

- Ca, Afferent Coupling 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2.

Security

The *security* sub-characteristic allows to draw conclusions about how secure software is. It correlates with metrics which measure attributes of software that allow to conclude about its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- Ca, Afferent Coupling 12.2,

- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2.

Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

11.2 Reliability

The *reliability* characteristic allows to draw conclusions about how well software maintains the level of system performance when used under specified conditions. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies reliability requirements.

Maturity

The *maturity* sub-characteristic allows to draw conclusions about how mature software is. It correlates with metrics which measure attributes of software that allow to conclude about the frequency of failure by faults in the software.

Highly Related Metrics

- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- NOC, Number Of Children 12.2,
- Lack Of Documentation (LOD) 12.3,

Fault-tolerance

The *fault-tolerance* sub-characteristic allows to draw conclusions about how fault-tolerant software is. It correlates with metrics which measure attributes of software that allow to conclude on its ability to maintain a specified level of performance in case of software faults or infringement of its specified interface.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,

- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- Ca, Afferent Coupling 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2.

Recoverability

The *recoverability* sub-characteristic allows to draw conclusions about how well software recovers from software faults or infringement of its specified interface. It correlates with metrics which measure attributes of software that allow to conclude on its ability to re-establish its level of performance and recover the data directly affected in case of a failure.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,

- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2.

Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

11.3 Usability

The *usability* characteristic allows to draw conclusions about how well software can be understood, learned, used and liked by the developer. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies usability requirements.

Remarks
See also Usability for Reuse 11.4.

Understandability

The *understandability* sub-characteristic allows to draw conclusions about how well users can recognize the logical concepts and applicability of software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for recognizing the logical concepts and applicability. Users should be able to select a software product which is suitable for their intended use.

Remarks
See also Understandability for Reuse 11.4.

Learnability

The *learnability* sub-characteristic allows to draw conclusions about how well users can learn the application of software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for learning the application of software.

Remarks
See also Learnability for Reuse 11.4.

Operability

The *operability* sub-characteristic allows to draw conclusions about how well users can operate software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for operation and operation control.

Remarks
See also Operability for Reuse – Programmability 11.4.

Attractiveness

The *attractiveness* sub-characteristic allows to draw conclusions about how attractive software is to the user. It correlates with metrics which measure attributes of software that allow to conclude on the capability of the software product to be attractive to the user.

Remarks
See also Attractiveness for Reuse – Programmability 11.4.

Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

Remarks
See also Compliance for Reuse – Programmability 11.4.

11.4 Re-Usability

Usability 11.3 is defined to be the characteristic that allows to draw conclusions about how well software can be understood, learned, used and liked by the developer.

Remarks
Re-Usability is a special case of the quality factor Usability 11.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as usability of software for integration in other systems, e.g., usability of libraries and components. Here the user is a software engineer/developer.

Hence, internal usability metrics are used for predicting the extend of which the software in question can be understood, learned, operated, is attractive and compliant with usability regulations and guidelines *where here using means integrating it in a larger software system.*

Understandability for Reuse

Understandability 11.3 is defined as the attributes of software that bear on the users' efforts for recognizing the logical concept and its applicability.

Remarks
Understandability for Reuse is a special case of the quality criterium Understandability 11.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as understandability of software for a software engineer/developer with the purpose of integrating it in new systems, e.g., understandability of libraries and components.

Hence, software engineers/developers should be able to select a software product which is suitable for their intended use. Internal understandability

reuse metrics assess whether new software engineers/developers can understand: whether the software is suitable; how it can be used for particular tasks.

Highly Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- NOC, Number Of Children 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2.
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- either not related, or not evaluated

Learnability for Reuse

Learnability 11.3 is defined as the attributes of software that bear on the users' efforts for learning its application.

Remarks
Learnability for Reuse is a special case of the quality criterium Learnability 11.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as learnability of software for a software engineer or developer with the purpose of integrating it in new systems, e.g., learnability of libraries and components.

Hence, internal learnability metrics assess how long software engineers or developers take to learn how to use particular functions, and the effectiveness of documentation. Learnability is strongly related to understandability, and understandability measurements can be indicators of the learnability potential of the software.

Highly Related Metrics

- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- NOC, Number Of Children 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- LOC, Lines of Code 12.1,
- RFC, Response For a Class 12.1,
- Ca, Afferent Coupling 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,

- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Operability for Reuse – Programmability

Operability 11.3 is defined as the attributes of software that bear on the users' efforts for operation and operation control.

Remarks
Operability for Reuse is a special case of the quality criterium Operability 11.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as programmability of software, i.e., the development effort for a software engineer/developer to integrate it in new systems, e.g., programmability software using libraries and components.

Hence, internal programmability metrics assess whether software engineers/developers can integrate and control the software.

General operability metrics can be categorized by the dialog principles in ISO 9241-10: suitability of the software for the task; self-descriptiveness of the software; controllability of the software; conformity of the software with user expectations; error tolerance of the software; suitability of the software for individualization. This also applies to operability in a reuse context.

Highly Related Metrics

- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- NOC, Number Of Children 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- LOC, Lines of Code 12.1,
- RFC, Response For a Class 12.1,
- Ca, Afferent Coupling 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Attractiveness for Reuse

Attractiveness 11.3 is defined as the attributes of software that bear on the capability of the software product to be attractive to the user.

Remarks
Attractiveness for Reuse is a special case of the quality criterium Attractiveness 11.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as attractiveness of software for a software engineer /developer to integrating it in new systems, e.g., attractiveness of libraries and components.

Internal attractiveness metrics assess the appearance of the software. In a reuse context, they will be influenced by factors such as code and documentation layout. In addition to appearance, internal attractiveness metrics for reuse also assess size and complexity of the reused code.

Highly Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- NOC, Number Of Children 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,

- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2.
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Related Metrics

- Lack Of Documentation (LOD) 12.3.

Compliance for Reuse

Compliance 11.3 is defined as the attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions.

Remarks
Compliance for Reuse is a special case of the quality criterium Compliance 11.3 and not specifically mentioned in the ISO 9126-3 standard. It can be understood as compliance to re-use standards or conventions or regulations in laws and similar prescriptions.

Internal compliance metrics assess adherence to standards, conventions, style guides or regulations relating to re-usability.

11.5 Efficiency

The *efficiency* characteristic allows to draw conclusions about how well software provides required performance relative to amount of resources used. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies efficiency requirements.

Time behavior

The *time behavior* sub-characteristic allows to draw conclusions about how well the time behavior of software is for a particular purpose. It correlates with metrics which measure attributes of software that allow to conclude about the time behavior of the software (or parts of it) in combination with the computer system during testing or operating.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Resource utilization

The *resource utilization* sub-characteristic allows to draw conclusions about the amount of resources utilized by the software. It correlates with metrics which measure attributes of software that allow to conclude about the amount and duration of resources used while performing its function.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

11.6 Maintainability

The *maintainability* characteristic allows to draw conclusions about how well software can be maintained. It can be used for assessing, controlling and predicting the effort needed to modify the software product (or parts of it) in question.

Analyzability

The *analyzability* sub-characteristic allows to draw conclusions about how well software can be analyzed. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.

Highly Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,

- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- NOC, Number Of Children 12.2.

Changeability

The *changeability* sub-characteristic allows to draw conclusions about how well software can be changed. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for modification, fault removal or for environmental change.

Highly Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,

- NOC, Number Of Children 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- either not related, or not evaluated

Stability

The *stability* sub-characteristic allows to draw conclusions about how stable software is. It correlates with metrics which measure attributes of software that allow to conclude about the risk of unexpected effects as result of modifications.

Highly Related Metrics

- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- NOC, Number Of Children 12.2,
- Ca, Afferent Coupling 12.2,
- Lack Of Documentation (LOD) 12.3.

Testability

The *testability* sub-characteristic allows to draw conclusions about how well software can be tested and is tested. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for validating the software and about the test coverage.

Highly Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- NOC, Number Of Children 12.2.

Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

11.7 Portability

The *portability* characteristic allows to draw conclusions about how well software can be ported from one environment to another. It can be used for assessing, controlling and predicting the extend to which the software product (or parts of it) in question satisfies portability requirements.

Adaptability

The *adaptability* sub-characteristic allows to draw conclusions about how well software can be adapted to environmental change. It correlates with metrics which measure attributes of software that allow to conclude about the amount of changes needed for the adaptation of software to different specified environments.

Highly Related Metrics

- LOC, Lines of Code 12.1,
- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- RFC, Response For a Class 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- CBO, Coupling Between Objects (CBO) 12.2,

- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- NOC, Number Of Children 12.2.

Installability

The *installability* sub-characteristic allows to draw conclusions about how well software can be installed in a designated environment. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed to install the software in a specified environment.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

Co-existence

The *co-existence* sub-characteristic allows to draw conclusions about how well software can co-exist with other software products in the same operational environment. It correlates with metrics which measure attributes of software that allow to conclude about the dependencies, concurrency behavior, or side effects.

Highly Related Metrics

- either not related, or not evaluated

Related Metrics

- either not related, or not evaluated

Replaceablity

The *replaceability* sub-characteristic allows to draw conclusions about how well software can replace other software or parts of it. It correlates with metrics which measure attributes of software that allow to conclude about opportunity and effort using it instead of specified other software in the environment of that software.

Highly Related Metrics

- SIZE2, Number of Attributes and Methods 12.1,
- NOM, Number of local Methods 12.1,
- CC, McCabe Cyclomatic Complexity 12.1,
- WMC, Weighted Method Count 12.1,
- DIT, Depth of Inheritance Tree 12.2,
- NOC, Number Of Children 12.2,
- Ca, Afferent Coupling 12.2,
- Lack Of Documentation (LOD) 12.3.

Related Metrics

- LOC, Lines of Code 12.1,
- RFC, Response For a Class 12.1,
- CBO, Coupling Between Objects (CBO) 12.2,
- Change Dependency Between Classes (CDBC) 12.2,
- Change Dependency Of Classes (CDOC) 12.2,
- Efferent Coupling (Ce) 12.2,
- Coupling Factor (CF) 12.2,
- Data Abstraction Coupling (DAC) 12.2,
- Instability (I) 12.2,
- Locality of Data (LD) 12.2,
- Message Passing Coupling (MPC) 12.2,
- Package Data Abstraction Coupling (PDAC) 12.2,
- Lack of Cohesion in Methods (LCOM) 12.2,
- Improvement of LCOM (ILCOM) 12.2,
- Tight Class Cohesion (TCC) 12.2.

Compliance

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

Chapter 12

Software Quality Metrics

This part of the document defines standard metrics reported in literature. Moreover, it discusses the applicability of each metric in one of the ISO 9126-3:2003 (factors/criteria) properties/sub properties, Section 11.

Metrics are classified as supporting mainly statements on software *Complexity* (cf. Section 12.1), software *Architecture and Structure* (cf. Section 12.2), and software *Design and Coding* (cf. Section 12.3).

We use the following description schema for each metric. It defines:

Handle A short name or symbol uniquely identifying the metric. This is the way it is referenced in this document and literature. If there are more than one variation of a metric, additional symbols will be added to distinguish the different variants.

Description A textual description of the metric and what it measures. This description is a summary of how the metric is described in literature. Since it is described in natural language, it is not suitable for an unambiguous definition of the metric.

Scope Identifies the program element the metric applies to. This could be *system* for the whole software system, *package/directory* for entities of a package or a directory, *class/module/file* for classes, modules and files, respectively, and *procedure/function/method* for procedures, functions and methods, respectively. Only the core scope is named.

Definition A formal definition of the metric using set theory and mathematics. The definition is based on the meta-model specified in the compendium. Usually a view is defined as well, see below.

View Definition of the *view*, describing its relevant elements, semantic relations, and the tree grammar.

Scale Defines the scale type of the metric. It is one of *Absolute*, *Rational*, *Interval*, *Ordinal*, and *Nominal*.

Domain Defines the domain of metric values.

Highly Related Software Quality Properties Discusses factors and criteria as listed in the ISO 9126-3:2003 standard that are assessable with the metric. See Section 4.2 for an overview, and Part II for details.

Related Software Quality Properties Discusses factors and criteria as listed in the ISO 9126-3:2003 standard that are somewhat assessable with the metric. See Section 4.2 for an overview, and Part II for details.

References Gives references to literature, projects, repositories, and programs using, implementing, or describing the metric. Moreover, it points to validating and evaluating experiments investigating its appropriateness.

Since Since which version of the compendium is the metric contained.

The definition of the metrics is based on the common meta-model as defined in Section 5.2. For each metric, we define a *view* further abstracting from the common meta-model to provide exactly the information required by that metric. The view is used for the actual metric definition. This approach makes the metric definitions independent from changes of the common meta-model (this is discussed in more detail in Chapter 6). Formally, views of a metric analysis A are defined as pairs $V^A = \left(G^A, R^A\right)$ and are bound to the common model with a mapping specification α^A. Again, G^A is a tree grammar specifying the set of view entities and their structural containment required by A. R^A is a set of semantic relations over view entities required by A.

The construction of concrete view models follows the same principles as the abstractions from front-end specific to common models: we ignore some common meta-model entity types, which leads to a filtering of the corresponding nodes. We propagate relevant descendants of filtered nodes to their relevant ancestors by adding them as direct children. Moreover, we ignore some relation types, and attach remaining relations defined over filtered nodes to the relevant ancestors of those nodes, as described in detail in [SLLL06].

To simplify metric definitions, we define some utility operation(s):

$succ(e, r)$ $[succ^*(e, r), succ^+(e, r)]$ – denotes the set of direct [transitive, transitive excluding e] successors of a node or set of nodes e over edges (relations) of type r.

$|S|$ – denotes the number of elements in a set S.

$pred(e, r)$ $[pred^*(e, r), pred^+(e, r)]$ – denotes the set of direct [transitive, transitive excluding e] predecessors of a node or set of nodes e over edges (relations) of type r.

$dist(e, e', r)$ – denotes the distance or set of distances (number of relations) between a node e and a node or set of nodes e'. It is 0, if it e equals e', and ∞, if there is no path between e and e'.

$max(S)$ – denotes the maximum in a set S of values.

$min(S)$ – denotes the minimum in a set S of values.

$scc(e, r)$ – denotes the set of strongly connected components in a set e of elements over edges (relations) of type r.

$outdegree(e, r)$ $[outdegree^+(e, r)]$ – denotes the number of outgoing relations of type r from a node e including [excluding] relations to e itself.

For additional simplification, we define some frequently used sets:

$M(c) = pred(c, \text{methodof})$ – denotes the set of all methods locally declared in a class c, ignoring visibility and other attributes.

$F(c) = pred(c, \text{fieldof})$ – denotes the set of all fields locally declared in a class c, ignoring visibility and other attributes.

12.1 Complexity

Complexity metrics refer to the static, i.e., structural size and complexity of software. They only indirectly measure execution complexity (if at all).

Size

Lines of Code (LOC) Works only with instances of the common metamodel which were created by a front-end calculating this metric and attaching its value as property to the specific elements. Currently only the Eclipse Java Front-end supports this, the UML Front-end can of course not calculate this metric, since there is no source code in UML.

Handle *LOC*

Description Lines of code simply counts the lines of source code (line break characters) of a certain software entity. It is a simple yet powerful metric to assess the complexity of software entities. Since it is depending on code conventions and format, it is critical to use it in generated codes since it may lack of line breaks. Additionally it can only be measured in the source code itself from the front-end and is therefore a front-end side metric.

Scope Method, Class, Compilation unit

View $V^{LOC} = (G^{LOC}, R^{LOC})$

- Grammar $G^{LOC} = (\{\text{scope}^{LOC}\}, \emptyset, \text{scope}^{LOC})$
- Relations $R^{LOC} : \emptyset$
- Mapping α^{LOC}:

$$\begin{aligned} \alpha^{LOC}(\text{Class}) &\mapsto \text{scope}^{LOC} \\ \alpha^{LOC}(\text{Method}) &\mapsto \text{scope}^{LOC} \\ \alpha^{LOC}(\text{CompilationUnit}) &\mapsto \text{scope}^{LOC} \end{aligned}$$

Definition The LOC value of a element $s \in scope^{LOC}$ is defined as:

$$LOC(s) = s.loc$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is both negatively and positively influenced by size.

Understandability for Reuse 11.4: Understanding whether a software entity E is suitable for reuse, or not, depends on its size.
Understandability declines with increasing LOC.

Attractiveness 11.4: Attractiveness of a software entity E depends on the size of the potentially reused code.
Attractiveness increases with increasing LOC.

Maintainability 11.6 declines with increasing LOC.

Analyzability 11.6: The effort and time for diagnosis of deficiencies or causes of failures in a software entity, or for identification of parts to be modified is directly related to its size.
Analyzability declines with increasing LOC.

Changeability 11.6: Changing a software entity requires prior understanding, which, in turn, is more complicated for large systems.
Changeability declines with increasing LOC.

Testability 11.6: Complete testing requires coverage of all execution paths. The number of possible execution paths of a software entity increases with its size.
Testability declines with increasing LOC.

Portability 11.7 declines with increasing LOC.

Adaptability 11.7: As for changeability 11.6, the size of a software entity has a direct impact. Each modification requires understanding which is more complicated for large systems.
Adaptability declines with increasing LOC.

Related Software Quality Properties

Functionality 11.1 might increase with increasing LOC.

Interoperability 11.1: Interoperability requires to be able to locate the parts of a system responsible for interoperability. The size of these parts might indicate a better ability to interact.
Interoperability might increase with increasing LOC.

Security 11.1: Relating LOC to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing LOC.

Reliability 11.2 might increase with increasing LOC.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs might be left in the software. Therefore, also maturity may be influenced negatively by code size.
Maturity might decline with increasing LOC.

Fault-tolerance 11.2: Relating LOC to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing LOC.

Recoverability 11.2: Relating LOC to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase with increasing LOC.

Re-Usability 11.4 might decrease with increasing LOC.

Learnability for Reuse 11.4: Learning if a software entity is suitable for reuse depends on the size and complexity of its interface. LOC is a general size metric, but not specifically assessing interface size and complexity.
Learnability might decline with increasing LOC.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends the complexity of its interface. LOC is a general size metric, but not specifically assessing interface size and complexity.
Programmability might decline with increasing LOC.

Efficiency 11.5 might decline with increasing LOC.

Time Behavior 11.5: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc.
Time behavior might get worse with increasing LOC.

Resource Utilization 11.5: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing LOC.

Maintainability 11.6 declines with increasing LOC.

Stability 11.6: Due to reduced analyzability 11.6 and testability 11.6, also stability may be influenced negatively by size.
Stability might decline with increasing LOC.

Portability 11.7 declines with increasing LOC.

Replaceablity 11.7: The substitute of a component must imitate its observable behavior. Large components have the potential of a more complex observable behavior making it more difficult to check substitutability and to actually substitute a component. Interface size and complexity, an attribute directly connected to replaceability, is not specifically assessed by LOC.
Replaceablity might declines with increasing LOC.

References

- LOC is extensively discussed in [Hum97, p22], [BBC+99, p22],
- and evaluated in a case study [PLLL05],
- it is implemented in the VizzAnalyzer Metrics Suite.

Interface Complexity

Number of Attributes and Methods ($SIZE2$) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the $SIZE2$ specific view).

Handle $SIZE2$

Description Number of Attributes and Methods simply counts the number of attributes and methods of a class. It is an object-oriented metric that can be applied to modular languages by considering the number of variables (globally visible in a module) and its number of functions and procedures.

Scope Class

View $V^{SIZE2} = (G^{SIZE2}, R^{SIZE2})$

- Grammar $G^{SIZE2} = (\{\text{class}^{SIZE2}, \text{method}^{SIZE2}, \text{attribute}^{SIZE2}\}, \emptyset, \text{class}^{SIZE2})$
- Relations $R^{SIZE2} : \{\text{attributeof}^{SIZE2} : \text{attribute}^{SIZE2} \times \text{class}^{SIZE2}, \text{methodof}^{SIZE2} : \text{method}^{SIZE2} \times \text{class}^{SIZE2}\}$
- Mapping α^{SIZE2}:

$$\begin{aligned} \alpha^{SIZE2}(\text{Class}) &\mapsto \text{class}^{SIZE2} \\ \alpha^{SIZE2}(\text{IsMethodOf}) &\mapsto \text{methodof}^{SIZE2} \\ \alpha^{SIZE2}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{SIZE2} \\ \alpha^{SIZE2}(\text{Method}) &\mapsto \text{method}^{SIZE2} \\ \alpha^{SIZE2}(\text{Field}) &\mapsto \text{attribute}^{SIZE2} \end{aligned}$$

Definition The $SIZE2$ value of a class $c \in class^{SIZE2}$ is defined as:

$$\begin{aligned} A(c) &= pred(c, \text{attributeof}^{SIZE2}) \\ &\quad \textit{-- set of attributes directly contained in } c \\ M(c) &= pred(c, \text{methodof}^{SIZE2}) \\ &\quad \textit{-- set of methods directly contained in } c \\ SIZE2(c) &= |A(c) \cup M(c)| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is both negatively and positively influenced by SIZE2.

Understandability for Reuse 11.4: Understanding if a class is suitable for reuse depends on its size.
Understandability declines with increasing SIZE2.

Learnability for Reuse 11.4: Learning if a class is suitable for reuse depends on the size and complexity of its interface. SIZE2 measures interface size.
Learnability might decline with increasing SIZE2.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends the complexity of its interface. SIZE2 measures interface size.
Programmability might decline with increasing SIZE2.

Attractiveness 11.4: Attractiveness of a class depends on the size of the potentially reused code.
Attractiveness increases with increasing SIZE2.

Maintainability 11.6 declines with increasing SIZE2.

Analyzability 11.6: The effort and time for diagnosis of deficiencies or causes of failures in software entity, or for identification of parts to be modified is directly related to its size.
Analyzability declines with increasing SIZE2.

Changeability 11.6: Changing a class requires prior understanding, which, in turn, is more complicated for large systems.
Changeability declines with increasing SIZE2.

Testability 11.6: Complete testing requires coverage of all execution paths. The number of possible execution paths of a system increases with its size.
Testability declines with increasing SIZE2.

Portability 11.7 declines with increasing SIZE2.

Adaptability 11.7: As for changeability 11.6, the size of software has a direct impact. Each modification requires understanding which is more complicated for large systems.
Adaptability declines with increasing SIZE2.

Replaceablity 11.7: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size is specifically assessed by SIZE2.
Replaceablity decline with increasing SIZE2.

Related Software Quality Properties

Functionality 11.1 might increase with increasing SIZE2.

Interoperability 11.1: Interoperability requires to be able to locate the parts of a system responsible for interoperability. The size of in these parts might indicate a better ability to interact.
Interoperability might increase with increasing SIZE2.

Security 11.1: Relating SIZE2 to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing SIZE2.

Reliability 11.2 might increase with increasing SIZE2.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by interface size.
Maturity might decline with increasing SIZE2.

Fault-tolerance 11.2: Relating SIZE2 to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing SIZE2.

Recoverability 11.2: Relating SIZE2 to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase with increasing SIZE2.

Efficiency 11.5 might decline with increasing SIZE2.

Time Behavior 11.5: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc.
Time behavior might get worse with increasing SIZE2.

Resource Utilization 11.5: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing SIZE2.

Maintainability 11.6 declines with increasing SIZE2.

Stability 11.6: Due to reduced analyzability 11.6 and testability 11.6, also stability may be influenced negatively by size. Stability might decline with increasing SIZE2.

References

- The SIZE2 metric is extensively discussed in [LH93a, BBC+99, BDW99, HM95],
- SIZE2 is implemented in the VizzAnalyzer Metrics Suite.

Number Of local Methods (NOM) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the *NOM* specific view).

Handle *NOM*

Description Number of local Methods measures the number of methods locally declared in a class. Inherited methods are not considered. It is the size of the interface of a class and allows conclusions on its complexity.

Scope Class

View $V^{NOM} = (G^{NOM}, R^{NOM})$

- Grammar $G^{NOM} = (\{\text{class}^{NOM}, \text{method}^{NOM}\}, \emptyset, \text{class}^{NOM})$
- Relations $R^{NOM} : \{\text{methodof}^{NOM} : \text{method}^{NOM} \times \text{class}^{NOM}\}$
- Mapping α^{NOM}:

$$\begin{aligned} \alpha^{NOM}(\text{Class}) &\mapsto \text{class}^{NOM} \\ \alpha^{NOM}(\text{IsMethodOf}) &\mapsto \text{methodof}^{NOM} \\ \alpha^{NOM}(\text{Method}) &\mapsto \text{method}^{NOM} \end{aligned}$$

Definition The *NOM* value of a class $c \in class^{NOM}$ is defined as:

$$\begin{aligned} M(c) &= pred(c, \text{methodof}^{NOM}) \\ &\quad \textit{-- set of methods directly contained in } c \\ NOM(c) &= |M(c)| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is both negatively and positively influenced by NOM.

Understandability for Reuse 11.4: Understanding if a class is suitable for reuse depends on its size.
Understandability declines with increasing NOM.

Learnability for Reuse 11.4: Learning if a class is suitable for reuse depends on the size and complexity of its interface. NOM measures interface size.
Learnability might decline with increasing NOM.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends the complexity of its interface. NOM measures interface size.
Programmability might decline with increasing NOM.

Attractiveness 11.4: Attractiveness of a class depends on the size of the potentially reused code.
Attractiveness increases with increasing NOM.

Maintainability 11.6 declines with increasing NOM.

Analyzability 11.6: The effort and time for diagnosis of deficiencies or causes of failures in software entity, or for identification of parts to be modified is directly related to its size.
Analyzability declines with increasing NOM.

Changeability 11.6: Changing a class requires prior understanding, which, in turn, is more complicated for large systems.
Changeability declines with increasing NOM.

Testability 11.6: Complete testing requires coverage of all execution paths. The number of possible execution paths of a system increases with its size.
Testability declines with increasing NOM.

Portability 11.7 declines with increasing NOM.

Adaptability 11.7: As for changeability 11.6, the size of software has a direct impact. Each modification requires understanding which is more complicated for large systems.
Adaptability declines with increasing NOM.

Replaceablity 11.7: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size is specifically assessed by NOM.
Replaceablity decline with increasing NOM.

Related Software Quality Properties

Functionality 11.1 might increase with increasing NOM.

Interoperability 11.1: Interoperability requires to be able to locate the parts of a system responsible for interoperability. The size of in these parts might indicate a better ability to interact.
Interoperability might increase with increasing NOM.

Security 11.1: Relating NOM to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing NOM.

Reliability 11.2 might increase with increasing NOM.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by interface size.
Maturity might decline with increasing NOM.

Fault-tolerance 11.2: Relating NOM to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing NOM.

Recoverability 11.2: Relating NOM to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase with increasing NOM.

Efficiency 11.5 might decline with increasing NOM.

Time Behavior 11.5: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc.
Time behavior might get worse with increasing NOM.

Resource Utilization 11.5: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing NOM.

Maintainability 11.6 declines with increasing NOM.

Stability 11.6: Due to reduced analyzability 11.6 and testability 11.6, also stability may be influenced negatively by size. Stability might decline with increasing NOM.

References

- The NOM metric is extensively discussed in [LH93a, BBC+99, BDW99, HM95],
- and evaluated in a case study [PLLL05],
- NOM is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Structural Complexity

McCabe Cyclomatic Complexity (CC) *CC* can be computed on instances of a common meta-model, as long as the required types are provided by the front-end. A UML front-end, e.g., would not construct nodes of required type. Still, this front-end works with the new meta-model and the metric is unambiguously defined.

Description *CC* is a measure of the control structure complexity of software. It is the number of linearly independent paths and therefore, the minimum number of independent paths when executing the software.

Scope Method

View $V^{CC} = (G^{CC}, \emptyset)$.

- Grammar $G^{CC} = (T^{CC}, P^{CC}, \text{method}^{CC})$
- Entities $T^{CC} = \{\text{method}^{CC}, \text{ctrl\_stmt}^{CC}\}$
- Productions P^{CC}:

$$\begin{aligned} \text{method}^{CC} &= \text{ctrl\_stmt}^{CC\,*} \\ \text{ctrl\_stmt}^{CC} &= \text{ctrl\_stmt}^{CC\,*} \end{aligned}$$

- Mapping α^{CC}:

$$\begin{aligned} \alpha^{CC}(\text{Method}) &\mapsto \mathit{method}^{CC} \\ \alpha^{CC}(\text{Switch}) &\mapsto \text{ctrl\_stmt}^{CC} \\ \alpha^{CC}(\text{Loop}) &\mapsto \text{ctrl\_stmt}^{CC} \end{aligned}$$

Definition The CC value of a method $m \in method^{CC}$ is defined as:

$$CC(m) := \left| succ^{+}\left(m, \text{contains}^{CC}\right)\right| + 1$$

Scale Absolute.

Domain Integers in $1..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is both negatively and positively influenced by attributed assess with Cyclomatic Complexity.

Understandability for Reuse 11.4: Understanding if a class is suitable for reuse depends on its complexity.
Understandability declines with increasing Cyclomatic Complexity.

Learnability for Reuse 11.4: Learning if a class is suitable for reuse depends on the complexity of its interface. Systems with high control complexity may also have a complex (behavioral) interface.
Learnability declines with increasing Cyclomatic Complexity.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends the complexity of its interface. Systems with high control complexity may also have a complex (behavioral) interface.
Programmability declines with increasing Cyclomatic Complexity.

Attractiveness 11.4: Attractiveness of a class depends on the complexity of the potentially reused code. Cyclomatic Complexity allows an assessment complexity.
Attractiveness increases with increasing Cyclomatic Complexity.

Maintainability 11.6 declines with increasing Cyclomatic Complexity.

Analyzability 11.6: The effort and time for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified is directly related to the number of execution paths, i.e. the complexity of the control flow.
Analyzability declines with increasing Cyclomatic Complexity.

Changeability 11.6: Each modification must be correct for all execution paths. Cyclomatic Complexity computes the number of the linearly independent paths, a lower bound of all execution paths ignoring multiple iterations.
Changeability declines with increasing Cyclomatic Complexity.

Testability 11.6: Complete testing requires coverage of all execution paths. Cyclomatic Complexity computes the number of the linearly independent paths, a lower bound of all execution paths ignoring multiple iterations.
Testability declines with increasing Cyclomatic Complexity.

Portability 11.7 declines with increasing Cyclomatic Complexity.

Adaptability 11.7: As for changeability 11.6, the complexity the control structure of software has a direct impact. Each modification must be correct for all execution paths. Cyclomatic Complexity computes the number of the linearly independent paths, a lower bound of all execution paths ignoring multiple iterations.
Adaptability declines with increasing Cyclomatic Complexity.

Replaceablity 11.7: The substitute of a component must imitate its observable behavior. Components with complex control structures might have a more complex observable behavior making it more difficult to check substitutability and to actually substitute a component.
Replaceablity declines with increasing Cyclomatic Complexity.

Related Software Quality Properties

Functionality 11.1 might increase with increasing Cyclomatic Complexity.

Interoperability 11.1: Relating Cyclomatic Complexity to interoperability requires to be able to locate the parts of a system responsible for interoperability. Complexity in these parts might indicate a better ability to interact.
Interoperability might increase with increasing CC.

Security 11.1: Relating Cyclomatic Complexity to security requires to be able to locate the parts of a system responsible for security. Complexity in these parts might indicate a higher security.

Security might increase with increasing Cyclomatic Complexity.

Reliability 11.2 might increase with increasing Cyclomatic Complexity.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by control-flow complexity.
Maturity might decline with increasing Cyclomatic Complexity.

Fault-tolerance 11.2: Relating Cyclomatic Complexity to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. Complexity in these parts might indicate a better ability to interact.
Fault-Tolerance might increase with increasing CC.

Recoverability 11.2: Relating Cyclomatic Complexity to recoverability requires to be able to locate the parts of a system responsible for recoverability. Complexity in these parts might indicate a higher recoverability.
Recoverability might increase with increasing CC.

Efficiency 11.5 might decline with increasing Cyclomatic Complexity.

Time Behavior 11.5: Static complexity might indicate a higher execution complexity.
Time behavior might get worse with increasing Cyclomatic Complexity.

Maintainability 11.6 declines with increasing Cyclomatic Complexity.

Stability 11.6: Due to reduced analyzability 11.6 and testability 11.6, also stability may be influenced negatively by control-flow complexity.
Stability might decline with increasing Cyclomatic Complexity.

References

- CC is extensively discussed in [WM96],
- it is used to assess design and structural complexity of a system in [MB89],

- Li and Henry [LH93b] propose to use Cyclomatic Complexity 12.1 as weight measure in Weighted Method Count 12.1,
- it is mentioned in the following resources:
  - `http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html`
  - `http://www.mccabe.com/iq_research_metrics.htm`
  - `http://mdp.ivv.nasa.gov/repository.html`
- CC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Weighted Method Count (WMC***)*** Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the WMC specific view).

Handle WMC

Description A weighted sum of methods implemented within a class. It is parameterized by a way to compute the weight of each method. Possible weight metrics are:

- McCabe Cyclomatic Complexity 12.1,
- Lines of Code 12.1,
- 1 (unweighted WMC).

This variant of WMC uses McCabe Cyclomatic Complexity 12.1 metric for calculating the weight for each method. Originally defined as an object-oriented metric, it can easily adapted to non-object-oriented systems computing the weighted sum of functions implemented within a module or file.

Scope Class

View $V^{WMC} = (G^{WMC}, R^{WMC})$

- Grammar $G^{WMC} = (\{\text{class}^{WMC}, \text{method}^{WMC}\}, \emptyset, \text{class}^{WMC})$
- Relations $R^{WMC} : \{\text{methodof}^{WMC} : \text{method}^{WMC} \times \text{class}^{WMC}\}$

- Mapping α^{WMC}:

$$\begin{aligned} \alpha^{WMC}(\text{Class}) &\mapsto \text{class}^{WMC} \\ \alpha^{WMC}(\text{IsMethodOf}) &\mapsto \text{methodof}^{WMC} \\ \alpha^{WMC}(\text{Method}) &\mapsto \text{method}^{WMC} \end{aligned}$$

Definition The WMC value of a class $c \in class^{WMC}$ is defined as:

$$\begin{aligned} M(c) &= pred(c, \text{methodof}^{WMC}) \\ &\quad \textit{-- set of methods directly contained in } c \\ WMC(c) &= \sum_{m \in M(c)} CC(m) \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 depends on the weight metric, which influences the attributes assessed with WMC negatively or positively.

Re-Usability declines with increasing unweighted WMC.

Understandability for Reuse 11.4: Understanding if a class is suitable for reuse depends on the size of its interface.
Understandability declines with increasing unweighted WMC.

Learnability for Reuse 11.4: Learning if a class is suitable for reuse depends on the size of its interface.
Learnability declines with increasing unweighted WMC.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends on the size of its interface.
Programmability declines with increasing unweighted WMC.

Attractiveness 11.4: Attractiveness of a class depends on the size and complexity of the potentially reused code. Depending on the weight metric, WMC allows an assessment of size or complexity.
Attractiveness increases with increasing unweighted WMC.

Maintainability 11.6 declines with increasing WMC.

Analyzability 11.6: The effort and time for diagnosis of deficiencies or causes of failures in a class, or for identification of

parts to be modified is directly related to the size and complexity of the class. Depending on the weight metric, WMC allows an assessment of size or complexity.
Analyzability declines with increasing unweighted WMC.

Changeability 11.6: Changing a class requires prior understanding, which, in turn, is more complicated for large and complex systems. Depending on the weight metric, WMC allows an assessment of size or complexity.
Changeability declines with increasing unweighted WMC.

Testability 11.6: Complete testing requires coverage of all execution paths. The number of possible execution paths of a class increases with the number of methods and their control flow complexity. Depending on the weight metric, WMC allows an assessment of the number of methods and their complexity.
Testability declines with increasing unweighted WMC.

Portability 11.7 declines with increasing WMC.

Adaptability 11.7: As for changeability 11.6, the size of software has a direct impact. Each modification requires understanding which is more complicated for large systems. Size is specifically assessed by the weighted versions of WMC.
Adaptability declines with increasing unweighted WMC.

Replaceablity 11.7: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size is specifically assessed by the unweighted WMC.
Replaceablity decline with increasing unweighted WMC.

Related Software Quality Properties

Functionality 11.1 might increase with increasing WMC.

Interoperability 11.1: Interoperability requires to be able to locate the parts of a system responsible for interoperability. The size of in these parts might indicate a better ability to interact. Interoperability might increase if the unweighted WMC increases.

Security 11.1: Relating WMC to security requires to be able to locate the parts of a system responsible for security. The size of these parts might indicate a higher security.
Security might increase with increasing unweighted WMC.

Reliability 11.2 might increase with increasing WMC.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by WMC.
Maturity might decline with increasing unweighted WMC.

Fault-tolerance 11.2: Relating WMC to fault-tolerance requires to be able to locate the parts of a system responsible for fault-tolerance. The size of these parts might indicate a better ability to interact.
Fault-tolerance might increase if the unweighted WMC increases.

Recoverability 11.2: Relating WMC to recoverability requires to be able to locate the parts of a system responsible for recoverability. The size of these parts might indicate a higher recoverability.
Recoverability might increase if the unweighted WMC increases.

Efficiency 11.5 might decline with increasing WMC.

Time Behavior 11.5: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc.
Time behavior might get worse with increasing unweighted WMC.

Resource Utilization 11.5: Static size might indicate a higher memory utilization.
Resource utilization might get worse if the unweighted WMC increases.

Maintainability 11.6 declines with increasing WMC.

Stability 11.6: Due to reduced analyzability 11.6 and testability 11.6, also stability may be influenced negatively by size.
Stability might decline with increasing unweighted WMC.

References

- WMC is extensively discussed in [CK91, CK94, LH93a, SK03, BK95, BBC+99, Bal96, CS95, HM95, HM96, MH99a, BDW99, HCN98b],
- Li and Henry [LH93b] propose to weight the methods according to Cyclomatic Complexity 12.1,

- it is evaluated in a case study [PLLL05],
- WMC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Response For a Class (RFC) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective call (Java) or message (UML) relations expressing the messages sent between two classes are stored in the CMM10 as Invokes, and are mapped onto relations of type call^{RFC} in the common meta-model (and the *RFC* specific view).

Handle *RFC*

Description Count of (public) methods in a class and methods directly called by these. *RFC* is only applicable to object-oriented systems.

Scope Class

View $V^{RFC} = (G^{RFC}, R^{RFC})$

- Grammar $G^{RFC} = (\{\text{class}^{RFC}, \text{method}^{RFC}\}, \emptyset, \text{class}^{RFC})$
- Relations $R^{RFC} : \{\text{contains}^{RFC} : \text{class}^{RFC} \times \text{method}^{RFC},$
  $\text{call}^{RFC} : \text{method}^{RFC} \times \text{method}^{RFC},$
  $\text{call}^{RFC} : \text{class}^{RFC} \times \text{method}^{RFC}\}$
- Mapping α^{RFC}:

$$\begin{aligned} \alpha^{RFC}(\text{Class}) &\mapsto \text{class}^{RFC} \\ \alpha^{RFC}(\text{Method}) &\mapsto \text{method}^{RFC} \\ \alpha^{RFC}(\text{IsMethodOf}) &\mapsto \text{methodof}^{RFC} \\ \alpha^{RFC}(\text{Invokes}) &\mapsto \text{call}^{RFC} \end{aligned}$$

Definition The *RFC* value of a class $c \in class^{RFC}$ is defined as:

$$\begin{aligned} M(c) &= pred(c, \text{methodof}^{RFC}) \\ &\quad \textit{-- set of methods contained in } c \\ R(c) &= succ(M(c), \text{call}^{RFC}) \\ &\quad \textit{-- set of methods called by methods contained in } c \\ RFC(c) &= |M(c) \cup R(c)| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is both negatively and positively influenced by attributed assessed with *Response For a Class*.

Understandability for Reuse 11.4: Understanding if a class is suitable for reuse depends on its complexity and size of the method set it is related to.
Understandability declines with increasing RFC.

Attractiveness 11.4: Attractiveness of a class depends on the complexity of the potentially reused code. Response For a Class allows an assessment of complexity.
Attractiveness increases with increasing RFC.

Maintainability 11.6 declines with increasing response set size.

Analysability 11.6: The effort and time for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified is directly related to the number of executed methods in response to a message.
Analysability declines with increasing RFC.

Changeability 11.6: Each modification must be correct for all execution paths. The size of the response set for a class (RFC) gives an idea about how many methods are potentially contributing to the size of the execution paths.
Changeability declines with increasing RFC.

Testability 11.6: Complete testing requires coverage of all execution paths. Response For a Class computes the number of methods (directly) involved in handling a particular message.
Testability declines with increasing RFC.

Portability 11.7 declines with increasing response set size.

Adaptability 11.7: As for changeability 11.6, the complexity the control structure of software has a direct impact. Each modification must be correct for all execution paths. The size of the response set for a class (RFC) gives an idea about how many methods are potentially contributing to the size of the execution paths.
Adaptability declines with increasing RFC.

Related Software Quality Properties

Reliability 11.2 might decrease with increasing response set size.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by response set size.
Maturity might decline with increasing RFC.

Re-Usability 11.4 is both negatively and positively influenced by attributed assess with Response For a Class.

Learnability for Reuse 11.4: Learning if a class is suitable for reuse depends on the complexity of its interface (public methods) and the number of methods in other classes called in response to a received message.
Learnability declines with increasing RFC.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends the complexity of its interface and dependency on other classes (implementing not local methods).
Programmability declines with increasing RFC.

Efficiency 11.5 might decline with increasing response set size.

Time Behavior 11.5: Static complexity might indicate a higher execution complexity.
Time behavior might get worse with increasing RFC.

Maintainability 11.6 declines with increasing response set size.

Stability 11.6: Due to reduced analysability 11.6 and testability 11.6, also stability may be influenced negatively by response set size.
Stability might decline with increasing RFC.

Portability 11.7 declines with increasing response set size.

Replaceablity 11.7: The substitute of a component must imitate its observable behavior. Components with complex control structures and response sets might have a more complex observable behavior making it more difficult to check substitutability and to actually substitute a component.
Replaceablity declines with increasing RFC.

References

- RFC is extensively discussed and validated in [CK91, CK94, BK95, LH93a, BBC+99, Bal96, CS95, HM95, HM96, HCN98b, BDW99, SK03, MH99a],
- RFC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

12.2 Architecture and Structure

Inheritance

Depth of Inheritance Tree (DIT) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the DIT specific view).

Handle *DIT*

Description Depth of Inheritance Tree (DIT) is the maximum length of a path from a class to a root class in the inheritance structure of a system. DIT measures how many super-classes can affect a class. DIT is only applicable to object-oriented systems.

Scope Class

View $V^{DIT} = (G^{DIT}, R^{DIT})$

- Grammar $G^{DIT} = (\{\text{class}^{DIT}\}, \emptyset, \text{class}^{DIT})$
- Relations $R^{DIT} : \{\text{inheritance}^{DIT} : \text{class}^{DIT} \times \text{class}^{DIT}\}$
- Mapping α^{DIT}:

$$\begin{aligned} \alpha^{DIT}(\text{Class}) &\mapsto \text{class}^{DIT} \\ \alpha^{DIT}(\text{Inheritance}) &\mapsto \text{inheritance}^{DIT} \end{aligned}$$

Definition The *DIT* value of a class $c \in class^{DIT}$ is defined as:

$$\begin{aligned} P(c) &= pred^*(c, \text{inheritance}^{DIT}) \\ &\quad \textit{-- set of classes, c inherits from directly or indirectly} \\ DIT(c) &= max(dist(c, P(c))) \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is both negatively and positively influenced by attributed assess with DIT.

Understandability for Reuse 11.4: Understanding if a class is suitable for reuse depends on the size of its interface. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Understandability declines with increasing DIT.

Learnability for Reuse 11.4: Learning how to use a class depends on the size of its interface. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Learnability declines with increasing DIT.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends on the size of its interface. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Programmability declines with increasing DIT.

Attractiveness 11.4: Attractiveness of a class depends on the size of the potentially reused code. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes.
Attractiveness increases with increasing DIT.

Maintainability 11.6 declines with increasing DIT.

Analyzability 11.6: The effort and time for diagnosis of deficiencies or causes of failures in a class, or for identification of parts to be modified is related to the number of methods of the class. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.

Analyzability declines with increasing DIT.

Changeability 11.6: Changing a class requires prior understanding, which, in turn, is more complicated for classes with many methods. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to understand it.
Changeability declines with increasing DIT.

Testability 11.6: Complete testing requires coverage of all execution paths. The number of possible execution paths of a class increases with the number of methods and their control flow complexity. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Due to late binding, super-class methods need to be tested again in the sub-class context. This makes it it potentially harder to test classes deep down in the classes hierarchy.
Testability declines with increasing DIT.

Portability 11.7 declines with increasing DIT.

Adaptability 11.7: As for changeability 11.6, the size of software has a direct impact. Classes that are deep down in the classes hierarchy potentially inherit many methods from super-classes. Moreover, the definitions of inherited methods are not local to the class making it even harder to analyze it.
Adaptability declines with increasing DIT.

Replaceablity 11.7: The substitute of a component must imitate its interface. Large interfaces are difficult to check for substitutability and to actually substitute. Interface size increases for classes that are deep down in the classes hierarchy.
Replaceablity decline with increasing DIT.

Related Software Quality Properties

Functionality 11.1 might increase with increasing DIT.

Interoperability 11.1: Relating DIT to interoperability requires to be able to locate hierarchy (sub-)structures of a system responsible for interoperability. A high DIT in these hierarchy (sub-)structures might indicate a better ability to interact.
Interoperability might increase with increasing DIT.

Security 11.1: Relating DIT to security requires to be able to locate hierarchy (sub-)structures of a system responsible for in-

teroperability. A high DIT in these hierarchy (sub-)structures might indicate a higher security.
Security might increase with increasing DIT.

Reliability 11.2 is both positively and negatively influenced by attributes assessed with DIT.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs mights be left in the software. Therefore, also maturity may be influenced negatively by DIT.
Maturity might decline with increasing DIT.

Fault-tolerance 11.2: Relating DIT to fault-tolerance requires to be able to locate hierarchy (sub-)structures of a system responsible for interoperability. A high DIT in these hierarchy (sub-)structures might indicate a better ability to interact.
Fault-Tolerance might increase with increasing DIT.

Recoverability 11.2: Relating DIT to recoverability requires to be able to locate hierarchy (sub-)structures of a system responsible for interoperability. A high DIT in these hierarchy (sub-)structures might indicate a higher recoverability.
Recoverability might increase with increasing DIT.

Efficiency 11.5 might decline with increasing DIT.

Time Behavior 11.5: Static size might indicate a higher execution time due to increased number of instruction cache misses, long jumps, etc. Classes inheriting many attributes are potentially large. Moreover, late binding requires indirect calls and prevents optimizations in the context of the caller.
Time behavior might get worse with increasing DIT.

Resource Utilization 11.5: Static size might indicate a higher memory utilization.
Resource utilization might get worse with increasing DIT.

Maintainability 11.6 declines with increasing DIT.

Stability 11.6: Due to reduced analyzability 11.6 and testability 11.6, also stability may be influenced negatively by size.
Stability might decline with increasing DIT.

References

- DIT is extensively discussed and evaluated in [CK91, CK94, BK95, LH93a, BBC+99, Bal96, CS95, HM95, HM96, HCN98b, BDW99, SK03, MH99a, PLLL05],

- DIT is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Number Of Children (NOC) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the NOC specific view).

Handle *NOC*

Description *NOC* is the number of immediate subclasses (children) subordinated to a class (parent) in the class hierarchy. *NOC* measures how many classes inherit directly methods or fields from a super-class. *NOC* is only applicable to object-oriented systems.

Scope Class

View $V^{NOC} = (G^{NOC}, R^{NOC})$

- Grammar $G^{NOC} = (\{\text{class}^{NOC}\}, \emptyset, \text{class}^{NOC})$
- Relations $R^{NOC} : \{\text{inheritance}^{NOC} : \text{class}^{NOC} \times \text{class}^{NOC}\}$
- Mapping α^{NOC}:

$$\begin{aligned} \alpha^{NOC}(\text{Class}) &\mapsto \text{class}^{NOC} \\ \alpha^{NOC}(\text{Inheritance}) &\mapsto \text{inheritance}^{NOC} \end{aligned}$$

Definition The *NOC* value of a class $c \in class^{NOC}$ is defined as:

$$NOC(c) := \left| succ(c, \text{inheritance}^{NOC}) \right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is positively influenced by attributes assessed with NOC.

Understandability for Reuse 11.4: Understanding a class is supported if it has a high number of children. First numerous children show directly a high re-use (re-usability) of the class under concern. Further with numerous children exist many examples on how to re-use the class, which helps to understand weather a class is suitable, and how it can be re-used for a particular tasks.
Understandability increases with increasing NOC.

Learnability for Reuse 11.4: Learning (acquiring and memorizing knowledge) how to (re-)use a class by extending it is influenced by the number of children the class under concern has. A high number of children, that is many re-using child classes, allows acquiring versatile knowledge on how to re-use a particular class in different situations. This supports through redundancy memorizing the knowledge about how to re-use a particular class.
Learnability increases with a increasing NOC.

Operability for Reuse 11.4: Understanding if a class can be integrated and controlled by software engineers/developers depends on the number of children a class has. A high number of children indicates that a particular class is well integrated into an existing software system. This means, that it is suitable for several different tasks, self-descriptive since there exist many examples on its usage, having a higher error tolerance, since it is involved each time one of its children is tested.
Operability increases with increasing NOC.

Attractiveness for Reuse 11.4: A class having a higher number of children appears more attractive to the software engineer/developer. It is a eye-catcher in a class diagram or other representations since it stands out by having many children. Associated assumptions could be, that the class is stable, since it is tested each time a child is tested, that is well documented and understood, since it has been extended so often, that it is easier to understand, since there are many examples of usage, that it helps to understand the children, if the parent class has been understood, that is plays a central role in the design, since many classes extend its functionality.
Attractiveness increases with increasing NOC.

Maintainability 11.6 is negatively and positively influenced by attributes assessed with NOC.

Changeability 11.6: A class having a higher number of children has a lower changeability. The effort spent on modification, fault removal or environmental change is increased, since many child classes are extending its functionality, depending on the parent class. The side effects of modifications are harder to predict. Fault removal effects child classes. The need for environmental change has to consider the child classes.
Changeability decreases with increasing NOC.

Portability 11.7 is negatively influenced by attributes assessed with NOC.

Replaceability 11.7: A class having a higher number of children is difficult to replace. The children are dependent on it, by extending specific functionality and can depend on certain functionality the parent class provides. It is difficult to find another class satisfying these specific needs, allowing to replace the parent class.
Replaceability decreases with increasing NOC.

Related Software Quality Properties

Reliability 11.2 is positively influenced by attributes assessed with *NOC*.

Maturity 11.2: A class having a higher number of children has a high maturity. The frequency of failure by faults is low, since the many faults have been identified in the various ways the child classes interact with the parents. The parent class is heavily used, since it is used each time a child class is used.
Maturity increases with increasing NOC.

Maintainability 11.6 is positively and negatively influenced by attributes assessed with NOC.

Analysability 11.6: To analyze a class having a higher number of children requires higher effort. Diagnosis of deficiencies or causes of failures involves the children of a parent class. They have access to the functionality and data provided by a class and have to be involved in the analysis. Identifying parts in a parent class which need to be modified requires the analysis of all child classes, which are effected by the modification. To analyze a class completely it is also necessary to look at its children to be able to have the complete picture.
Analysability decreases with increasing NOC.

Stability 11.6: A class having a higher number of children bears a higher risk of unexpected effect of modification. Child classes are re-using parent classes in various ways. They are directly effected by modifications making it difficult to predict how stable a class or software product would be after modification.
Stability decreases with increasing NOC.

Testability 11.6: A class having a higher number of children requires a higher effort for validating the modified software. All depending child classes need to be included in the tests as well, since modifications in the parent class have direct impact on the extending child classes. The amount of designed and implemented autonomous test aid functions and test complexity is increased.
Testability decreases with increasing NOC.

Portability 11.7 is negatively influenced by attributes assessed with NOC.

Adaptability 11.7: A class having a higher number of children is difficult adapt to different specified environments. The children are dependent on it, by extending specific functionality and can depend on certain functionality the parent class provides. Adapting the parent class to a new environment, can make it unsuitable for the children, requiring adaptation in them as well.
Adaptability decreases with increasing NOC.

References

- NOC is extensively discussed and evaluated in [CK91, CK94, CS95, BK95, LH93a, BBC+99, Bal96, HM95, HM96, HCN98b, BDW99, SK03, MH99a, PLLL05],
- NOC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Coupling

Afferent Coupling (Ca) Works with all instances of a common metamodel, regardless if they where produced with the Java or the UML frontend. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express

the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model and further to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Afferent Coupling between packages (Ca) measures the total number of external classes coupled to classes of a package due to incoming coupling (coupling from classes external classes of the package, uses CBO definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes do not use the package's classes. Ca is primarily applicable to object-oriented systems.

Scope Package

View $V^{Ca} = (G^{Ca}, R^{Ca})$

- Grammar $G^{Ca} = (\{\text{package}^{Ca}, \text{class}^{Ca}\}, \emptyset, \text{package}^{Ca})$
- Relations $R^{Ca} = \{\text{coupling}^{Ca} : \text{class}^{Ca} \times \text{class}^{Ca}\}$
- Mapping α^{Ca}:

$$\begin{aligned}
\alpha^{Ca}(\text{Class}) &\mapsto \text{class}^{Ca} \\
\alpha^{Ca}(\text{Package}) &\mapsto \text{package}^{Ca} \\
\alpha^{Ca}(\text{Invokes}) &\mapsto \text{coupling}^{Ca} \\
\alpha^{Ca}(\text{Accesses}) &\mapsto \text{coupling}^{Ca} \\
\alpha^{Ca}(\text{IsOfType}) &\mapsto \text{coupling}^{Ca}
\end{aligned}$$

Definition The Ca value of a package $p \in \text{package}^{Ca}$ is defined:

$$\begin{aligned}
CIP(p) &= succ^*(p, \text{contains}^{Ca}) \\
&\quad \textit{-- set of classes inside/contained in } p \\
COP(p) &= \{c \in \text{class}^{Ca} | c \notin CIP(p)\} \\
&\quad \textit{-- set of classes outside } p \\
Coupled(p) &= \{c \in \text{class}^{Ca} | c \in pred(CIP(p), \text{coupling}^{Ca}) \wedge \\
&\quad c \in COP(p)\} \\
&\quad \textit{-- set of classes coupled to } p \textit{ over afferent coupling} \\
Ca(p) &= |Coupled(p)|
\end{aligned}$$

Scale Absolute.

Domain Integers in $0..\infty$.

Highly Related Software Quality Properties

Portability 11.7 is negatively influenced by attributes assessed with Ca.

Replaceability 11.7 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be highly inversely related to replaceability, since other parts depend on it. Replaceability decreases with increasing Ca.

Related Software Quality Properties

Functionality 11.1 is positively and negatively influenced by attributes assessed with Ca.

Interoperability 11.1 Parts of a system showing a high afferent (ingoing) coupling to other system parts may be directly related to interoperability, since they are used/interacted with from other parts of the system.
Interoperability decreases with increasing Ca.

Security 11.1 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to security, since they can be influenced in many ways from other parts of the system.
Security decreases with increasing Ca.

Reliability 11.2 is negatively influenced by attributes assessed with Ca.

Fault-tolerance 11.2 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to fault-tolerance, since a local fault might be propagated to other parts of the system.
Fault-tolerance decreases with increasing Ca.

Re-Usability 11.4 is positively influenced by attributes assessed with Ca.

Learnability 11.3 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be directly related to learnability, since other parts of the system using them serve as examples.
Learnability increases with increasing Ca.

Operability for Reuse – Programmability 11.4 The part of a system that has a high afferent (ingoing) coupling from other system parts may be directly related to programmability, since other parts of the system using it serve as examples.
Operability for Reuse – Programmability increases with increasing Ca.

Attractiveness 11.4 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be directly related to attractiveness, since other parts of the system use them might show a good re-usability.
Attractiveness increases with increasing Ca.

Maintainability 11.6 is negatively influenced by attributes assessed with Ca.

Stability 11.6 Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to stability, since other parts are affected by them.
Stability decreases with increasing Ca.

References

- Ca is discussed in [Mar94],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Coupling Between Objects (CBO) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the CBO specific view).

Handle CBO

Description Coupling Between Objects (CBO) is the number of other classes that a class is coupled to. CBO is only applicable to object-oriented systems.

Scope Class

View $V^{CBO} = (G^{CBO}, R^{CBO})$

- Grammar $G^{CBO} = (\{\text{class}^{CBO}, \text{method}^{CBO}\}, \emptyset, \text{class}^{CBO})$

- Relations $R^{CBO} = \{\text{coupling}^{CBO} : \text{class}^{CBO} \times \text{class}^{CBO}\}$
- Mapping α^{CBO}:

$$\begin{aligned}
\alpha^{CBO}(\text{Class}) &\mapsto \text{class}^{CBO} \\
\alpha^{CBO}(\text{IsDefinedInTermsOf}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{IsParameterOf}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{InheritsFrom}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{IsOfType}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{Accessess}) &\mapsto \text{coupling}^{CBO} \\
\alpha^{CBO}(\text{Invokes}) &\mapsto \text{coupling}^{CBO}
\end{aligned}$$

Definition The CBO value of a class $c \in class^{CBO}$ is defined as:

$$CBO(c) = \left| succ(c, \text{coupling}^{CBO}) \backslash c \right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: Parts, which have a high (outgoing) efferent coupling may be highly inversely related to understandability, since they are using other parts of the system which need to be understood as well.
Understandability decreases with increasing CBO.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing CBO.

Maintainability 11.6 decreases with increasing CBO.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CBO.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CBO.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CBO.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CBO.

Portability 11.7 decreases with increasing CBO.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system that might need to be adapted as well.
Adaptability decreases with increasing CBO.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CBO.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing CBO.

Reliability 11.2 might decrease with increasing CBO.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CBO.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their

data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CBO.

Re-Usability 11.4 might decrease with increasing CBO.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CBO.

Operability for Reuse – Programmability 11.4: Parts having a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CBO.

Efficiency 11.5 might decrease with increasing CBO.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CBO.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CBO.

References

- CBO is extensively discussed and validated in [CK91, CK94, CS95, BK95, LH93a, BBC+99, Bal96, HM95, HM96, HCN98b, BDW99, SK03, MH99a, PLLL05],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Change Dependency Between Classes (CDBC) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are

mapped onto relations of type inheritance in the common meta-model (and the $CDBC$ specific view).

Handle $CDBC$

Description The Change Dependency Between Classes ($CDBC$) measures the class level coupling. It is a measure assigned to pairs of classes describing how dependent one class (client class) is on the other (server class). This allows conclusions on the follow-up work to be done in a client class, when the server class is changed in the course of re-engineering.

Scope Class pairs

View $V^{CDBC} = (G^{CDBC}, R^{CDBC})$

- Grammar $G^{CDBC} = (\{\text{class}^{CDBC}, \text{method}^{CDBC}\}, \emptyset, \text{class}^{CDBC})$
- Relations $R^{CDBC} = \{\text{call}^{CDBC} : \text{method}^{CDBC} \times \text{method}^{CDBC}$,
  $\text{access}^{CDBC} : \text{method}^{CDBC} \times \text{class}^{CDBC}$,
  $\text{containsMethod}^{CDBC} : \text{class}^{CDBC} \times \text{method}^{CDBC}$,
  $\text{inherits}^{CDBC} : \text{class}^{CDBC} \times \text{class}^{CDBC}\}$
- Mapping α^{CDBC}:

$$\begin{aligned}
\alpha^{CDBC}(\text{Class}) &\mapsto \text{class}^{CDBC} \\
\alpha^{CDBC}(\text{Method}) &\mapsto \text{method}^{CDBC} \\
\alpha^{CDBC}(\text{IsMethodOf}) &\mapsto \text{containsMethod}^{CDBC} \\
\alpha^{CDBC}(\text{InheritsFrom}) &\mapsto \text{inherits}^{CDBC} \\
\alpha^{CDBC}(\text{IsOfType}) &\mapsto \text{typeRef}^{CDBC} \\
\alpha^{CDBC}(\text{Accessess}) &\mapsto \text{access}^{CDBC} \\
\alpha^{CDBC}(\text{Invokes}) &\mapsto \text{call}^{CDBC}
\end{aligned}$$

Definition The $CDBC$ value of a pair of classes $CC, SC \in class^{CDBC}$ is

defined as:

$$
\begin{aligned}
CDBC(CC, SC) &= min\,(n, A) \\
&\quad \textit{-- with } n \textit{ is the number of methods} \\
A &= \sum_{impl=1}^{m_1} \alpha_{impl} + (1-k) \sum_{interface=1}^{m_2} \alpha_{interface} \\
&\quad \textit{-- with } m_1 \textit{ are the methods in } CC \textit{ accessing} \\
&\quad \textit{the implementation of } SC \textit{ being possibly} \\
&\quad \textit{affected} \\
&\quad \textit{-- with } m_2 \textit{ are the methods in } CC \textit{ accessing} \\
&\quad \textit{the interface of } SC \textit{ being possibly affected}
\end{aligned}
$$

α is determined according to the following table:

Relationship types between CC and SC	α = number of methods of CC potentially affected by a change
SC is not used by CC at all	0
SC is the class of an instance variable of CC	n
Local variables of type SC are used within j methods of CC	j
SC is a superclass of CC	n
SC is the parameter type of j methods of CC	j
CC accesses a global variable of class SC	n

Table 12.1: Relationship types between CC and SC and their corresponding contribution α to change dependency.

Scale Ordinal.

Domain Integers in $0..\infty$, 0 means there is no change dependency between the pair of classes.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: Parts, which have a high (outgoing) efferent coupling may be highly inversely related to understandability, since they are using other parts of the system which need to be understood as well.
Understandability decreases with increasing CDBC.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing CDBC.

Maintainability 11.6 decreases with increasing CDBC.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CDBC.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CDBC.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CDBC.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CDBC.

Portability 11.7 decreases with increasing CDBC.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system that might need to be adapted as well.
Adaptability decreases with increasing CDBC.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CDBC.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing CDBC.

Reliability 11.2 might decrease with increasing CDBC.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CDBC.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CDBC.

Re-Usability 11.4 might decrease with increasing CDBC.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CDBC.

Operability for Reuse – Programmability 11.4: Parts having a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CDBC.

Efficiency 11.5 might decrease with increasing CDBC.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CDBC.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CDBC.

References

- CDBC is extensively discussed and evaluated in [HM95, HM96, BBC+99, PLLL05],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Change Dependency Of Classes (*$CDOC$*) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the $CDOC$ specific view).

Handle $CDOC$

Description The Change Dependency Of Classes ($CDOC$) measures the class level coupling. It is a measure assigned to classes describing how dependent other classes (client classes) are on this class (server class). This allows conclusions on the follow-up work to be done in all client class, when the server class is changed in the course of re-engineering. It is a accumulation of the $CDBC$ metric, for a server class (SC) and all its client classes (CC).

Scope Class

View $V^{CDOC} = (G^{CDOC}, R^{CDOC})$

- Grammar $G^{CDOC} = (\{\text{class}^{CDOC}\}, \emptyset, \text{class}^{CDOC})$
- Relations $R^{CDOC} = \{\emptyset\}$
- Mapping α^{CDOC}:

$$\alpha^{CDOC}(\text{Class}) \quad \mapsto \quad \text{class}^{CDOC}$$

Definition The $CDOC$ value of a class $c \in class^{CDOC}$ is defined as:

$$CDOC(c) \quad = \sum_{s \in class^{CDOC} \setminus c} CDBC(s, c)$$

Scale Absolute.

Domain Integers in $0..\infty$, 0 means there is no change dependency between the class to any other class in the system.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: Parts having a high (outgoing) efferent coupling may be highly inversely related to understandability, since they are using other parts of the system which need to be understood as well.
Understandability decreases with increasing CDOC.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing CDOC.

Maintainability 11.6 decreases with increasing CDOC.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CDOC.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CDOC.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CDOC.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CDOC.

Portability 11.7 decreases with increasing CDOC.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they

are using other parts of the system that might need to be adapted as well.
Adaptability decreases with increasing CDOC.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CDOC.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing CDOC.

Reliability 11.2 might decrease with increasing CDOC.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CDOC.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CDOC.

Re-Usability 11.4 might decrease with increasing CDOC.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CDOC.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CDOC.

Efficiency 11.5 might decrease with increasing CDOC.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CDOC.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CDOC.

References

- CDOC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Efferent Coupling (Ce) Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model and further to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Efferent Coupling between packages (Ce) measures the total number of external classes coupled to classes of a package due to outgoing coupling (coupling to classes external classes of the package, uses Ce definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes are not used by the package's classes. Ce is primarily applicable to object-oriented systems.

Scope Package

View $V^{Ce} = (G^{Ce}, R^{Ce})$

- Grammar $G^{Ce} = (\{\text{package}^{Ce}, \text{class}^{Ce}\}, \emptyset, \text{package}^{Ce})$

- Relations $R^{Ce} = \{\text{coupling}^{Ce} : \text{class}^{Ce} \times \text{class}^{Ce}\}$
- Mapping α^{Ce}:

$$\begin{aligned}
\alpha^{Ce}(\text{Class}) &\mapsto \text{class}^{Ce} \\
\alpha^{Ce}(\text{Package}) &\mapsto \text{package}^{Ce} \\
\alpha^{Ce}(\text{Invokes}) &\mapsto \text{coupling}^{Ce} \\
\alpha^{Ce}(\text{Accesses}) &\mapsto \text{coupling}^{Ce} \\
\alpha^{Ce}(\text{IsOfType}) &\mapsto \text{coupling}^{Ce}
\end{aligned}$$

Definition The Ce value of a package $p \in \text{package}^{Ce}$ is defined:

$$\begin{aligned}
CIP(p) &= succ^*(p, \text{contains}^{Ce}) \\
&\quad \textit{-- set of classes inside/contained in } p \\
COP(p) &= \{c \in \text{class}^{Ce} | c \notin CIP(p)\} \\
&\quad \textit{-- set of classes outside } p \\
Coupled(p) &= \{c \in \text{class}^{Ce} | c \in succ(CIP(p), \text{coupling}^{Ce}) \wedge \\
&\quad c \in COP(p)\} \\
&\quad \textit{-- set of classes coupled to } p \textit{ over afferent coupling} \\
Ce(p) &= |Coupled(p)|
\end{aligned}$$

Scale Absolute.

Domain Integers in $0..\infty$.

Highly Related Software Quality Properties :

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing Ce.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing Ce.

Maintainability 11.6 decreases with increasing Ce.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing Ce.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing Ce.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing Ce.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing Ce.

Portability 11.7 decreases with increasing Ce.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing Ce.

Related Software Quality Properties :

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing Ce.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing Ce.

Reliability 11.2 might decrease with increasing Ce.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing Ce.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing Ce.

Re-Usability 11.4 might decrease with increasing Ce.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing Ce.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing Ce.

Efficiency 11.5 might decrease with increasing Ce.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing Ce.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing Ce.

References

- Ce is discussed in [Mar94],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Coupling Factor (CF) Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and type reference relations (UML) express the coupling (exclusive inheritance) between two classes. They are mapped to relations of type Invokes, Accesses, and "Is Of Type", respectively, in the common meta model and further to type coupling in the view. By defining a view containing only classes and packages as elements, the metric definition can ignore methods and fields as part of its description, since the relations originating from them are lifted to the class element.

Description Coupling Factor (CF) measures the coupling between classes excluding coupling due to inheritance. It is the ratio between the number of actually coupled pairs of classes in a scope (e.g., package) and the possible number of coupled pairs of classes. CF is primarily applicable to object-oriented systems.

Scope Package

View $V^{CF} = (G^{CF}, R^{CF})$

- Grammar $G^{CF} = (\{\text{package}^{CF}, \text{class}^{CF}\}, P^{CF}, \text{package}^{CF})$
- Productions $P^{CF} = \{\text{package}^{CF} ::= \text{class}^{CF\ *}\}$
- Relations $R^{CF} : \{\text{coupling}^{CF}\}$[1]
- Mapping α^{CF}:

$$\begin{aligned} \alpha^{CF}(\text{Class}) &\mapsto \text{class}^{CF} \\ \alpha^{CF}(\text{Package}) &\mapsto \text{package}^{CF} \\ \alpha^{CF}(\text{Invokes}) &\mapsto \text{coupling}^{CF} \\ \alpha^{CF}(\text{Accesses}) &\mapsto \text{coupling}^{CF} \\ \alpha^{CF}(\text{IsOfType}) &\mapsto \text{coupling}^{CF} \end{aligned}$$

[1] The structural contains^{CF} relation is implicitly defined by the productions P^{CF}.

Definition The CF value of a package $p \in \text{package}^{CF}$ is defined:

$$
\begin{aligned}
Classes(p) &= succ^*(p, \text{contains}^{CF}) \cap \text{class}^{\texttt{CF}} \\
&\quad \textit{-- set of classes contained in } p \\
Coupled(p,c) &= succ(c, \text{coupling}^{CF}) \cap Classes(p) \\
&\quad \textit{-- set of classes contained in } p, \\
&\quad \textit{-- which } c \textit{ is coupled to} \\
CF(p) &= \frac{\sum_{c \in Classes(p)} \mid Coupled(p,c) \mid}{0.5 * \mid Classes(p) \mid^2 - \mid Classes(p) \mid}
\end{aligned}
$$

Scale Absolute.

Domain Integers in $0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: A art of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing CF.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing CF.

Maintainability 11.6 decreases with increasing CF.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing CF.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing CF.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing CF.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing CF.

Portability 11.7 decreases with increasing CF.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing CF.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing CF.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing CF.

Reliability 11.2 might decrease with increasing CF.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing CF.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing CF.

Re-Usability 11.4 might decrease with increasing CF.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing CF.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing CF.

Efficiency 11.5 might decrease with increasing CF.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing CF.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing CF.

References

- CF is discussed in [eAC94, BBC+99, MH99a, HCN98a],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Data Abstraction Coupling (DAC) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the *DAC* specific view).

Handle *DAC*

Description The DAC measures the coupling complexity caused by Abstract Data Types (ADTs). This metric is concerned with the coupling

between classes representing a major aspect of the object oriented design, since the reuse degree, the maintenance and testing effort for a class are decisively influenced by the coupling level between classes. Basically same as CBO, but coupling limited to type references.

Scope Class

View $V^{DAC} = (G^{DAC}, R^{DAC})$

- Grammar $G^{DAC} = (\{\text{class}^{DAC}\}, \emptyset, \text{class}^{DAC})$
- Relations $R^{DAC} : \{\text{referencestype}^{DAC} : \text{class}^{DAC} \times \text{class}^{DAC}\}$
- Mapping α^{DAC}:

$$\begin{aligned} \alpha^{DAC}(\text{Class}) &\mapsto \text{class}^{DAC} \\ \alpha^{DAC}(\text{IsOfType}) &\mapsto \text{referencestype}^{DAC} \end{aligned}$$

Definition The DAC value of a class $c \in class^{DAC}$ is defined as:

$$DAC(c) = \left| succ(c, \text{referencestype}^{DAC}) \backslash c \right|$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it is uses other parts of the system which need to be understood as well.
Understandability decreases with increasing DAC.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing DAC.

Maintainability 11.6 decreases with increasing DAC.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing DAC.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing DAC.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing DAC.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing DAC.

Portability 11.7 decreases with increasing DAC.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing DAC.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing DAC.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing DAC.

Reliability 11.2 might decrease with increasing DAC.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing DAC.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing DAC.

Re-Usability 11.4 might decrease with increasing DAC.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing DAC.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing DAC.

Efficiency 11.5 might decrease with increasing DAC.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing DAC.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing DAC.

References

- DAC is extensively discussed and validated in [LH93a, BBC+99, BDW99, HM95, PLLL05],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Instability (I) Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. It does not relay on relations since it uses existing metric values for its calculation. By defining a view containing only packages as elements, the metric definition can ignore classes, methods and fields as part of its description, since the relations originating from them are not relevant.

Description Instability between packages (I) measures the ratio between the outgoing and the total number of in- and outgoing couplings from classes inside the package from/to classes outside the package (coupling to classes external classes of the package, uses I definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes are not used by the package's classes. I is primarily applicable to object-oriented systems.

Scope Package

View $V^I = (G^I, R^I)$

- Grammar $G^I = (\{\text{package}^I\}, \emptyset, \text{package}^I)$
- Relations $R^I = \{\emptyset\}$
- Mapping α^I:

$$\alpha^I(\text{Package}) \quad \mapsto \quad \text{package}^I$$

Definition The I value of a package $p \in \text{package}^I$ is defined:

$$I(p) \quad = \quad \frac{p.Ce}{p.Ca + p.Ce}$$

Scale Absolute.

Domain Rationale in $0..1.0\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing I.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing I.

Maintainability 11.6 decreases with increasing I.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing I.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing I.

Stability 11.6: Parts of a system showing a high afferent (ingoing) coupling from other system parts may be inversely related to stability, since other parts are affected by them. Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing I.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing I.

Portability 11.7 decreases with increasing I.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing I.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing I.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing I.

Reliability 11.2 might decrease with increasing I.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing I.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing I.

Re-Usability 11.4 might decrease with increasing I.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing I.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing I.

Efficiency 11.5 might decrease with increasing I.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing I.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utiliza-

tion, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing I.

References

- I is discussed in [Mar94],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Locality of Data (LD) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the *LD* specific view).

Handle *LD*

Description The Locality of Data metric relates the amount of data being local the class to the total amount of data used by the class. This relates to the quality of abstraction embodied by the class and allows conclusions on the reuse potential of the class and testability.

Scope Class

View $V^{LD} = (G^{LD}, R^{LD})$

- Grammar $G^{LD} = (\{\text{class}^{LD}, \text{data}^{LD}\}, \emptyset, \text{class}^{LD})$
- Relations $R^{LD} : \{\text{uses}^{LD} : \text{method}^{LD} \times \text{data}^{LD}$,
  $R^{LD} : \{\text{containeddata}^{LD} : \text{class}^{LD} \times \text{data}^{LD}$,
  $R^{LD} : \{\text{containedmethods}^{LD} : \text{class}^{LD} \times \text{method}^{LD}\}$
- Mapping α^{LD}:

$$\begin{aligned}
\alpha^{LD}(\text{Class}) &\mapsto \text{class}^{LD} \\
\alpha^{LD}(\text{AVariable}) &\mapsto \text{data}^{LD} \\
\alpha^{LD}(\text{Method}) &\mapsto \text{method}^{LD} \\
\alpha^{LD}(\text{IsFieldOf}) &\mapsto \text{isdataof}^{LD} \\
\alpha^{LD}(\text{IsParameterOf}) &\mapsto \text{isdataof}^{LD} \\
\alpha^{LD}(\text{IsMethodOf}) &\mapsto \text{ismethodof}^{LD} \\
\alpha^{LD}(\text{inheritsFrom}) &\mapsto \text{inheritsfrom}^{LD}
\end{aligned}$$

$$\begin{aligned} \alpha^{LD}(\text{Accessess}) &\mapsto \text{uses}^{LD} \\ \alpha^{LD}(\text{Invokes}) &\mapsto \text{call}^{LD} \end{aligned}$$

Definition The LD value of a class $c \in class^{LD}$ is defined as:

$$\begin{aligned} S(c) &= succ^{+}(c, \text{inheritsfrom}^{LD}) \\ &\quad \textit{-- set of super classes c inherits from,} \\ &\quad \textit{-- excluding c itself} \\ V^{super}(c) &= \{v_i | v_i \in pred(S(c), \text{isdataof}^{LD} \wedge \\ &\quad v_i.visibility = protected)\} \\ &\quad \textit{-- set of inherited protected instance variables} \\ &\quad \textit{-- of super-classes of c} \\ V(c) &= \{v_i | v_i \in pred(C(c), \text{isdataof}^{LD} \wedge \\ &\quad v_i.visibility \neq public)\} \\ &\quad \textit{-- set of all non-public instance variables of c} \\ &\quad \textit{-- including static variables defined locally in M(c)} \\ V^{local}(c) &= V^{super}(c) \cup V(c) \\ &\quad \textit{-- set of all non-public class local} \\ &\quad \textit{-- instance variables of c} \\ M(c) &= pred(c, \text{ismethodof}^{LD}) \\ &\quad \textit{-- set of methods contained in c} \\ M^{*}(c) &= \{m_i | m_i.isGetterSetter \neq true \wedge m_i \in M(c)\} \\ &\quad \textit{-- set of methods contained in c,} \\ &\quad \textit{-- excluding trivial read/write methods} \\ M^{r/w}(c) &= \{m_i | m_i.isGetterSetter = true \wedge m_i \in M(c)\} \\ &\quad \textit{-- set of all read/write methods contained in c} \\ M^{r/w\ used}(c) &= \{m_i | m_i \in M^{r/w}(c) \cap succ(M^{*}(c), \text{call}^{LD})\} \\ &\quad \textit{-- set of all read/write methods contained in c and} \\ &\quad \textit{-- called by non-read/write methods contained in c} \\ T(c) &= \{v_i | v_i \in succ(M^{*}(c), \text{uses}^{LD})\} \\ &\quad \textit{-- set of all variables used in } M^{*}(c) \textit{ of c,} \\ &\quad \textit{-- excluding local variables defined in } M^{*}(c) \end{aligned}$$

$$
\begin{aligned}
L^{direct}(c) &= T(c) \cap V^{class-local}(c) \\
&\quad \textit{-- set of all variables of c directly used in } M^*(c), \\
&\quad \textit{-- excluding local variables, and variable used} \\
&\quad \textit{-- over getter/setter mtheods} \\
L^{indirect}(c) &= succ(M^{r/w-used}(c), \text{uses}^{LD}) \cap V^{class-local}(c) \\
&\quad \textit{-- set of all variables of c indirectly used in } M^*(c) \\
&\quad \textit{-- over read/write methods} \\
L(c) &= L^{direct}(c) \cup L^{indirect}(c) \\
LD(c) &= \frac{L(c)}{T(c)}
\end{aligned}
$$

Scale Ratio

Domain Rational in 0.0..1.0.

Highly Related Software Quality Properties

Re-Usability 11.4 is positively influenced by coupling.

Understandability for Reuse 11.4: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability increases with increasing LD.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness increases with increasing LD.

Maintainability 11.6 decreases with increasing LD.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability increases with increasing LD.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability increases with increasing LD.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability increases with increasing LD.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability increases with increasing LD.

Portability 11.7 decreases with increasing LD.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability increases with increasing LD.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might decrease with increasing LD.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing LD.

Reliability 11.2 might decrease with increasing LD.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might increase with increasing LD.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might increase with increasing LD.

Re-Usability 11.4 might decrease with increasing LD.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might increase with increasing LD.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might increase with increasing LD.

Efficiency 11.5 might decrease with increasing LD.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might increase with increasing LD.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might increase with increasing LD.

References

- LD is extensively discussed and evaluated in [HM95, PLLL05, BBC+99, HM96],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Message Passing Coupling (MPC ***)*** Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the MPC specific view).

Handle MPC

Description The MPC measures the number of method calls defined in methods of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes. It allows for conclusions on the message passing (method calls) between objects of the involved classes. This allows for conclusions on re-useability, maintenance and testing effort.

Scope Class

View $V^{MPC} = (G^{MPC}, R^{MPC})$

- Grammar $G^{MPC} = (\{\text{class}^{MPC}\}, \emptyset, \text{class}^{MPC})$
- Relations $R^{MPC} : \{\text{call}^{MPC} : \text{class}^{MPC} \times \text{class}^{MPC}\}$
- Mapping α^{MPC}:

$$\begin{aligned} \alpha^{MPC}(\text{Class}) &\mapsto \text{class}^{MPC} \\ \alpha^{MPC}(\text{Invokes}) &\mapsto \text{call}^{MPC} \end{aligned}$$

Definition The MPC value of a class $c \in class^{MPC}$ is defined as:

$$MPC(c) = outdegree^*(c, \text{call}^{MPC})$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing MPC.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing MPC.

Maintainability 11.6 decreases with increasing MPC.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing MPC.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing MPC.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing MPC.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing MPC.

Portability 11.7 decreases with increasing MPC.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.
Adaptability decreases with increasing MPC.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing MPC.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing MPC.

Reliability 11.2 might decrease with increasing MPC.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing MPC.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing MPC.

Re-Usability 11.4 might decrease with increasing MPC.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing MPC.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing MPC.

Efficiency 11.5 might decrease with increasing MPC.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing MPC.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing MPC.

References

- MPC is extensively discussed and validated in [LH93a, BBC+99, BDW99, HM95, PLLL05],
- it is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Package Data Abstraction Coupling ($PDAC$***)*** Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the DAC specific view).

Handle $PDAC$

Description The PDAC measures the coupling complexity caused by ADTs on package level. Based on DAC it transfers the effects of the coupling between classes on the reuse degree, maintenance and testing effort to more abstract organization units like packages and modules, which are as well decisively influenced by the coupling between classes of different packages.

Scope Package

View $V^{PDAC} = (G^{PDAC}, R^{PDAC})$

- Grammar $G^{PDAC} = (\{\text{class}^{PDAC}, \text{package}^{PDAC}\}, \emptyset, \text{package}^{PDAC})$
- Relations $R^{PDAC} : \{\text{referencestype}^{PDAC} : \text{class}^{PDAC} \times \text{class}^{PDAC}, \text{contains}^{PDAC} : \text{package}^{PDAC} \times \text{class}^{PDAC}\}$
- Mapping α^{PDAC}:

$$\begin{aligned} \alpha^{PDAC}(\text{Class}) &\mapsto \text{class}^{PDAC} \\ \alpha^{PDAC}(\text{Package}) &\mapsto \text{package}^{PDAC} \\ \alpha^{PDAC}(\text{IsOfType}) &\mapsto \text{referencestype}^{PDAC} \\ \alpha^{PDAC}(\text{Contains}) &\mapsto \text{contains}^{PDAC} \end{aligned}$$

Definition The $PDAC$ value of a package $p \in package^{PDAC}$ is defined as:

$$\begin{aligned} C(p) &= succ(p, \text{contains}^{PDAC}) \\ PDAC(p) &= \left| succ(C(p), \text{referencestype}^{PDAC}) \backslash C(p) \right| \end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: A part of a system that has a high (outgoing) efferent coupling may be highly inversely related to understandability, since it uses other parts of the system which need to be understood as well.
Understandability decreases with increasing PDAC.

Attractiveness 11.4: Parts that have a high (outgoing) efferent coupling may be highly inversely related to attractiveness, since they are using other parts of the system which need to be understood as well, and represent dependencies.
Attractiveness decreases with increasing PDAC.

Maintainability 11.6 decreases with increasing PDAC.

Analyzability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to analyzability, since they are using other parts of the system which need to be analyzed as well.
Analyzability decreases with increasing PDAC.

Changeability 11.6: Parts that have a high (outgoing) efferent coupling may be inversely related to changeability, since they are using other parts of the system which might need to be changed as well.
Changeability decreases with increasing PDAC.

Stability 11.6: Parts showing a high (outgoing) efferent coupling may be inversely related to stability, since they are using other parts of the system, which are can affect them.
Stability decreases with increasing PDAC.

Testability 11.6: Parts that have a high (outgoing) efferent coupling may be highly inversely related to testability, since they are using other parts of the system which increase the number of possible test paths.
Testability decreases with increasing PDAC.

Portability 11.7 decreases with increasing PDAC.

Adaptability 11.7: Parts that have a high (outgoing) efferent coupling may be inversely related to adaptability, since they are using other parts of the system which might need to be adapted as well.

Adaptability decreases with increasing PDAC.

Related Software Quality Properties

Functionality 11.1 is both negatively and positively influenced by coupling.

Interoperability 11.1: Parts that have a high (outgoing) efferent coupling may be directly related to interoperability, since they are using/interacting with other parts of the system.
Interoperability might increase with increasing PDAC.

Security 11.1: Parts that have a high (outgoing) efferent coupling may be inversely related to security, since they can be affected by security problems in other parts of the system.
Security might decrease with increasing PDAC.

Reliability 11.2 might decrease with increasing PDAC.

Fault-tolerance 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to fault-tolerance, since they can be affected by faults in other parts of the system.
Fault-Tolerance might decrease with increasing PDAC.

Recoverability 11.2: Parts that have a high (outgoing) efferent coupling may be inversely related to recoverability, since their data is distributed in other parts of the system making their recovery difficult.
Recoverability might decrease with increasing PDAC.

Re-Usability 11.4 might decrease with increasing PDAC.

Learnability for Reuse 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system which need to be understood as well.
Learnability might decrease with increasing PDAC.

Operability for Reuse – Programmability 11.4: Parts that have a high (outgoing) efferent coupling may be inversely related to learnability, since they are using other parts of the system, which represent dependencies.
Programmability might decrease with increasing PDAC.

Efficiency 11.5 might decrease with increasing PDAC.

Time Behavior 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to time behavior, since they are using other parts of the system, thus execution during test

or operation does not stay local, but might involve huge parts of the system.
Time behavior might get worse with increasing PDAC.

Resource Utilization 11.5: Parts that have a high (outgoing) efferent coupling may be inversely related to resource utilization, since they are using other parts of the system, thus execution during test or operation does not stay local, but might involve huge parts of the system.
Resource utilization might get worse with increasing PDAC.

References

- PDAC is implemented in the VizzAnalyzer Metrics Suite.

Since Compendium 1.0

Cohesion

Lack of Cohesion in Methods (LCOM) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the *LCOM* specific view).

Handle *LCOM*

Description The Lack of Cohesion in Methods metric is a measure for the number of not connected method pairs in a class representing independent parts having no cohesion. It represents the difference between the number of method pairs not having instance variables in common, and the number of method pairs having common instance variables.

Scope Class

View $V^{LCOM} = (G^{LCOM}, R^{LCOM})$

- Grammar $G^{LCOM} = (\{\text{class}^{LCOM}, \text{method}^{LCOM}, \text{field}^{LCOM}\}, \emptyset, \text{class}^{LCOM})$
- Relations
  $R^{LCOM} : \{\text{methodof}^{LCOM} : \text{method}^{LCOM} \times \text{class}^{LCOM},$
  $\text{fieldof}^{LCOM} : \text{field}^{LCOM} \times \text{class}^{LCOM},$
  $\text{uses}^{LCOM} : \text{method}^{LCOM} \times \text{field}^{LCOM}\}$

- Mapping α^{LCOM}:

$$\begin{aligned}
\alpha^{LCOM}(\text{Class}) &\mapsto \text{class}^{LCOM} \\
\alpha^{LCOM}(\text{field}) &\mapsto \text{field}^{LCOM} \\
\alpha^{LCOM}(\text{Method}) &\mapsto \text{method}^{LCOM} \\
\alpha^{LCOM}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{LCOM} \\
\alpha^{LCOM}(\text{IsMethodOf}) &\mapsto \text{methodof}^{LCOM} \\
\alpha^{LCOM}(\text{Accessess}) &\mapsto \text{uses}^{LCOM}
\end{aligned}$$

Definition The $LCOM$ value of a class $c \in class^{LCOM}$, with n methods $m_1, m_2, \ldots, m_n \in M(c)$, having $\{I(c, m_i) \in F(c)\}$ represent the set of instance variables used by method m_i - there are n such sets $\{I_1, \ldots, I_n \in F(c)\}$ - is defined as:

$$\begin{aligned}
F(c) &= pred(c, \text{fieldof}^{LCOM}) \\
&\quad \textit{-- set of fields contained in } c \\
M(c) &= pred(c, \text{methodof}^{LCOM}) \\
&\quad \textit{-- set of methods contained in } c \\
I(c, m) &= succ(m, \text{uses}^{LCOM}) \cap F(c) \\
&\quad \textit{-- set of fields contained in } c \textit{ and used by } m \\
P(c) &= \{(m_i, m_j) | m_i, m_j \in M(c) \wedge I(c, m_i) \cap I(c, m_j) = \emptyset\} \\
&\quad \textit{-- set of disjunct sets of instant variables of } c \\
Q(c) &= \{(m_i, m_j) | m_i, m_j \in M(c) \wedge I(c, m_i) \cap I(c, m_j) \neq \emptyset\} \\
&\quad \textit{-- set of conjunct sets of instant variables of } c \\
LCOM(c) &= \begin{cases} |P(c)| - |Q(c)| & if\ |P(c)| > |Q(c)| \\ 0 & otherwise \end{cases}
\end{aligned}$$

Scale Absolute.

Domain Integers $\in 0..\infty$.

Highly Related Software Quality Properties

Reliability 11.2 is positively influenced by cohesion.

Maturity 11.2: Parts of a system showing a high cohesion may be highly directly related to maturity, since a mature system ought to have high cohesion values.
Maturity decreases with increasing LCOM.

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: Parts of a system showing a high cohesion may be highly directly related to understandability for reuse, since they implement only one concept.
Understandability decreases with increasing LCOM.

Attractiveness 11.4: Parts of a system showing a high cohesion may be highly directly related to attractiveness for reuse, since they implement only one concept.
Attractiveness decreases with increasing LCOM.

Maintainability 11.6 decreases with increasing LCOM.

Analyzability 11.6: Parts of a system showing a high cohesion may be highly directly related to analyzability, since they implement only one concept.
Analyzability decreases with increasing LCOM.

Changeability 11.6: Parts of a system showing a high cohesion may be highly directly related to changeability, since they implement only one concept.
Changeability decreases with increasing LCOM.

Stability 11.6: Parts of a system showing a high cohesion may be highly directly related to stability, since they implement only one concept.
Stability decreases with increasing LCOM.

Testability 11.6: Parts of a system showing a high cohesion may be highly directly related to testability, since they implement only one concept.
Testability decreases with increasing LCOM.

Portability 11.7 decreases with increasing LCOM.

Adaptability 11.7: Parts of a system showing a high cohesion may be highly directly related to adaptability, since they implement only one concept.
Adaptability decreases with increasing LCOM.

Related Software Quality Properties

Re-Usability 11.4 might decrease with increasing LCOM.

Learnability for Reuse 11.4: Parts of a system showing a high cohesion may be highly directly related to learnability, since they implement only one concept.
Learnability might decrease with increasing LCOM.

Operability for Reuse – Programmability 11.4: Parts of a system showing a high cohesion may be highly directly related to programmability, since they implement only one concept.
Programmability might decrease with increasing LCOM.

Efficiency 11.5 might decrease with increasing LCOM.

Time Behavior 11.5: Parts of a system showing a high cohesion may be directly related to time behavior, since they implement only one concept, and do not do any unrelated time consuming tasks.
Time behavior might get worse with increasing LCOM.

Resource Utilization 11.5: Parts of a system showing a high cohesion may be directly related to resource utilization, since they implement only one concept, and do not do any unrelated resource utilization.
Resource utilization might get worse with increasing LCOM.

Portability 11.7 decreases with increasing LCOM.

Replaceablity 11.7: Parts of a system showing a high cohesion may be directly related to replaceability, since they implement only one concept.
Replaceablity might decreases with increasing LCOM.

References

- LCOM is extensively discussed and evaluated in [CK91, CK94, LH93a, BK95, BBC+99, Bal96, CS95, HM95, HM96, HCN98b, BDW99, SK03, MH99a, PLLL05],
- LCOM is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Improvement of LCOM ($ILCOM$ ***)*** Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the $ILCOM$ specific view).

Handle $ILCOM$

Description The Improvement of LCOM (cf. Lack of Cohesion in Methods 12.2) metric is a measure for the number of connected components

in a class. A component are methods of a class sharing (being connected by) instance variables of the class. The less separate components there are the higher is the cohesion of the methods in the class.

Scope Class

View $V^{ILCOM} = (G^{ILCOM}, R^{ILCOM})$

- Grammar $G^{ILCOM} = (\{\text{class}^{ILCOM}, \text{method}^{ILCOM}, \text{field}^{ILCOM}\}, \emptyset, \text{class}^{ILCOM})$
- Relations
  $R^{ILCOM} : \{\text{methodof}^{ILCOM} : \text{method}^{ILCOM} \times \text{class}^{ILCOM}, \text{fieldof}^{ILCOM} : \text{field}^{ILCOM} \times \text{class}^{ILCOM}, \text{uses}^{ILCOM} : \text{method}^{ILCOM} \times \text{fieILCOM}^{ILCOM}\}$
- Mapping α^{ILCOM}:

$$\begin{aligned}
\alpha^{ILCOM}(\text{Class}) &\mapsto \text{class}^{ILCOM} \\
\alpha^{ILCOM}(\text{Field}) &\mapsto \text{field}^{ILCOM} \\
\alpha^{ILCOM}(\text{Method}) &\mapsto \text{method}^{ILCOM} \\
\alpha^{ILCOM}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{ILCOM} \\
\alpha^{ILCOM}(\text{IsMethodOf}) &\mapsto \text{methodof}^{ILCOM} \\
\alpha^{ILCOM}(\text{Accessess}) &\mapsto \text{uses}^{ILCOM}
\end{aligned}$$

Definition The $ILCOM$ value of a class $c \in class^{ILCOM}$, with n methods $m_1, m_2, \ldots, m_n \in M(c)$, having $\{I(c, m_i) \in F(c)\}$ represent the set of instance variables used by method m_i - there are n such sets $\{I_1, \ldots, I_n \in F(c)\}$ - is defined as:

$$\begin{aligned}
F(c) &= pred(c, \text{fieldof}^{ILCOM}) \\
&\quad \textit{-- set of fields contained in } c \\
M(c) &= pred(c, \text{methodof}^{ILCOM}) \\
&\quad \textit{-- set of methods contained in } c \\
ILCOM(c) &= \left|scc(F(c) \cup M(c), \text{uses}^{ILCOM})\right|
\end{aligned}$$

Scale Absolute.

Domain Integers $\in 1..\infty$.

Highly Related Software Quality Properties

Reliability 11.2 is positively influenced by cohesion.

Maturity 11.2: Parts of a system showing a high cohesion may be highly directly related to maturity, since a mature system ought to have high cohesion values.
Maturity decreases with increasing ILCOM.

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: Parts of a system showing a high cohesion may be highly directly related to understandability for reuse, since they implement only one concept.
Understandability decreases with increasing ILCOM.

Attractiveness 11.4: Parts of a system showing a high cohesion may be highly directly related to attractiveness for reuse, since they implement only one concept.
Attractiveness decreases with increasing ILCOM.

Maintainability 11.6 decreases with increasing ILCOM.

Analyzability 11.6: Parts of a system showing a high cohesion may be highly directly related to analyzability, since they implement only one concept.
Analyzability decreases with increasing ILCOM.

Changeability 11.6: Parts of a system showing a high cohesion may be highly directly related to changeability, since they implement only one concept.
Changeability decreases with increasing ILCOM.

Stability 11.6: Parts of a system showing a high cohesion may be highly directly related to stability, since they implement only one concept.
Stability decreases with increasing ILCOM.

Testability 11.6: Parts of a system showing a high cohesion may be highly directly related to testability, since they implement only one concept.
Testability decreases with increasing ILCOM.

Portability 11.7 decreases with increasing ILCOM.

Adaptability 11.7: Parts of a system showing a high cohesion may be highly directly related to adaptability, since they implement only one concept.
Adaptability decreases with increasing ILCOM.

Related Software Quality Properties

Re-Usability 11.4 might decrease with increasing ILCOM.

Learnability for Reuse 11.4: Parts of a system showing a high cohesion may be highly directly related to learnability, since they implement only one concept.
Learnability might decrease with increasing ILCOM.

Operability for Reuse – Programmability 11.4: Parts of a system showing a high cohesion may be highly directly related to programmability, since they implement only one concept.
Programmability might decrease with increasing ILCOM.

Efficiency 11.5 might decrease with increasing ILCOM.

Time Behavior 11.5: Parts of a system showing a high cohesion may be directly related to time behavior, since they implement only one concept, and do not do any unrelated time consuming tasks.
Time behavior might get worse with increasing ILCOM.

Resource Utilization 11.5: Parts of a system showing a high cohesion may be directly related to resource utilization, since they implement only one concept, and do not do any unrelated resource utilization.
Resource utilization might get worse with increasing ILCOM.

Portability 11.7 decreases with increasing ILCOM.

Replaceablity 11.7: Parts of a system showing a high cohesion may be directly related to replaceability, since they implement only one concept.
Replaceablity might decreases with increasing ILCOM.

References

- ILCOM is extensively discussed and evaluated in [HM95, HM96, BBC+99, PLLL05],
- ILCOM is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

Tight Class Cohesion (TCC) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. The respective extends (Java) or generalization (UML) relations expressing the inheritance between two classes are mapped onto relations of type inheritance in the common meta-model (and the *TCC* specific view).

Handle *TCC*

Description The Tight Class Cohesion metric measures the cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a low cohesion indicate errors in the design.

Scope Class

View $V^{TCC} = (G^{TCC}, R^{TCC})$

- Grammar $G^{TCC} = (\{\text{class}^{TCC}, \text{method}^{TCC}, \text{field}^{TCC}\}, \emptyset, \text{class}^{TCC})$
- Relations R^{TCC} : {methodof^{TCC} : method^{TCC} $\times$ class^{TCC}, fieldof^{TCC} : field^{TCC} $\times$ class^{TCC}, uses^{TCC} : method^{TCC} $\times$ field^{TCC}}
- Mapping α^{TCC}:

$$\begin{aligned}
\alpha^{TCC}(\text{Class}) &\mapsto \text{class}^{TCC} \\
\alpha^{TCC}(\text{Field}) &\mapsto \text{field}^{TCC} \\
\alpha^{TCC}(\text{Method}) &\mapsto \text{method}^{TCC} \\
\alpha^{TCC}(\text{IsFieldOf}) &\mapsto \text{fieldof}^{TCC} \\
\alpha^{TCC}(\text{IsMethodOf}) &\mapsto \text{methodof}^{TCC} \\
\alpha^{TCC}(\text{Accessess}) &\mapsto \text{uses}^{TCC}
\end{aligned}$$

Definition The Tight Class Cohesion (TCC) measures the ratio between the actual number of visible directly connected methods in a class $NDC(C)$ divided by the number of maximal possible number of connections between the visible methods of a class $NP(C)$. Two visible methods are directly connected, if they are accessing the same instance variables of

the class. n is the number of visible methods leading to:

$$
\begin{aligned}
M(c) &= pred(c, \text{methodof}^{TCC}) \\
&\quad \textit{-- set of methods contained in } c \\
M^{public}(c) &= \{m_i | m_i.\text{visibility} = \text{public} \wedge m_i \in M(c)\} \\
&\quad \textit{-- set of methods contained in } c \\
&\quad \textit{-- having public visibility} \\
I(c,m) &= succ(m, \text{uses}^{TCC}) \cap F(c) \\
&\quad \textit{-- set of fields contained in } c \textit{ and used by } m \\
NDC(c) &= \{(m_i, m_j) | I(c,m_i) \cap I(c,m_j) \neq \emptyset \wedge m_i, m_j \in M(c)\} \\
&\quad \textit{-- set of public method pairs contained in } c \\
&\quad \textit{-- accessing the same field} \\
NP(c) &= \frac{\left|M^{public}(c)\right| \left(\left|M^{public}(c)\right| - 1\right)}{2} \\
TCC(c) &= \frac{NDC(c)}{NP(c)}
\end{aligned}
$$

Scale Ratio

Domain Rational $\in$ 0.0..1.0.

Highly Related Software Quality Properties

Reliability 11.2 is positively influenced by cohesion.

Maturity 11.2: Parts of a system showing a high cohesion may be highly directly related to maturity, since a mature system ought to have high cohesion values.
Maturity increases with increasing TCC.

Re-Usability 11.4 is negatively influenced by coupling.

Understandability for Reuse 11.4: Parts of a system showing a high cohesion may be highly directly related to understandability for reuse, since they implement only one concept.
Understandability increases with increasing TCC.

Attractiveness 11.4: Parts of a system showing a high cohesion may be highly directly related to attractiveness for reuse, since they implement only one concept.
Attractiveness increases with increasing TCC.

Maintainability 11.6 increases with increasing TCC.

Analyzability 11.6: Parts of a system showing a high cohesion may be highly directly related to analyzability, since they implement only one concept.
Analyzability increases with increasing TCC.

Changeability 11.6: Parts of a system showing a high cohesion may be highly directly related to changeability, since they implement only one concept.
Changeability increases with increasing TCC.

Stability 11.6: Parts of a system showing a high cohesion may be highly directly related to stability, since they implement only one concept.
Stability increases with increasing TCC.

Testability 11.6: Parts of a system showing a high cohesion may be highly directly related to testability, since they implement only one concept.
Testability increases with increasing TCC.

Portability 11.7 increases with increasing TCC.

Adaptability 11.7: Parts of a system showing a high cohesion may be highly directly related to adaptability, since they implement only one concept.
Adaptability increases with increasing TCC.

Related Software Quality Properties

Re-Usability 11.4 might decrease with increasing TCC.

Learnability for Reuse 11.4: Parts of a system showing a high cohesion may be highly directly related to learnability, since they implement only one concept.
Learnability might decrease with increasing TCC.

Operability for Reuse – Programmability 11.4: Parts of a system showing a high cohesion may be highly directly related to programmability, since they implement only one concept.
Programmability might decrease with increasing TCC.

Efficiency 11.5 might decrease with increasing TCC.

Time Behavior 11.5: Parts of a system showing a high cohesion may be directly related to time behavior, since they implement only one concept, and do not do any unrelated time consuming tasks.
Time behavior might get worse with increasing TCC.

Resource Utilization 11.5: Parts of a system showing a high cohesion may be directly related to resource utilization, since they implement only one concept, and do not do any unrelated resource utilization.
Resource utilization might get worse with increasing TCC.

Portability 11.7 increases with increasing TCC.

Replaceablity 11.7: Parts of a system showing a high cohesion may be directly related to replaceability, since they implement only one concept.
Replaceablity might increases with increasing TCC.

References

- TCC is extensively discussed and evaluated in [BK95, BBC+99, PLLL05],
- TCC is implemented in the VizzAnalyzer Metrics Suite.

Since 1.0

12.3 Design Guidelines and Code Conventions

Documentation

Lack Of Documentation (LOD) Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end.

Handle *LOD*

Description How many comments are lacking in a class, considering one class comment and a comment per method as optimum. Structure and content of the comments are ignored.

Scope Class

View $V^{LOD} = (G^{LOD}, R^{LOD})$

- Grammar $G^{LOD} = (\{\text{scope}^{LOC}\}, \emptyset, \text{scope}^{LOC})$
- Relations $R^{LOD} : IsMethodOf$

- Mapping α^{LOD}:

$$\begin{aligned} \alpha^{LOD}(\text{Class}) &\mapsto \text{scope}^{LOD} \\ \alpha^{LOD}(\text{Method}) &\mapsto \text{method}^{LOD} \\ \alpha^{LOD}(\text{IsMethodOf}) &\mapsto \text{containsMethod}^{LOC} \end{aligned}$$

Definition The LOD value of a element $c \in scope^{LOD}$ is defined as:

$$\begin{aligned} T(c) &= succ(c, containsMethod) \cup c \\ D(c) &= \{e \in T(c) | e.hasJavaDoc == true\} \\ LOD(c) &= 1.0 - \frac{|D(c)|}{|T(c)|} \end{aligned}$$

Scale Rational.

Domain Rational numbers $\in 0.0..1.0\infty$.

Highly Related Software Quality Properties

Re-Usability 11.4 is inversely influenced by LOD.

Understandability for Reuse 11.4: Understanding if a class is suitable for reuse depends on its degree of documentation.
Understandability decreases with increasing LOD.

Learnability for Reuse 11.4: Learning if a class is suitable for reuse depends on the degree of documentation.
Learnability decreases with increasing LOD.

Operability for Reuse – Programmability 11.4: How well a class can be integrated depends on the degree of documentation.
Programmability decreases with increasing LOD.

Maintainability 11.6 decreases with increasing LOD.

Analyzability 11.6: The effort and time for diagnosis of deficiencies or causes of failures in software entity, or for identification of parts to be modified is directly related to its degree of documentation.
Analyzability decreases with increasing LOD.

Changeability 11.6: Changing a class requires prior understanding, which, in turn, is more complicated for undocumented classes.
Changeability decreases with increasing LOD.

Testability 11.6: Writing test cases for classes and methods requires understanding, which, in turn, is more complicated for undocumented classes.
Testability decreases with increasing LOD.

Portability 11.7 decreases with increasing LOD.

Adaptability 11.7: As for changeability 11.6, the degree of documentation of software has a direct impact. Each modification requires understanding which is more complicated for undocumented systems.
Adaptability decreases with increasing LOD.

Replaceablity 11.7: The substitute of a component must imitate its interface. Undocumented interfaces are difficult to check for substitutability and to actually substitute.
Replaceablity decline with increasing LOD.

Related Software Quality Properties

Reliability 11.2 decreases with increasing LOD.

Maturity 11.2: Due to reduced analyzability 11.6 and testability 11.6, bugs might be left in undocumented software. Therefore, maturity may be influenced by degree of documentation.
Maturity decreases with increasing LOD.

Re-Usability 11.4 is inversely influenced by LOD.

Attractiveness 11.3: Attractiveness of a class depends on its adherence to coding conventions such as degree of documentation.
Attractiveness decreases with increasing LOD.

Maintainability 11.6 decreases with increasing LOD.

Stability 11.6: Due to reduced analyzability 11.6 and testability 11.6, also stability may be influenced negatively by size.
Stability decreases with increasing LOD.

References

- LOD is implemented in the VizzAnalyzer Metric Suite.

Code Conventions

None so far.

Bibliography

[ACN05] ACNielsen. The ISO Survey 2004. `http://www.iso.org/iso/en/prods-services/otherpubs/pdf/survey2004.pdf`, 2005. Vienna, Austria.

[AR90] R. Adamov and L. Richter. A Proposal for Measuring the Structural Complexity of Programs. *J. Syst. Softw.*, 12(1):55–70, 1990.

[Bal96] N. V. Balasubramanian. Object-Oriented Metrics. In *APSEC '96: Proceedings of the Third Asia-Pacific Software Engineering Conference*, page 30, Washington, DC, USA, 1996. IEEE Computer Society.

[Bar02] A. L. Baroni. Formal Definition of Object-Oriented Design Metrics. Master's thesis, Vrije Universiteit Brussel, Faculty of Sciences (Belgium) together with Ecole des Mines de Nantes (France) and Universidade Nova de Lisboa (Portugal), 2002.

[BBC+99] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook. `http://www.iam.unibe.ch/~famoos/handbook/`, October 1999.

[BBeA02] A. L. Baroni, S. Braz, and F. Brito e Abreu. Using OCL to Formalize Object-Oriented Design Metrics Definitions. In *ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative Evaluation of Software Quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[BBM95] V. R. Basili, L. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. Technical report, University of Maryland, April 1995.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[BCPeA05] A. L. Baroni, C. Calero, M. Piattini, and F. Brito e Abreu. A Formal Definition for Object-Relational Database Metrics. In *Proceedings of 7th International Conference on Enterprise Information Systems (ICEIS 2005)*, Miami, USA, May 2005.

[BD97] L. C. Briand and J. W. Daly. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. In *Proceedings of the Fourth Conference on METRICS'97*, pages 43–53, November 1997.

[BDM97] L. Briand, P. Devanbu, and W. Melo. An Investigation into Coupling Measures for C++. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 412–421, New York, NY, USA, 1997. ACM Press.

[BDW99] L. C. Briand, J. W. Daly, and J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, 1999.

[Bea00] L. R. Beaumont. *ISO 9001, The Standard Interpretation*. ISO Easy, Middletown, third edition, 2000.

[BeA03a] A. L. Baroni and F. Brito e Abreu. A Formal Library for Aiding Metrics Extraction. In *4th International Workshop on Object-Oriented Re-Engineering at ECOOP'2003, Darmstadt, Germany*, 2003.

[BeA03b] A. L. Baroni and F. Brito e Abreu. An OCL-Based Formalization of the MOOSE Metric Suite. In *7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2003.

[BK95] J. M. Bieman and B. Kang. Cohesion and Reuse in an Object-Oriented System. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 259–262, New York, NY, USA, 1995. ACM Press.

[BMB96] L. C. Briand, S. Morasca, and V. R. Basili. Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, 1996.

[BR88] V. R. Basili and H. D. Rombach. The TAME Project: Towards Improvement-Oriented Software Environments. In *IEEE Transactions on Software Engineering*, volume 14 (6), pages 758–773. IEEE, June 1988.

[BS98] A. B. Binkley and S. R. Schach. Validation of the Coupling Dependency Metric as a Predictor of Run-time Failures and Maintenance Measures. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 452–455, Washington, DC, USA, 1998. IEEE Computer Society.

[BWDP00] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the Relationship between Design Measures and Software Quality in Object-Oriented Systems. *J. Syst. Softw.*, 51(3):245–273, 2000.

[BWIL99] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis. Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 345–354, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[BZ88] W. G. Bail and M. V. Zelkowitz. Program Complexity using Hierarchical Abstract Computers. *Comput. Lang.*, 13(3-4):109–123, 1988.

[CDK98] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Trans. Softw. Eng.*, 24(8):629–639, 1998.

[CK91] S. R. Chidamber and C. F. Kemerer. Towards a Metrics Suite for Object Oriented Design. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, New York, NY, USA, 1991. ACM Press.

[CK94] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. In *IEEE Transactions on Software Engineering*, volume 20 (6), pages 476–493, June 1994.

[CKKW06] P. Crescenzi, V. Kann, M. Karpinski, and G. Woeginger. A Compendium of NP Optimization Problems. `http://www.nada.kth.se/~viggo/wwwcompendium/`, 2006.

[CM78] J. P. Cavano and J. A. McCall. A Framework for the Measurement of Software Quality. In *Proceedings of the software quality assurance workshop on Functional and performance issues*, pages 133–139, 1978.

[CMU07a] Software Engineering Institute (SEI) Carnegie Mellon University. Capability Maturity Model Integration (CMMI) - Overview. `http://www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview05.pdf`, jan 2007.

[CMU07b] Software Engineering Institute (SEI) Carnegie Mellon University. Capability Maturity Model Integration (CMMI), Version 1.1, CMMI for Software Engineering, staged representation. `http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr029.pdf`, January 2007. CMMI Product Team, Pittsburgh 2002.

[CMU07c] Software Engineering Institute (SEI) Carnegie Mellon University. Frequently asked questions about ISO 15504. `http://www.sei.cmu.edu/cmmi/faq/15504-faq.html`, January 2007.

[Cor03] Microsoft Corporation. Microsoft Encarta Reference Library 2003, 2003.

[CS95] N. Churcher and M. Shepperd. Comments on "A Metrics Suite for Object Oriented Design". In *IEEE Transactions on Software Engineering*, volume 21 (3), pages 263–265, 1995.

[CS00] M. Cartwright and M. Shepperd. An Empirical Investigation of an Object-Oriented Software System. *IEEE Trans. Softw. Eng.*, 26(8):786–796, 2000.

[DeM82] T. DeMarco. *Controlling Software Projects: Management, Measurement & Estimation.* Yourdon Press, New York, USA, 1982.

[Dro96] R. G. Dromey. Cornering the Chimera. *IEEE Softw.*, 13(1):33–43, 1996.

[DTS99] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX - The FAMOOS Information Exchange Model, version 2.0. `http://www.iam.unibe.ch/~famoos/FAMIX/`, 1999.

[eAC94] F. Brito e Abreu and R. Carapuça. Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In *4th International Conference on Software Quality*, October 1994. McLean, Virginia, USA.

[EBGR01] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650, 2001.

[Erl00] L. Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Pro*, pages 17–23, May 2000.

[EUR07] EUROCONTROL. Overall Target Architecture Activity (OATA). `http://www.eurocontrol.be/oca/public/standard_page/overall_arch.html`, January 2007.

[EWEBRF05] M. El-Wakil, A. El-Bastawisi, M. Riad, and A. Fahmy. A Novel Approach to Formalize Object-Oriented Design Metrics. In *Proceedings of Evaluation and Assessment in Software Engineering (EASE)*, Keele, UK, 2005.

[FP98] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[fQ07a] American Society for Quality. Glossary. `http://www.asq.org/glossary/q.html`, jan 2007.

[fQ07b] American Society for Quality. The History of Quality – Total Quality. `http://www.asq.org/learn-about-quality/history-of-quality/overview/total-quality.html`, jan 2007.

[fQ07c] American Society for Quality. The History of Quality- Beyond Total Quality. `http://www.asq.org/learn-about-quality/history-of-quality/overview/beyond-total-quality.html`, jan 2007.

[Fre07] C. Freericks. ISO.3 Conformity between V-Model and ISO 12207. `http://www.informatik.uni-bremen.de/gdpa/part3/p3iso2.htm`, March 2007. accessed 2007-01-22.

[GDDC97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-Oriented Languages. In *Procedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, pages 108–124, 1997.

[GeA04] M. Goulão and F. Brito e Abre. Formalizing Metrics for COTS. In *Proceedings of the International Workshop on Models and Processes for the Evaluation of COTS Components (MPEC'04) at the ICSE'2004*, Edinburgh, Scotland, May 2004.

[GGZ04] S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer. *IT Information Technology*, 46(5), 2004.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[Hal77] M. H. Halstead. *Elements of Software Science (Operating and programming systems series).* Elsevier Science Inc., New York, NY, USA, 1977.

[HCN98a] R. Harrison, S. J. Counsell, and R. V. Nithi. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Trans. Softw. Eng.*, 24(6):491–496, 1998.

[HCN98b] R. Harrison, S. J. Counsell, and R. V. Nithi. An Investigation into the Applicability and Validity of Object-Oriented Design Metrics. *Empirical Software Engineering*, 3(3):255–273, 1998.

[HK81] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[HM95] M. Hitz and B. Montazeri. Measure Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95)*, October 1995.

[HM96] M. Hitz and B. Montazeri. Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Trans. Softw. Eng.*, 22(4):267–271, 1996.

[Hum97] W. Humphrey. *Introduction to the Personal Software Process.* SEI Series in Software Engineering. Addison-Wesley, 1997.

[IEE] IEEE. IEEE Std 610.12-1990 "IEEE Standard Glossary of Software Engineering Terminology". `http://ieeexplore.ieee.org/servlet/opac?punumber=2238`.

[Inc05] SPSS Inc. SPSS. `http://www.spss.com`, 2005.

[ISO94] ISO. ISO/IEC 8402 "Quality management and quality assurance – Vocabulary ", 1994.

[ISO00] ISO. ISO 9004:2000 "Quality management systems - Guidelines for performance improvement", 2000.

[ISO01a] ISO. ISO 9001:2000 "Quality management systems - Requirements", 2001.

[ISO01b] ISO. ISO/IEC 9126-1 "Software engineering - Product Quality - Part 1: Quality model", 2001.

[ISO03] ISO. ISO/IEC 9126-3 "Software engineering - Product Quality - Part 3: Internal metrics", 2003.

[ISO05] ISO. ISO 9000:2005 "Quality management systems - Fundamentals and vocabulary", 2005.

[KA01] N. Kececi and A. Abran. Analysing, Measuring and Assessing Software Quality In a Logic Based Graphical Model. `http://www.lrgl.uqam.ca/publications/pdf/602.pdf`, 2001. QUALITA 2001, Annecy, France, 2001, pp. 48-55.

[Kar95] E.-A. Karlsson, editor. *Software Reuse: A Holistic Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[LA96] B. Laguë and A. April. Mapping of Datrix(TM) Software Metrics Set to ISO 9126 Maintainability Sub-Characteristics. `http://www.gelog.etsmtl.ca/publications/pdf/944.pdf`, October 1996. SES '96, Forum on Software Engineering Standards Issues, Montreal, Canada.

[LA97] D. A. Lamb and J. R. Abounader. Data Model for Object-Oriented Design Metrics. Technical report, Queen's University, Kingston, ON, October 1997. ISSN-0836-0227-1997-409.

[LC00] Y. Lee and K. H. Chang. Reusability and Maintainability Metrics for Object-Oriented Software. In *ACM-SE 38: Proceedings of the 38th annual on Southeast regional conference*, pages 88–94, New York, NY, USA, 2000. ACM Press.

[LD02] M. Lanza and S. Ducasse. Beyond Language Independent Object Oriented Metrics: Model Independent Metrics. In *6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2002.

[LH93a] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. *Software Metrics Symposium, 1993. Proceedings., First International*, pages 52–60, 1993.

[LH93b] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. In *IEEE Proc. of the 1st Int. Software Metrics Symposium*, pages 52–60, May 1993.

[LH93c] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[LHKS95] W. Li, S. M. Henry, D. G. Kafura, and R. S. Schulman. Measuring Object-Oriented Design. *Journal of Object-Oriented Programming*, 8(4):48–55, 1995.

[LL05] R. Lincke and W. Löwe. Compendium of Software Quality Standards and Metrics. `http://www.arisa.se/compendium/`, 2005.

[LP05a] W. Löwe and Th. Panas. Rapid Construction of Software Comprehension Tools. *International Journal of Software Engineering and Knowledge Engineering*, 15(6):905–1023, December 2005.

[LP05b] W. Löwe and Th. Panas. Rapid Construction of Software Comprehension Tools. *IJSEKE*, August 2005.

[LS81] B. P. Lientz and E. Swanson. Problems in Application Software Maintenance. In *Communications of the ACM*, volume 24 (11), pages 763–769. ACM, 1981.

[LTP04] T. C. Lethbridge, S. Tichelaar, and E. Ploedereder. The Dagstuhl Middle Metamodel. In *Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003)*, volume 94, pages 7–18, may 2004.

[Mar94] R. Martin. OO Design Quality Metrics – An Analysis of Dependencies (position paper). In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94*, oct 1994.

[MB89] Th. J. McCabe and Ch. W. Butler. Design Complexity Measurement and Testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[McC76] Th. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[McC78] C. L. McClure. A Model for Program Complexity Analysis. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 149–157, Piscataway, NJ, USA, 1978. IEEE Press.

[MH99a] T. Mayer and T. Hall. A Critical Analysis of Current OO Design Metrics. *Software Quality Control*, 8(2):97–110, 1999.

[MH99b] T. Mayer and T. Hall. Measuring OO Systems: A Critical Analysis of the MOOD Metrics. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 108, Washington, DC, USA, 1999. IEEE Computer Society.

[Mor89] K. L. Morris. Metrics for Object-Oriented Software Development Environments. Master's thesis, Massachusetts Institute of Technology, 1989.

[MP06] J. A. McQuillan and J. F. Power. Towards Re-usable Metric Definitions at the Meta-Level. In *PhD Workshop of the 20th European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, July 2006.

[MRW77] J. A. McCall, P. G. Richards, and G. F. Walters. Factors in Software Quality. Technical Report Volume I, NTIS, NTIS Springfield, VA, 1977. NTIS AD/A-049 014.

[OMG99] OMG. Object Management Group. OMG unified modeling language specification version 1.3, 1999.

[OMG06] OMG. Object Management Group. OMG unified modeling language specification version 2.0, 2006.

[OPR03] M. Ortega, M. Pérez, and T. Rojas. Construction of a Systemic Quality Model for Evaluating a Software Product. *Software Quality Control*, 11(3):219–242, 2003.

[Pid03] W. Pidcock. What are the differences between a vocabulary, a taxonomy, a thesaurus, an ontology, and a metamodel? `http://www.metamodel.com/article.php?story=20030115211223271`, 1 2003.

[PLL04] Th. Panas, J. Lundberg, and W. Löwe. Reuse in Reverse Engineering. In *12th International Workshop on Reverse Engineering, Bari, Italy*, June 2004.

[PLL05] Th. Panas, R. Lincke, and W. Löwe. Online-Configuration of Software Visualizations with Vizz3D. In *Proc. of ACM Software Visualization Conference, St. Louis, USA*, 2005.

[PLLL05] Th. Panas, R. Lincke, J. Lundberg, and W. Löwe. A Qualitative Evaluation of a Software Development and Re-Engineering Project. In *SEW '05: Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, pages 66–75, Washington, DC, USA, 2005. IEEE Computer Society.

[PS05] Th. Panas and M. Staron. Evaluation of a Framework for Reverse Engineering Tool Construction. In *ICSM 2005, Budapest, Hungary*, September 2005.

[RB89] P. N. Robillard and G. Boloix. The Interconnectivity Metrics: A New Metric Showing How a Program is Organized. *Journal of Systems and Software*, 10(1):29–39, 1989.

[Rei01] R. Reißing. Towards a Model for Object-Oriented Design Measurement. In *Proceedings 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 71–84, 2001.

[RFB04] C. Raak, M. Fögen, and J. Battenfeld. CMMI, SPICE, ISO 15504. `http://www.wibas.de/download/CMMIundSPICE.pdf`, 2004. accessed 2007-01-22.

[SEI06] SEI. Capability Maturity Model Integration (CMMI). `http://www.sei.cmu.edu/cmmi/cmmi.html`, 2006.

[SK03] R. Subramanyam and M. S. Krishnan. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[SLLL06] D. Strein, R. Lincke, J. Lundberg, and W. Löwe. An Extensible Meta-Model for Program Analysis. In *22nd IEEE International Conference on Software Maintenance*, 2006.

[Som04] I. Sommerville. *Software Engineering (7th ed.)*. Addison Wesley, 2004.

[SPI07] What is SPICE? `http://www.sqi.gu.edu.au/spice/`, January 2007.

[SSS01] M. Spiegel, J. Schiller, and A. Srinivasan. *Probability and Statistics*. McGraw-Hill, New York, 2001.

[TKC99] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An Empirical Study on Object-Oriented Metrics. In *METRICS '99: Proceedings of the 6th International Symposium on Software Metrics*, page 242, Washington, DC, USA, 1999. IEEE Computer Society.

[TP00] F. Tip and J. Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, 2000.

[Uni06] University of Karlsruhe. RECODER. `http://recoder.sourceforge.net`, 2006.

[Wal02] R. E. Walpole. *Probability and Statistics for Engineers and Scientists*. Prentice Hill, NJ, 2002.

[Wey88] E. J. Weyuker. Evaluating Software Complexity Measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, 1988.

[WH02] F. G. Wilkie and T. J. Harmer. Tool Support for Measuring Complexity in Heterogeneous Object-Oriented Software. In *18th IEEE International Conference on Software Maintenance (ICSM'02)*, page 152, 2002.

[Whi97] S. A. Whitmire. *Object Oriented Design Measurement*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[Wik07] Wikipedia.org. Quality. `http://en.wikipedia.org/wiki/Quality`, jan 2007.

[WM96] A. H. Watson and Th. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Publication 500-235*, 1996.

[Woo80] S. N. Woodfield. *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*. PhD thesis, Purdue University, 1980.

[WYF03] H. Washizaki, H. Yamamoto, and Y. Fukazawa. A Metrics Suite for Measuring Reusability of Software Components. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 211, Washington, DC, USA, 2003. IEEE Computer Society.

Växjö University

School of Mathematics and Systems Engineering

ISSN 1650-2647 ISRN VXU-MSI-DA-R--07046--SE