

Continuous Quality Control of Long-Lived Software Systems

Florian Deißeböck



Technische Universität München

Institut für Informatik
der Technischen Universität München

Continuous Quality Control of Long-Lived Software Systems

Florian Deißeböck

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Arndt Bode

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Dr. h.c. H. Dieter Rombach
Technische Universität Kaiserslautern

Die Dissertation wurde am 11.05.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.10.2009 angenommen.

Abstract

Virtually any software dependent organization has a vital interest in reducing its spending for software maintenance activities. In addition to financial savings, for many organizations, the time needed to complete a software maintenance task largely determines their ability to adapt their business processes to changing market situations or to implement innovative products and services. With the present yet increasing dependency on large scale software systems, the ability to change existing software in a timely and economical manner hence becomes critical for numerous enterprises of diverse branches.

The ability of a software system to be changed and extended in an efficient manner is commonly referred to with the term *maintainability*. Despite its widely acknowledged importance, many software developing organizations today do not explicitly define processes nor apply specialized techniques to ensure maintainability. This is especially precarious as the quality of software systems typically undergoes a gradual decay in the process of their evolution and therefore needs to be controlled continuously. We claim that a major obstacle to a mature discipline of maintainability engineering is posed by the unsatisfyingly vague definitions of maintainability used today. Various approaches, usually in the form of quality models, have been proposed over the last four decades to remedy this problem. However, no comprehensive basis for controlling the maintainability of large software systems in a continuous manner has been established so far.

This thesis proposes a novel approach for modeling maintainability that explicitly associates system properties with the activities carried out during maintenance and thereby facilitates a structured decomposition of maintainability. The separation of activities and properties supports the identification of sound quality criteria and allows to reason about their interdependencies. As the activities are the main cost factor in software maintenance, we consider this separation a crucial step towards the ultimate goal of a truly economically justified practice of maintainability engineering. The approach is based on a quality metamodel that supports a systematic construction of maintainability models and fosters preciseness as well as completeness.

Furthermore, we describe how maintainability models defined by the presented metamodel can be operationalized in the maintenance processes to support continuous quality control. This includes the definition a maintainability assurance process based on the presented concepts and a set of supporting tools. These tools enable the design of quality models based on our metamodel as well as the automatic generation of guideline documents to communicate quality requirements to the developers. To support quality assurance activities, the tools allow the generation of review checklists and provide an integration of a defined maintainability model with quality assessment tools used for automated analyses.

We demonstrate the applicability of our maintainability modeling approach, the generation of developer guidelines and review checklists as well as the integrated use of quality assessment tools in multiple case studies carried out in academical and industrial contexts.

Acknowledgements

I'd like to express my gratitude to all people who helped me making this project a success. First, I want to thank Prof. Manfred Broy for his support and for providing a truly open-minded research environment that was, and still is, a fertile ground for my ideas. Also, I want to thank Prof. Dieter Rombach for co-supervising this thesis.

Important parts of my work could not have been realized without the continuous support of our industrial partners. I'd like to use this opportunity to thank everybody I worked with at ABB, Interasco, MAN, Munich Re and Siemens. I particularly enjoyed ABB's warm hospitality during our visits to Finland and Poland.

Furthermore, my thanks go to all the people of Prof. Broy's research group I had the pleasure to work with during the last years. In particular, I want to thank Markus Pizka for initiating me in the world of scientific research and, most importantly, for teaching me to question everybody and everything. My colleagues and friends Elmar Jürgens, Benjamin Hummel and Stefan Wagner were not only tough sparring partners in our frequent discussions but also a tremendous support. Not a single request for helping me out when things got over my head was ever turned out. My deepest thanks for this! Additionally, I want to thank Martin Feilkas, Judith Hartmann, Markus Herrmannsdörfer, Silke Müller, Birgit Penzenstadler, Christian Pfaller, Daniel Ratiu, Sabine Rittmann, Wolfgang Schwitzer, Tilman Seifert and Sebastian Winter for their assistance.

I'd like to thank my friends Eckart Foos, Lorenz Herdeis, Lucien Hoogendoorn and Fabian Nagelmüller for the distraction that was vital to be able to complete this work. Beyond this, I thank Lorenz for inspiring scientific discussions outside the world of software engineering.

The completion of this thesis concludes a 25 year period of education. During all these years my parents and grandparents never ceased to support me morally and materially. What I achieved would not have been possible without them. I want to thank them not only for their assistance but, particularly, for the unconditional and trusting way they gave it. My brother I simply thank for the way he is. His thinking often puts mine back into perspective.

Above all, I want to thank my wife for her enduring support, patience and for giving me the greatest joy of my life, our daughters Johanna and Sophia. I have to thank Johanna for the countless nights of carrying her around. These vain attempts to get her to sleep gave rise to some of the best ideas in this thesis. I am grateful to Sophia for postponing her birth until this thesis was (almost) completed. Tini, Johanna and Sophia, you are doubtlessly the most important factor of success for this work and for all the happiness in my life!

Contents

Abstract	3
Acknowledgments	5
1 Introduction	11
1.1 Problem Statement	11
1.2 Contribution	12
1.3 Contents	14
2 Software Maintenance & Software Product Quality	15
2.1 Software Maintenance & Evolution	15
2.1.1 Terms & Definitions	16
2.1.2 Characteristics of Software Maintenance	18
2.1.3 Software Maintenance Process	19
2.1.4 Software Maintenance Productivity	21
2.2 Software Quality	23
2.2.1 Process Quality vs Product Quality	23
2.2.2 Product Quality	24
2.2.3 Quality Attributes	26
2.2.4 Cost of Quality	26
2.3 Product Quality in Long-lived Systems	27
2.4 Summary	30
3 State of the Art	33
3.1 Maintainability Engineering	33
3.2 Definitional Approaches	34
3.2.1 Concrete Quality Models	35
3.2.2 Quality Modeling Frameworks	38
3.3 Constructive Approaches	44
3.4 Analytic Approaches	47
3.4.1 Software Metrics	47
3.4.2 Metric Methodologies	53
3.4.3 Reviews & Inspections	55
3.4.4 Quality Analysis Tools	56
3.5 Summary	60
4 Defining & Controlling Maintainability	63
4.1 Maintainability Management	63

4.2	An Activity-Based Model for Maintainability	66
4.2.1	Modeling Rationale	67
4.2.2	Overview	71
4.2.3	The Quality Metamodel QMM	76
4.2.4	Example	83
4.3	Operationalization	86
4.3.1	Manual Reviews	88
4.3.2	Automated Assessments	89
4.3.3	Guidelines	92
4.4	Summary	95
5	Tool Support	97
5.1	Quality Model Editor QMM.editor	97
5.1.1	Metamodel Implementation	97
5.1.2	Overview	99
5.1.3	Model Design & Maintenance	101
5.1.4	Checklist & Guideline Generation	102
5.1.5	Implementation & Architecture	103
5.1.6	Summary	104
5.2	Quality Control Toolkit ConQAT	105
5.2.1	Requirements	105
5.2.2	Design Considerations	108
5.2.3	Architecture	109
5.2.4	Configuration	111
5.2.5	Modularization & Feature Overview	114
5.2.6	Documentation	120
5.2.7	Configuration Editor cq.edit	121
5.2.8	Integration with the QMM	122
5.2.9	Summary	124
5.3	Summary	125
6	Case Studies	127
6.1	Model-Based Development of Embedded Systems (MAN)	127
6.1.1	Environment	127
6.1.2	Goals	128
6.1.3	Study Description	128
6.1.4	Results	131
6.1.5	Discussion	135
6.2	Web User Interface Frameworks (Interasco GmbH)	136
6.2.1	Environment	136
6.2.2	Goals	136
6.2.3	Study Description	136
6.2.4	Results	140
6.2.5	Discussion	141
6.3	Mainframe Development Infrastructure (BMW)	143
6.3.1	Environment	143

6.3.2	Goals	144
6.3.3	Study Description	144
6.3.4	Results	148
6.3.5	Discussion	151
6.4	Quality Dashboards (ABB & Munich Re)	152
6.4.1	Environment	152
6.4.2	Goals	152
6.4.3	Study Description	152
6.4.4	Results	156
6.4.5	Discussion	162
6.5	Integrating Manual and Automatic Quality Analysis	163
6.5.1	Environment	163
6.5.2	Goals	164
6.5.3	Study Description	164
6.5.4	Results	169
6.5.5	Discussion	171
6.6	Summary	172
7	Beyond Maintainability	173
7.1	Modeling Usability	173
7.1.1	State of the Art	173
7.1.2	Activity-Based Modeling of Usability	176
7.1.3	A Quality Model for Usability	177
7.1.4	Modeling the ISO 15005	181
7.1.5	Discussion	183
7.2	An Integrated Approach for Quality Modeling	185
7.3	Summary	187
8	Summary and Outlook	189
8.1	Summary	189
8.2	Outlook	191
	Bibliography	197

»Because software maintenance adds functionality,
it's a solution, not a problem.«

Robert L. Glass

1 Introduction

Between 60% and 80% of the total life-cycle cost of long-lived systems are spent during their maintenance phase rather than the initial development phase [37,99,119,190,215]. Importantly, half of these efforts are not devoted to fixing defects or to adapt the systems to a changing technical environment. Instead, they are expended to change existing functionality and to implement new requirements that enable organizations to adapt their business processes to changing market situations or to implement innovative products and services. Due to the high dynamics of requirements in most domains, the ability to implement new requirements in a cost-effective and timely manner is therefore a key factor for the commercial success of today's software systems. With a focus on the software systems themselves, this ability is commonly referred to as *maintainability*.

1.1 Problem Statement

Although the crucial importance of efficient and effective software maintenance is generally acknowledged, software developing organizations rarely apply specialized processes and techniques to assure maintainability. As the quality of long-lived software systems typically undergoes a gradual decay [25,97,224] and, hence, needs to be controlled continuously, this gives cause for concern. The situation is fundamentally different from the area of reliability where *constructive* approaches to prevent defects as well as *analytic* approaches to detect defects have been used for several decades. It is often assumed, that the reluctance to actively control maintainability is due to the fact that maintainability issues typically have long-term and therefore less obvious consequences than reliability issues. While this is certainly true, we claim that another major obstacle to a mature way of assuring maintainability is posed by the unsatisfyingly vague definitions of maintainability used today.

Currently, most software developing organizations strive to define maintainability with guidelines that state what developers should do and what they should not do in order to improve the maintainability of software artifacts. To assess conformance to these guidelines, organizations typically use a combination of manual reviews activities and automated checks with static analysis tools. In most cases the guidelines are based on quality models like the well-known ISO standard 9126 for software product quality [152]. However, as such models specify only high level quality goals like *changeability* or *testability*, organizations are required to augment them with more detailed definitions that can be operationalized in the context of a specific software development project. Due to a lack of accepted techniques for the definition and assessment of maintainability, this is usually done in an ad-hoc manner; i. e. guideline and review checklist authors as well as analysis tool operators individually define their notions of maintainability. Inevitably, this leads to multiple definitions of maintainability that are neither complete nor consistent.

Various approaches, usually in the form of quality models, have been proposed over the last four decades to remedy this problem. However, no comprehensive basis for assessing and improving

the maintainability of large software systems has been established so far. Typically, existing models exhibit at least one of the following problems:

First, they do not define criteria for maintainability at a level that is suitable for an actual assessment. Hence, it is not possible to evaluate if a system complies to stated quality requirements or not. Second, the models tend to omit the rationale behind the required properties of the system. This makes it difficult to describe impacts precisely and therefore to convince developers of the importance of the proposed quality criteria. Third, existing models often use ambiguous decomposition dimensions which leads to inconsistent models and hampers the revelation of omissions and inconsistencies in these models. Fourth, most approaches do not provide details on the operationalization of quality models for analytic and constructive quality assurance activities. This makes it difficult to use quality models as a basis for a continuous quality control practice that counters quality decay that software systems are known to undergo during their evolution.

It can be concluded, that today we lack a discipline of »maintainability engineering« that includes a structured approach for defining what constitutes maintainability plus the required methods for achieving the desired maintainability as well as techniques for the continuous assessment of maintainability to prevent decay.

1.2 Contribution

For a long time, software engineers have recognized that complete and precise specifications are required to build correct and reliable software systems. This thesis proposes to transfer this insight to the area of software maintainability. Concretely, it suggests to base all constructive and analytic approaches to improve and assess quality with appropriate *definitional approaches* that unambiguously define what constitutes maintainability (Fig. 1.1).

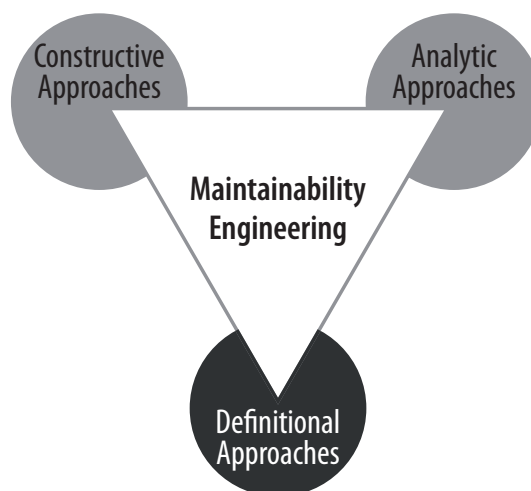


Figure 1.1: The 3 Dimensions of Maintainability Engineering

To achieve this, this thesis introduces a novel approach for modeling maintainability that explicitly associates system properties with the activities carried out during maintenance and thereby facilitates a structured decomposition of maintainability. The separation of activities and properties supports the identification of sound quality criteria and allows to reason about their interdependencies. As the activities are the main cost factor in software maintenance, we consider this separation a crucial step towards the ultimate goal of a truly economically justified practice of maintainability engineering. The approach is based on a quality metamodel that supports a systematic construction of maintainability models and fosters preciseness as well as completeness.

Furthermore, we describe how maintainability models defined by the presented metamodel can be operationalized in the maintenance processes to support continuous quality control. This includes the definition of a maintainability assurance process based on the presented concepts and a set of supporting tools. These tools enable the design of quality models based on our metamodel as well as the automatic generation of guideline documents to communicate quality requirements to the developers. To support quality assurance activities, the tools allow the generation of review checklists and provide an integration of a defined maintainability model with quality assessment tools used for automated analyses.

The latter point is of paramount importance since many quality criteria are subject to gradual decay and, hence, need to be controlled in a continuous and timely manner. As these assessments are known to be cost-intensive, the software maintenance research community and industry created a plethora of analysis tools to automate them. However, these tools are rarely used in a coordinated manner and almost never integrated with a comprehensive quality definition given by a quality model. Hence, we present a flexible quality assessment framework that enables the rapid construction of quality dashboard applications that integrate multiple assessment tools. These applications generate highly aggregated, concise quality reports that are backed by a defined maintainability model. They aid in making project decisions as they provide an integrated view on the project's current quality status.

We demonstrate the applicability of our maintainability modeling and analysis approach with multiple case studies carried out in academical and industrial contexts:

- *MANTUMa*. The largest case study describes the application of the maintainability modeling approach, the generation of developer guidelines and the use of quality assessment tools in the context of model-based development in the automotive domain. The case study was carried out with MAN Nutzfahrzeuge Group.
- *EcoCAP*. The third case study focuses on a quantitative evaluation of the impact that variations in the project infrastructure have on different maintenance activities. The case study was carried out with the BMW Group in the context of mainframe software development.
- *GUI*. In this case study we show how the quality metamodel can be used to compare the expected maintenance efforts of web applications developed with different user interface frameworks. The case study was carried out with INTERASCO GmbH.
- *Dashboards*. Two case studies describe the application of quality dashboards for continuous quality control in industrial contexts. The case studies were carried out with Munich Re and ABB.

- *cq.edit*. The last case study describes the tight integration of manual and automated quality assessment techniques for source code developed by 14 students in a university lab course.

1.3 Contents

The remainder of this thesis is organized as follow: In Chap. 2 we review the basics that are needed for the following chapters. In particular, we describe the field of software maintenance, how it is influenced by software product quality and how quality assurance with focus on maintainability is carried out today. In Chap. 3 we discuss the current state-of-the-art in defining, assessing and improving maintainability. This chapter highlights previous work in the areas of development guidelines, quality modeling, software metrics and quality analysis tools. The main contribution of this thesis is presented in Chap. 4. Here we explain the requirements for operationalizable maintainability models, illustrate the basic principles of our proposed approach and present examples for maintainability models. Finally, we give a formalization of the underlying quality metamodel and explain how models based on it can be operationalized in the software maintenance process. Chap. 5 presents the tools that are required to operationalize the maintainability model. This includes tools for the design of quality models, the generation of guidelines and review checklists, the quality assessment toolkit ConQAT as well as more specialized tools for the detection of duplication in software artifacts. Chap. 6 presents four case studies that were used to demonstrate the feasibility of our approach. The case studies describe applications of our approach for the definition of maintainability and its assessments for business information systems as well as embedded systems in the automotive domain. Chap. 7 broadens the scope of this thesis by explaining how the presented approach can be used for quality goals different from maintainability. We present how the proposed concepts can be used to model *usability* and explain how the metamodel can be used as a basis for an integrated quality assurance that takes multiple quality aspects into account. We close with final conclusions and an outlook on further research in Chap. 8.

Previously Published Material The material covered in this thesis is based, in part, on our contributions in [42, 43, 74, 75, 77–82, 84, 85, 291, 293–295, 304].

»...maintaining software, although costly,
is very worthwhile.«

Gio Wiederhold

2 Software Maintenance & Software Product Quality

This chapter introduces the basic terms and concepts relevant for the remainder of this thesis. Sec. 2.1 discusses the relevance of software maintenance today, gives definitions of important terms, details on its characteristics and illustrates important factors for productivity in software maintenance. Background on software quality in general and the costs associated with assuring and improving quality is given in Sec. 2.2. Sec. 2.3 illustrates reasons and remedies for rapid quality in long-lived software systems before Sec. 2.4 summarizes the chapter.

Previous work cited here serves as basis for this thesis but is not at the core of its contribution. Hence, no discussion of advance on this work is given here. Work directly related to the contributions of this thesis is discussed in the next chapter.

2.1 Software Maintenance & Evolution

The software inventory owned by a company represents a significant share of their property [300]. This is true not only for software companies but for all companies involved with computing. Today, this includes most major companies from industries as diverse as manufacturing or banking. Hence, companies have a vital interest in preserving or extending the value of their software repositories. To achieve this, companies need to counter the gradual decay that a software system's value is known to undergo [186, 224] by continuously adapting the system to changing requirements. This process is usually referred to as *software maintenance* or *software evolution*.

Fig. 2.1 illustrates software maintenance by presenting the software system as a mediator between the problem and the solution domain: The software system realizes the requirements posed by the problem domain with help of the solution domain that provides hardware, programming languages, libraries and other base software like operating systems. The problem domain and the solution domain are known to change over time, e. g. by new or updated requirements or new versions of base software such as libraries. To adapt a software system to implement new requirements of the problem domain with help of a solution domain that is also changing, it needs to be *maintained*.

With respect to the total life cycle cost of a software system, software maintenance is known to account for more than 70% [37, 99, 119, 190, 215]. The growing importance of software maintenance is illustrated by Fig. 2.2 that shows how maintenance costs have developed over the last 35 years with respect to development and hardware costs. In contrast to popular belief, these findings are not limited to business software systems but likewise affect embedded systems software [192]. In fact, some researchers even report that maintenance of embedded systems is one order of magnitude more expensive than maintenance of business software [59].

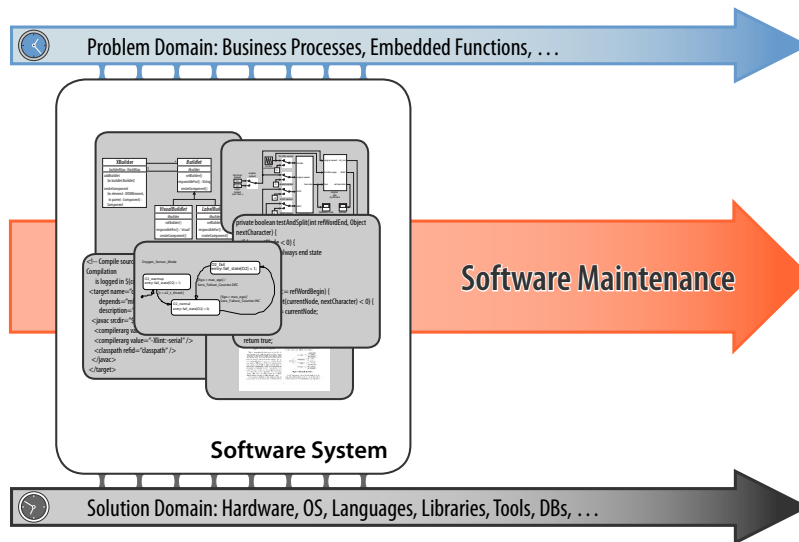


Figure 2.1: Software as Mediator between Problem and Solution Domain [232]

2.1.1 Terms & Definitions

There are numerous definitions of software maintenance, e. g. in [143, 144], that typically define software maintenance as the process of modifying software after its initial delivery. None of the available definitions fully matches our understanding of software maintenance as they either

- state not clearly enough that maintenance concerns all artifacts of software systems including its specifications, documentation, models, configuration files, build scripts and many others, or
- do not include *preventive maintenance*, i. e. maintenance carried out to prepare a software product for future changes, or
- focus on the software process perspective and define maintenance as the part of the software process that starts after delivery, whereas we understand software maintenance as an activity that is not exclusively bound to a specific phase of the software process.

This thesis uses the following definition of software maintenance, that clarifies that software maintenance is an *activity* that affects all artifacts of a software system, possibly at all times. It is important, not to limit the definition of the software maintenance activity to the time after the first delivery of a software system as important maintenance activities like debugging and program comprehension are in fact carried out at all stages of the software process; including initial development.

Definition 1 (Software Maintenance) *The activity of modifying a software system's artifacts to adapt the system to changed requirements, a changed environment, to correct faults or to prepare it for future changes.*

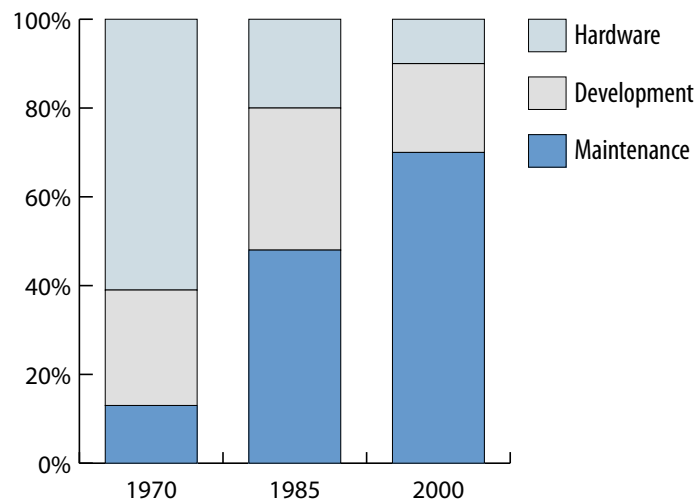


Figure 2.2: Maintenance Efforts

To distinguish the *activity* of maintaining software from the role of software maintenance within the software process, the following definition of the *software maintenance phase* is used. Please note that, for recent development methodologies that are based on small iteratively developed increments, e. g. eXtreme programming, the event of »first delivery« is ill-suited to distinguish the initial development from the maintenance phase. In this cases, the two phases overlap and it is not possible (or sensible) to clearly distinguish between them. However, the given definition of maintenance is not endangered by this problem as it exclusively focuses on the *activity* of maintaining a system.

Definition 2 (Software Maintenance Phase) *The part of the software development process that starts after the first delivery of the system and ends with the system's phase-out.*

The activities carried out when maintaining a software system are structured by the software maintenance process. Please note, that this definition of the maintenance process does not require maintenance to be carried out in a designated *maintenance phase*.

Definition 3 (Software Maintenance Process) *The process that defines the steps and their ordering carried out when software is maintained.*

There is no standard definition for the term *software evolution*. Some researchers and practitioners use it synonymously to *software maintenance* [27] while others use it to refer to particular phases of the software life cycle [240]. However, this thesis uses the term to refer to the evolution process that spans the entire software life cycle.

Definition 4 (Software Evolution) *Software evolution describes the process of developing software initially and then repeatedly updating it. Software evolution comprises the development as well as the maintenance phase.*

Fig. 2.3 illustrates the terms defined above for the case of clearly separated development and maintenance phases within the software process.

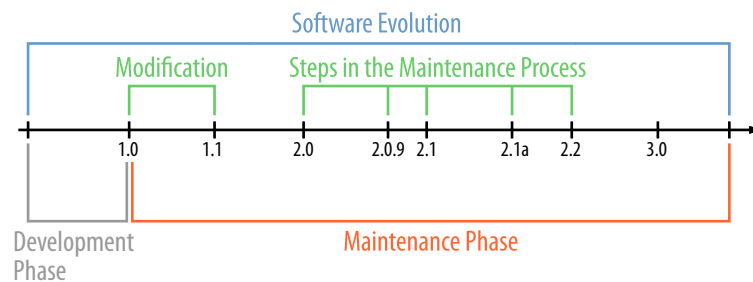


Figure 2.3: Software Maintenance & Evolution [232]

2.1.2 Characteristics of Software Maintenance

Software maintenance is often viewed as a »dirty« type of work that is mainly concerned with fixing bugs in legacy systems written in legacy programming languages like COBOL or PL/1 [120]. In contrast to popular belief, however, software maintenance is mainly concerned with extending the functionality of a software system and not with fixing defects. It thereby increases a software system's value and hence is, as Robert Glass put it in [119], »a solution, not a problem«.

To appropriately capture the diversity of maintenance tasks, previous work usually discusses them with respect to the following categories [143, 190, 215, 231, 274]. Please note, that not all references fully agree on all categories and that the naming of the categories is sometimes inconsistent. Moreover, some researchers [47, 54, 181] propose different, more detailed categorizations. However, for the purpose of this thesis the »classic« categorization proves to be adequate.

- *Perfective Maintenance.* Software maintenance activities that change existing functionality or add new functionality to a software system. This type of maintenance is triggered from the problem domain, e. g. by changes to business processes or new user requests. With respect to the total maintenance costs this type of maintenance typically accounts for about 60% of the maintenance efforts [190, 215].
- *Adaptive Maintenance.* Software maintenance activities that adapt a system to a changing environment. This type of maintenance is triggered from the solution domain, e. g. by changes to base technology like operating systems or changes to third party software systems the system under maintenance interfaces with. This type of maintenance is usually reported to account for about 20% of the maintenance efforts [190, 215].
- *Corrective Maintenance.* Software maintenance activities that correct faults in a software system. This type of maintenance is triggered neither from the problem nor the solution domain but by defects in the system itself. This type of maintenance is usually reported to account for about 17% of the maintenance efforts [190, 215]. While efforts for adaptive and corrective maintenance are both close to 20% , most studies agree that more effort is spent on adaptive than on corrective maintenance.

- *Preventive Maintenance.* Software maintenance activities that prepare a system for prospective changes. This type of maintenance has no explicit trigger but is performed to enhance the efficiency of future maintenance tasks. Examples are restructuring (refactoring), consolidation or redocumentation. Currently, most organizations perform hardly any preventive maintenance. So, the efforts for this type of maintenance are often not reported. If they are, they typically account for about 4% [171].

Fig. 2.4 illustrates the four maintenance types and their share of the maintenance efforts with respect to the entities that trigger them.

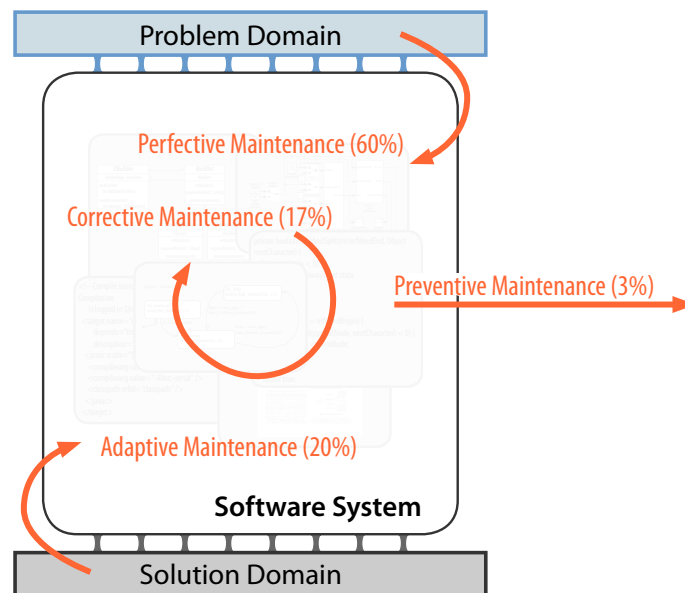


Figure 2.4: Types of Software Maintenance

2.1.3 Software Maintenance Process

For the remainder of this dissertation, it is important to have a thorough understanding of the nature of the software maintenance process. This section gives an introduction of the software maintenance processes and presents the process that is used in this thesis. Most of the maintenance processes proposed in the literature [53, 144, 145] are change-request driven and hence define the required process steps with respect to a single modification request (MR) or change request (CR). The proposed steps are essentially based on the classic waterfall model [252] consisting of an analysis, a design, an implementation and a testing phase. This thesis uses the following process that is modeled on the IEEE standard 1219 [145].

1. *Modification classification.* In this phase, a modification request (MR) or change request (CR) is classified and prioritized. The classification is done based on the previously introduced maintenance types: perfective, adaptive, corrective and preventive. The prioritization is part of the

release planning and is usually based on a company-specific approach that respects various factors, e. g. user-relevance, available resources and dependencies to other CRs [16,254].

2. *Analysis.* The analysis phase studies the feasibility and scope of the modification to devise a preliminary plan for design, implementation, test, and delivery. It usually consists of a *feasibility analysis* and a *detailed analysis* component:
 - a) *Feasibility analysis.* The feasibility analysis investigates, among others, the impact of the modification, alternate solutions, safety and security implications, short-term and long-term costs as well as the value of the benefit of making the modification.
 - b) *Detailed analysis.* The detailed analysis is carried out to define firm requirements for the modification, identify the elements of modification, devise a test strategy and to develop an implementation plan.
3. *Design.* The design phase uses the output of the analysis phase as well as the existing software and its documentation to design the modification to the system.
4. *Implementation.* In the implementation phase, the proposed changes are implemented and all affected artifacts, including the documentation, are updated accordingly.
5. *System test.* In this phase system tests are carried out. This usually includes regression tests to ensure that the modified code did not introduce faults that did not exist prior to the maintenance activity.
6. *Acceptance test.* The acceptance test is usually performed by the customer to ensure that the products of the modification are satisfactory to him.
7. *Delivery.* Depending on the type of software system being maintained, the delivery phase may include installation at the customer facility, notification of the user community and the development of an archival version of the system for backup. When a system modification affects user interfaces or represents a significant modification of the system's functionality, user training may be necessary.

While there is a relatively good understanding of the total maintenance cost and their distribution with respect to the maintenance types introduced above, the research community still lacks a thorough understanding of how these costs are distributed over the phases of the maintenance process [271].

Gaining this understanding from previous studies is still difficult as most of them use differing process models and are therefore hard to compare. Moreover, the comparison is complicated by the large number of factors that influence the effort distribution [213] and often lead to drastic differences across projects [176]. Nevertheless, Table 2.1 aims to give an impression of the distribution by presenting the results from three studies.

¹Although the paper discusses *isolation* (or analysis) as a distinctive maintenance phase, it does not report on the effort spent on it. However, it is not assumed that the authors claim that the analysis phase requires zero effort. In another context [246], Rombach reports on a case where isolation effort is, in fact, higher than the implementation effort.

²The paper compares efforts in separated and joint maintenance environments. Table 2.1 refers to the joint alternative.

Phase	Rombach et al. 1992 [248] ¹	Basili et al. 1996 [16]	Yeh & Jeng 2002 [305] ²
Analysis	–	13%	26%
Design	30%	16%	19%
Implementation	22%	29%	26%
Test	22%	24%	17%
Other	26%	18%	12%

Table 2.1: Distribution of Maintenance Efforts

Interestingly, some of these figures contradict the *reverse engineering* community's claim that understanding the existing program accounts for half of the software maintenance efforts [65, 106, 120, 222]³.

Nevertheless it can be safely stated, that the major differences between initial development of software and software maintenance can be found in the *analysis phase* [13]. Software maintenance is never carried out on the »green field« but needs to take the existing system into account. This does not only limit the space of possible solutions, but foremost requires the existing system to be understood. Consequently, this task, commonly referred to as *program comprehension* or *program understanding*, is a topic of highly active research [60, 288]. A problem that appears to be central to program comprehension is the establishment of a mapping between the program (solution domain) and the real-world (problem domain) [243, 244]. Establishing this mapping is usually referred to as *concept location*, *feature location* or *concept assignment* [32, 55, 239]. Generally, it is assumed that the source code of the system under maintenance does not store all information required by the software maintainer [31, 83]. As additional maintenance documentation is rarely available [215, 221] complex *reverse engineering* activities need to be carried out to regain this information [57, 58, 264].

2.1.4 Software Maintenance Productivity

As pointed out in the introduction, productivity in software maintenance is crucial for virtually any software-developing organization. Consequently, a significant amount of research work has been and is still devoted to gaining a better understanding of the factors that influence maintenance efforts and, hence, productivity in software maintenance. This includes work on maintenance productivity in general, empirical studies [52, 162], productivity models [12, 133], and productivity measurement techniques [7, 273, 276], as well as on the specific productivity factors discussed in the following sections.

While most of the previous research differs in its focus as well as in its conclusions, there appears to be a general agreement that the factors for productivity can be categorized as *process*-related, *personal*-related, *environment*-related and *product*-related. In his work on cost drivers for software maintenance [270], Sneed summarized this with the illustration shown in Fig. 2.5.

The following sections give an overview of factors in the four categories and highlight important previous work dedicated to a better understanding of software maintenance productivity. It has to be noted, that there is relatively little research on maintenance productivity that includes all of the four

³Even more interestingly, it appears that this claim is mainly based on single study that is more than 25 years old [106].

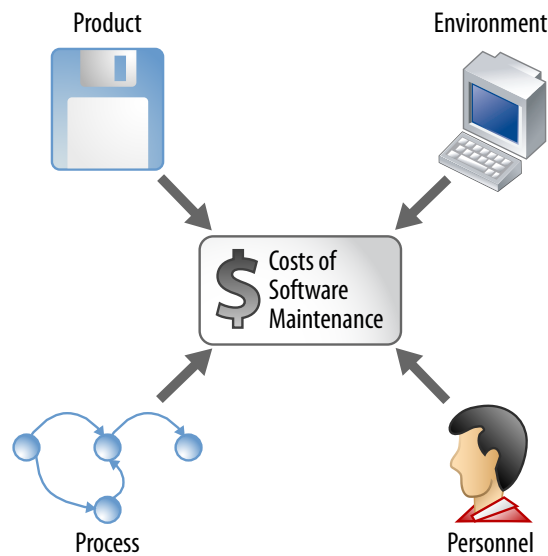


Figure 2.5: Cost Drivers in Software Maintenance [270]

categories [133, 242]. Hence, little is known today about the significance of the various factors with respect to the totality of factors.

Maintenance Personnel Since the early days of software engineering research it is well-known that people are a key factor for the success of software development projects [46]. Hence, numerous publications on productivity show that productivity in software engineering is strongly influenced by the people who carry it out [37, 86, 220]. In fact, various studies found that good developers can be multiple orders of magnitude more productive than average or bad programmers [70, 255]. Recently, agile development methods try to honor these findings by emphasizing the human factor in productivity [136, 237]. While there are relatively few studies dedicated to the factor of personnel in software maintenance productivity [11, 191, 193], there is no reason to believe that this factor may be less significant for software maintenance than for initial software development.

Maintenance Process Likewise, maintenance efforts are influenced by the processes used by the maintaining organization. This applies to special maintenance activities like the change management or version management as well as to standard software development processes used during software maintenance. Again, there is relatively little research dedicated to maintenance processes [81, 236, 240] but there is no reason to believe that this factor may be less significant for software maintenance than initial software development. On the contrary, it is expected that complex maintenance activities like change, release, build and software configuration management are strongly influenced by their underlying processes [14].

Maintenance Environment Another important factor is the environment in which the maintenance activities take place, as appropriate tool support and infrastructure, e. g. debuggers, ver-

sion management systems, and communication systems are known to influence maintenance efforts. There are multiple studies on the productivity gains to be obtained by using tools in software engineering in general [44, 131] and software maintenance in particular [89, 187]. Next to this, there are countless papers on specific maintenance tools like program slicers [72], program visualizers [10], clone detectors [26], maintenance documentation tools [216] and refactoring tools [206] as well as a plethora of similar tools offered by commercial vendors. However, most of these publications lack an evaluation of the tools' contribution to productivity enhancements.

Maintained Product Arguably, the greatest share of research efforts was and still is devoted to the investigation of characteristics of software systems that support or hamper their maintenance. Obviously, this kind of research is completely maintenance-specific as initial development does not need to consider an existing product for development productivity. Previous research concludes that system characteristics as different as system structure [13, 98, 115, 246], module complexity [118], source code format [219], identifier naming [79, 242], comments [112, 126, 133] or combinations of these characteristics [29] influence software maintenance productivity [27]. More work on product characteristics that influence software maintenance is discussed in detail in Sec. 3.4.1.

2.2 Software Quality

The characteristics of software systems that influence software maintenance are often subsumed under the quality attribute *maintainability*. To prepare the discussion of maintainability in the remainder of this thesis, this section first gives an introduction to software quality in general.

In his seminal paper on product quality, Garvin concludes that »Quality is a complex and multifaceted concept« and states that is »also the source of great confusion [...]«. Similarly, [245] finds that »Regardless of the time period or context in which quality is examined, the concept has had multiple and often muddled definitions and has been used to describe a wide variety of phenomena«. While these quotations refer to quality in general, it is well-known that they apply to software quality [178], too. The following sections provide background information to clarify the notion of quality and introduce concepts that are central to the quality modeling approach presented in this thesis.

2.2.1 Process Quality vs Product Quality

For the discussion of software quality it is important to distinguish between *process quality* and *product quality*. *Process quality* describes quality attributes of the software development process whereas *product quality* describes the quality of the developed product. Based on the experience with traditional disciplines, particularly with manufacturing, it is often assumed that high quality processes automatically lead to high quality products [274]. This process-oriented view was the basis for the development of widely-used process quality (maturity) models like CMM [225] and SPICE (ISO 15504) [158]. In some areas, this process orientation, in fact, lead to »quality assessment that is virtually independent of the product itself.« [178].

However, while there is clear link between process and product quality for *manufacturing* processes, the link is not as well established for *development* processes in general and *software development processes* in particular [178,287]. This is illustrated by studies carried out by Capers Jones in 2000 [164]. While researching the correlation between a company's CMM level and the number of shipped defects per function point in their products, he found that, on average, the number of defects decreases with raising CMM level. However, he also found that the best companies at CMM level 1 produce software with less defects than the worst companies at level 5 (Table 2.2).

It is important to remember, that process quality is no end in itself but only a step to product quality. Jones' findings emphasize that, independent from the quality of the applied processes, it is necessary to closely monitor the quality of the outcome of the process (the product).

CMM Level	Minimum	Average	Maximum
1	0.150	0.750	4.500
2	0.120	0.624	3.600
3	0.075	0.473	2.250
4	0.023	0.228	1.200
5	0.002	0.105	0.500

Table 2.2: Delivered Defects per Function Point at CMM Levels [164]

2.2.2 Product Quality

Different disciplines, at different times used, different definitions of product quality [245]. This provides for a rich but sometimes bewildering variety of views on product quality. To clarify this, the following section discusses the categorization presented by Garvin in [114] and puts the categories in context with well-known approaches to software quality (Garvin's five views have also been discussed in the context of software quality in [178]):

- *Transcendental view.* The *transcendental view* is based on Plato's discussion of beauty and defines quality as »innate excellence«. As such it cannot be defined precisely but only be recognized through experience. Nevertheless, it serves as the ideal towards which a product should strive.
- *Manufacturing view.* The *manufacturing view* focuses on the production process of a product and states »getting it right the first time« as its main goal. It exclusively defines quality by »conformance to requirements« and aims at the reduction of cost by avoiding costly rework caused by quality deficiencies. Hence, the *manufacturing view's* focus is internal and does not take into account the users/customers quality needs. Process quality models like CMM and SPICE (ISO 15504) are typically founded on the *manufacturing view*.
- *User view.* While the *manufacturing view* has an internal focus, the *user view* is purely external. It is assumed that quality »lies in the eye of the beholder«. Consequently, definitions of quality that are based on the *user view* are usually highly subjective. While a *user-based definition* raises the practical problem of identifying the user's quality expectation, its relevance has been recognized in software engineering. For example, the ISO 9126 [152] standard defines the

concept of *quality in use*. Obviously, quality aspects related to usability are usually discussed from the user's perspective [30].

- *Product view*. The product view assumes that »differences in quality reflect differences in the quantity of some ingredient or attribute possessed by the product« [114]. Hence, quality can be precisely defined and measured by specifying and assessing product attributes. This view serves as basis for various metrics-based approaches to software quality that measure product characteristics like source code nesting depth to assess the quality of a product.
- *Value-based view*. From the value-based view, quality needs to be defined with respect to cost and benefit. According to this view »a quality product is one that provides performance for an acceptable price or conformance at acceptable cost« [114]. This view extends the previous four with the cost/price aspect and can, hence, be seen as orthogonal to the other views. For software quality, value-based based approaches gained more attention relatively recently [138, 267].

It is important to understand that the different views on quality originate from different contexts and that each of them suits that context well. For example, the manufacturing view was born of the necessities of mass-production that started in the middle of 19th century. Mass-production is inherently based on interchangeable parts and can only work if each part unconditionally conforms to its specification. Hence, »conformance to specification« was the dominant quality requirement that shaped the manufacturing view. From the middle of the 20th century until today, most western societies experienced a significant shift from a production economy to a service economy. The quality definition based on »conformance to specification« that was well-suited for manufacturing turned out to be ill-suited for most services, as they lack a manufacturing process. Consequently, the prevalent view on quality moved from a manufacturing view to a user view that put the satisfaction of the user above everything else [245]. Moreover, it is not only the views on quality that change over time but also the perceived importance of quality attributes. An example is *durability* which today is considered an important element of quality for most products. Before the Industrial Revolution in the early 19th century, durable goods were usually purchased by the poor, as wealthy individuals could afford to buy products that required frequent repair. This resulted in a persistent association of durability with goods of low quality [114].

This excursus on the different notions of quality used by different disciplines at different times is meant to convey that, while the five views on quality can help a substantiated discussion of quality, none of them is used exclusively in any given context, let alone in software engineering. In fact, to do justice to the complex nature of software products, multiple views on quality are required. Not least, this is caused by the multitude of stakeholders typically involved with a software product: While *users* obviously take a user view, *system integrators* are concerned with conformance to specification and hence take a manufacturing view. *Operators* on the other hand study specific product characteristics like memory consumption to ensure smooth operation and thereby adopt a product view. *Managers* of the producer or *purchasers* of the customer, however, need to maximize profit or minimize cost and, hence, take a value-based view. Additional evidence for the need of multiple views on quality to describe software quality can be found in the IEEE's twofold definition of software quality [143]:

- 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.

2.2.3 Quality Attributes

The multitude of views on quality already indicates that finding an unambiguous definition of quality in a given context is challenging. This is aggravated by another dimension of quality usually referred to as *quality attributes*. An example of a quality attribute, *durability*, was given in the previous section. According to the IEEE [143], a quality attribute is »a feature or characteristic that affects an item's quality«. There is no generally accepted set of quality attributes and, hence, different researcher from different communities advocate different attributes using different terms. Examples are *performance*, *reliability*, *conformance*, *serviceability* [114] or *maintainability*, *portability* and *usability* [152]. It is important to note, that different terms, e. g. *serviceability* and *maintainability*, may refer to the same (or similar) attributes. Vice versa, the same term can refer to different attributes in different communities. For example Garvin uses *performance* to describe the »primary operating characteristics« of a product [114] while for software system performance typically refers to execution time and memory characteristics. Even within the software engineering community, the terminology is ambiguous. For example, the IEEE uses the term *performance* for an attribute that the ISO calls *efficiency* while IEEE's *efficiency* appears to refer to ISO's *resource utilization* [143, 152].

Even more problematic is the fact, that it is currently hard to resolve this terminological inconsistencies, as the definition of the attributes is too vague to allow a substantiated discussion. For example, the ISO defines *changeability* as »the capability of the software product to enable a specified modification to be implemented« [152]. While this definition is intuitively understandable, it is difficult to understand its differences or commonalties to IEEE's definition of *extensibility*: »The ease with which a system or component can be modified to increase its storage or functional capacity« [143]. These examples are specifically selected for this thesis and, hence, relate to *maintainability*, the attribute considered important for software maintenance. Nevertheless, similar examples can be found for other attributes like *usability*.

These examples are presented here to illustrate the current confusion regarding the definition of software product quality. An in-depth discussion of quality attributes used in context with software maintenance is given in Chap. 3.

2.2.4 Cost of Quality

Independently from a precise definition of quality or a particular view on it, a thorough understanding of the cost associated with quality and, even more important, a lack thereof is required. Consequently, cost of quality (COQ) was introduced in the 1950ies [102] as a potent tool for the discussion of product quality and meanwhile became widely used in manufacturing [129, 141] and other industries like construction [1]. More recently, cost of quality approaches are also employed in *knowledge work*-disciplines [160]. A survey of previous work on cost of quality can be found in [302].

Quality costs are perceived as a powerful tool to discuss quality as »money is the basic language of upper management« [129]. Hence, quality related issues can be conveyed to upper management more easily in terms of cost than in terms of technical details. Moreover, costs are an appropriate means for the discussion of quality as they provide a unifying view on the diverse aspects of quality. Quality aspects that are highly different in nature, can be interrelated using monetary units. For example, the outdatedness of system documentation and the number of defects caused by memory leaks, two

quality defects measured on totally different scales, can be compared easily by the costs they incur. This, of course, requires these costs to be known.

Traditionally, cost of quality is expressed in the *prevention, appraisal, fault* or PAF-model [129] that distinguishes the cost types shown in Fig. 2.6.

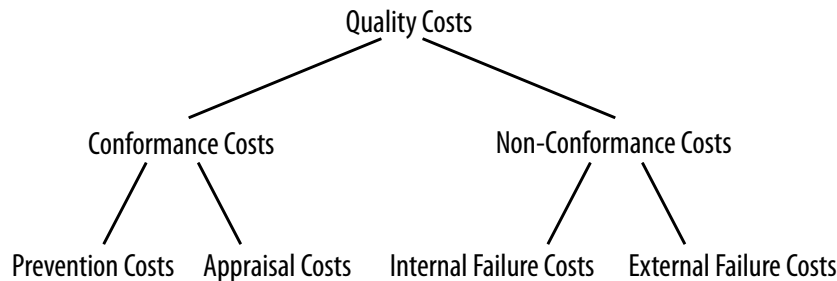


Figure 2.6: Types of Quality Costs

- *Conformance Costs.* The conformance costs comprise all costs that need to be spent to ensure that the software conforms to the quality requirements. This can be further broken down into *prevention* and *appraisal* costs. Prevention costs are costs for means that prevent quality defects. Examples are costs for development tools or costs for the preparation of quality guidelines. The appraisal costs are caused by all means that assess product quality. Examples are cost for reviews or license costs for quality analysis tools.
- *Non-Conformance Costs.* Non-conformance costs are costs caused by the software or parts thereof that do not conform to the quality requirements. Typically, these costs are divided in internal and external failure costs. Internal costs are costs that occur in-house, during development and external costs result from failures at the client's site. For software, non-conformance costs do not only include costs caused by the classic *bugs* but also costs that are caused by less evident failures, e. g. by a lack of usability. Non-conformance costs are also referred to as *price of non-conformance (PONC)* [67] or as *Resultant Poor Quality Costs (Resultant PQC)* [132].

Only relatively recently COQ came into the focus of research and practice in software product quality [182, 184, 269]. To illustrate how COQ is applied to software, Table 2.3 lists types of conformance and non-conformance costs discussed in [197]. It is to be noted that, up to now, quality costs are considered mostly in the context of *dependability* [36, 138, 197, 289] and not with respect to maintainability.

2.3 Product Quality in Long-lived Systems

Assuring the quality of a newly developed software system before its initial release, is challenging already. However, as this section will show, it proves to be even more difficult to continuously control the quality of a software system that is under active maintenance.

Conformance Costs		Non-Conformance Costs
Prevention Costs	Appraisal Costs	
training	review of req. spec.	error-fixing
tools	review of design spec.	re-review of document
methods	review of module spec.	retest of procedure
standards, policies, and procedures	review, inspection	re-review of module of code
consulting	testing process	computer usage for tests
quality planning	beta test, field trial	lab allocations
team prevention meetings	product audits	fixing problems with released systems
fast prototyping		expediting fixes (FAX, overnight mail, e-mail, overtime, etc.)
quality data gathering		
root cause analysis		

Table 2.3: Examples of Quality Costs in Software Engineering [197]

Being immaterial in nature, software systems do not actually wear out like their material counterparts. In the absence of change to its environment, software could function essentially forever as it was originally designed⁴. However, the environment of a software system is never static since the problem domain as well as the solution domain undergo continuous changes. This fact is epitomized by Lehman's first law of software evolution [25]:

I. Law of continuing change. A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.

Changes in both, the problem and the solution domain, inevitably lead to a decline in product quality, if the software is not adapted appropriately:

- *Problem Domain Changes.* If the system is not adapted to meet changing requirements, its effectiveness is reduced. Particularly, the system must implement new requirements to ensure customer satisfaction.
- *Solution Domain Changes.* Certain changes in the problem domain, e. g. modified operating system or API interfaces, can prevent a software system from running if it is not adapted to the changes. More subtly, also changes in the problem domain that do not necessarily demand a system to be adapted to make it work, may lead to a decline in quality. A prominent example is the continuous improvement of user interface (UI) technology. While a program that still uses UI technology from the 1980ies may still work, it will not satisfy users anymore as they are used to modern UIs and, hence, perceive the program as outdated.

In his illustrative analogy between quality decay during software evolution and the aging of the human body, Parnas refers to this reason of quality decay as »lack of movement« [224]. The other main cause for quality decay identified by Parnas is »ignorant surgery«, i. e. changes to the system that are inconsistent with the concepts the designers of the system originally had in mind. This effect is described by Lehman's second law of software evolution [25]:

⁴Provided the software is 100% free of bugs.

II. Law of increasing entropy. The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.

Besides Lehman's work [25, 186], several other studies found that software systems' structures disintegrate over time when continually changed [97, 170, 285]. Furthermore, it was found that quality decay is not limited to system structure but also concerns aspects as diverse as code cloning [185] and quality of identifier naming [79].

The underlying reason for the decay is our inherent inability to predict the future when developing a software system [224]. Even if a system is designed for change, the future will, with high probability, bring changes that the system was not prepared for [241]. Examples for such unanticipated changes are changes caused by new laws or regulations that are decided outside the organization's sphere of influence. Unanticipated changes potentially disrupt the original design of the software system since fundamental assumptions, made when the system was initially designed, may no longer be valid. Consequently, implementing the changes can be costly and time-consuming because the system must undergo a radical change.

However, due to high cost and schedule pressure [120] a thorough implementation of the changes is often skipped in favor of a work-around that is quicker to implement. While the work-around may be initially quicker and cheaper to implement, it constitutes precisely the »ignorant surgery« that leads to quality decay. The situation is worsened as documentation of the changes is often neglected due to the same reasons of high cost and schedule pressure. Furthermore, due to high personnel turnover changes are often performed by people other than the initial developers. This leads to situations where »[...] the original designers no longer understand the product. Those who made the changes, never did. In other words, *nobody* understands the modified product« [224]. As *understanding the existing product* is one of the essential activities of software maintenance this finally leads to an increase in maintenance costs.

The quality decay of software systems is aggravated by a psychological effect described by the *Broken Windows* theory [175, 303]. This theory is originally based on an experiment carried out by Philip Zimbardo in the field of crime prevention but is known to apply to software product quality as well [139]. The experiment showed that a car that already has one window smashed is far more prone to be vandalized than an intact car. Similarly, parts of a software system that already exhibit signals of poor quality, are prone to be handled with less care than appropriate. For crime prevention the effect is successfully countered with a *Zero-Tolerance* policy that prevents Broken Windows by not tolerating »little« crimes. Accordingly, »little« quality problems in software systems need to be addressed immediately to prevent serious problems from creeping into the system [140].

Depending on the type and seriousness of the quality defects, one or more of the following methods are used to remedy the quality problems during software evolution. The *Software Reengineering Assessment Handbook* [284] describes a structured decision process that supports the selection of an appropriate strategy.

- *Refactoring*. One approach to correct quality problems during software evolution is *refactoring*, i. e. »[...] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure« [108]. Refactoring recently became a powerful tool as many often-used refactorings like *Extract Method* or *Rename*

Variable are now well-supported by development environments like Eclipse⁵. The automation of such refactorings increases the efficiency of the refactoring process, helps to avoid mistakes and, above all, reduces developers' reluctance to improve existing code due to its convenience. However, studies have shown that there are limits to refactoring's capability to clean up chaotic code [233].

- *Restructuring*. While refactoring is usually concerned with correcting quality defects on the level of methods, classes or packages, *restructuring* can be applied to resolve defects on an architectural level, i. e. related to larger entities like components.
- *Retargeting*. In many cases quality defects are not caused by a system itself but by technology used by the system. Examples are libraries, frameworks, databases or even the operating system. *Retargeting* describes the process of changing a system to work with a substitute of defect-causing technology. An example is the substitution of Web technology like JSP⁶ that often leads to hard-to-maintain mix of different programming languages within one file, with a new technology, e. g. GWT⁷, that does not suffer from this problem.
- *Translation*. Legacy applications, in particular, suffer from quality defects that are caused by the programming language itself, e. g. COBOL or PL/1 that are considered hard to read in comparison to more recent languages. In such cases quality problems can be resolved by *translating* the existing system to a better suited implementation language.
- *Redevelopment*. In certain situations, quality defects can be so severe, that *developing* or *rewriting* a system from scratch may be the most worthwhile approach to resolve the quality problems.
- *Redocumentation*. Potentially, quality defects can be addressed not by changing the system itself but by providing better documentation to assist maintenance programmers [224, 250].

Please note that none of these measures entails a modification of the functionality of the system. However, in practice quality improvements and changes to functionality often go hand in hand, especially for elaborate processes like retargeting and redevelopment.

2.4 Summary

Due to the prevalent importance of software in virtually all industries today, the software inventory owned by a company represents a significant share of their property. Hence, maintaining this existing software is vital to implement new requirements that enable organizations to adapt their business processes to changing market situations or to implement innovative products and services. However, long-term maintenance of software systems poses a major challenge as it leads to declining quality if no countermeasures are taken. On the long run, this quality decay reduces maintenance efficiency, increases its cost and thereby limits an organization's ability to cope with changing requirements and a changing technical environment.

⁵<http://www.eclipse.org>

⁶Java Server Pages – <http://java.sun.com/products/jsp>

⁷Google Web Toolkit – <http://code.google.com/webtoolkit/>

Industry and academia appear to be well aware of this danger of quality decay in long-lived systems. Nevertheless, most development and maintenance processes do not explicitly address this problem with dedicated activities for continuous quality assessment and improvement. An exception are agile development methods like *eXtreme Programming* [24] that explicitly include phases of refactoring that do not add functionality but exclusively serve to improve product quality.

We claim that this precarious situation is caused by the lack of an established discipline of *maintainability engineering* and the unsatisfactorily vague definition of software quality in general and maintainability in particular. In order to further illustrate this point, the following chapters provide an in-depth discussion of strengths and weaknesses of means currently used to define and assure maintainability.

»It is not enough to do your best; you must know what to do, and then do your best.«

W. Edwards Deming

3 State of the Art

This chapter describes the state of the art in defining, achieving and assessing maintainability. To structure the chapter, we first outline the different dimensions of maintainability engineering and then discuss individual *definitional*, *constructive* and *analytic* approaches. The chapter concludes with a summary of open issues.

3.1 Maintainability Engineering

Currently, neither the state of the art nor the state of the practice has an established discipline of maintainability engineering for software products. For the state of the practice, the lack of an established discipline is illustrated by the results of a survey carried out in Germany in 2003 [174]. In this survey 47 software developing companies from various industries were asked about multiple maintenance-related topics. Interestingly, only 20% of the responding companies analyze their systems for maintainability although 60% of the responding companies spend 40% or more of their total development efforts on software maintenance.

While it is hard to show the non-existence of maintainability engineering for the state of the art, it can be exemplified by opening any introductory book on software engineering. In most cases, such books offer elaborate advice on assuring the reliability of software systems and contain detailed descriptions for the definition of requirements and their verification. However, the same books do rarely offer any advice on the specification of maintainability and its assessment in a given project context. Another hint is given by the number of search results returned by ACM's Digital Library for the search term »software maintainability engineering«: Zero results are returned, whereas the search for the term »software reliability engineering« returns over thousand documents (at the time of writing this thesis).

Although there is no clearly defined discipline of maintainability engineering, various approaches for defining and assuring maintainability were proposed by academia and are nowadays applied by many software developing organizations. As indicated in the introduction (Fig. 1.1), these approaches can be categorized as

- *Definitional Approaches.* For program correctness and reliability, quality is straightforwardly defined as »minimal deviation from requirements«. In maintainability engineering, however, nowadays there usually is no explicit formulation of requirements. Definitional approaches are used to remedy this problem and specify how maintainability is defined. Examples are quality standards like the ISO 9126.
- *Constructive Approaches.* Constructive approaches aim at the *prevention* of quality defects. Examples are guidelines that support developers in creating maintainable systems or the definition of processes that are expected to ensure product quality.

- *Analytic Approaches.* Analytic approaches aim at the *detection* of quality defects in the product. Examples are software metrics, reviews and inspections as well as quality analysis tools.

Obviously, the boundaries between these categories are not clear-cut, e. g. the commonly used *coding conventions* have a definitional as well as a constructive nature as they usually define how maintainable code should look like but also support developer in achieving this goal. In some cases the distinction between the categories is mainly defined by the background and intentions of the original authors that proposed an approach. This is especially true for metric-based approaches where some authors clearly focus on the definitional aspects while others highlight the application in an analytic context. Based on the most prominent features of each approach they are classified either as a quality model or a metric approach and discussed in the respective section. Although the classification of approaches is bound to be ambiguous and, hence, expected to be a matter of debate, we are convinced that it facilitates a structured discussion of the numerous approaches brought forward by the different research communities.

The following sections discuss the state of the art. For each category the most important results are summarized and its strengths and weaknesses are identified. If appropriate, the discussion includes previous work that is not specifically geared towards maintainability but to software quality in general.

3.2 Definitional Approaches

Today, the most important approach to *define* maintainability are *quality models* that were first introduced in the 1970ies to define software quality in general. As quality itself, the term *quality model* is currently not defined as rigidly as it would be desirable. A central problem is that the term *quality model* is often used for concrete models as well as for *quality modeling frameworks* that enable users to build customized quality models. For the latter, the term *quality model* typically describes the underlying *metamodel* of the concrete model instances but often also covers a methodology for developing and using the concrete instances.

For clarity's sake the terms are used in the following way in this thesis:

Definition 5 (Quality Model) *A structured collection of criteria for the systematic assessment of an entity's quality.*¹

Definition 6 (Quality Metamodel) *A model of the constructs and rules needed to build specific quality models.*²

Definition 7 (Quality Modeling Framework) *A framework to define, evaluate and improve quality. This usually includes a quality metamodel as well as a methodology that describes how to instantiate the metamodel and use the model instances for defining, evaluating and improving quality.*

¹Based on the definition given by <http://www.software-kompetenz.de/?11134>

²Based on the definition of metamodel given by <http://www.metamodel.com/article.php?story=20030115211223271>

Please note that the following discussion includes only quality models that are either specifically designed in the context of software maintenance or include maintenance-related aspects. Not considered are quality models for specific quality attributes like *usability*, e. g. [260], *reliability*, e. g. [121] or *dependability*, e. g. [17]. Furthermore, non-software-related models for product quality like [278] are excluded from the discussion.

3.2.1 Concrete Quality Models

The first software quality models were presented in the late 1970ies by McCall et al. [51, 205] and Boehm [38]. These models aim at describing complex quality criteria by breaking them down into more manageable subcriteria. They are designed in a tree-like fashion with abstract quality attributes like *maintainability* or *reliability* at the top and more concrete ones like *analyzability* or *changeability* on lower levels. The leaf factors are ideally detailed enough to be assessed with software metrics. As an example Fig. 3.1 shows the part of Boehm's *Software Quality Characteristics Tree* that describes maintainability [38]. This and similar approaches are called hierarchical or *Factor-Criteria-Metric* (FCM) approaches, since McCall et al. called the top-most nodes *factors*.

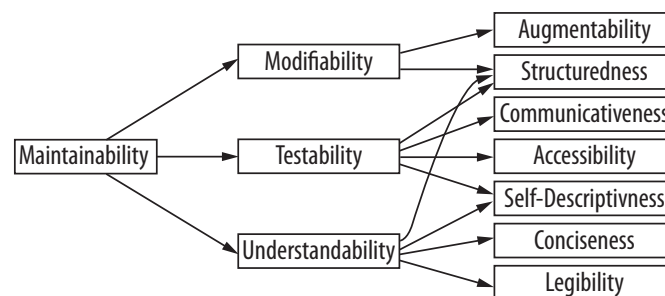


Figure 3.1: Extract of Boehm's *Software Quality Characteristics Tree* [38]

Based on these early works, in 1991 the ISO published the standard 9126 that describes a hierarchical quality model for software. The current version of the standard was defined in 2001 [151]. The model defines the six top-level quality characteristics and 27 subcharacteristics shown in Fig. 3.2. It differs from earlier models as it is strictly hierarchical, i. e. each subcharacteristic is associated with exactly *one* characteristic. The quality model itself does not define metrics for assessing the characteristics but extensions to the standard [154–156] do. Within the scope of the SQuARE initiative the ISO currently reworks the standard 9126 and combines it with several other standards to form the new ISO standard 25000. This standard will contain an updated quality model (ISO 25010) and a specific data quality model (ISO 25012). While these models have not been made publicly available, descriptions in early publications by members of the ISO [2, 34] suggest, that the new model will not significantly differ from the old one³.

Since the introduction of hierarchical quality models, several researchers proposed variations of the approach, e. g. the FURPS [125] and the SATC model [142]. However, these approaches are not discussed in detail here, as they do not fundamentally differ from the initial models. One notable

³Personal communication with a member of the group that develops the ISO 25000 has confirmed this impression.

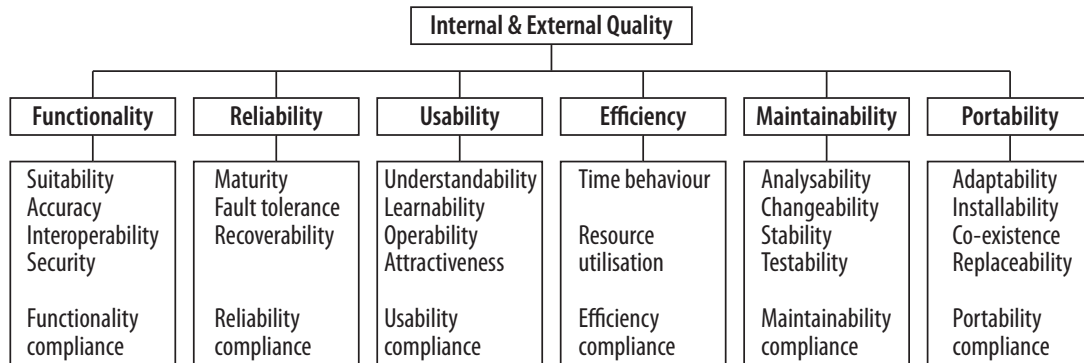


Figure 3.2: ISO 9126 Quality Model [151]

exception is the model presented by Oman & Hagemester in 1992 [218]. This model goes beyond the classic models as it uses criteria that are notably more concrete than the typically used »-ilities« like maintainability and portability. For example, it contains criteria that describe specific properties of the program control and data flow as well as the format of the program source code. Consequently, the model, which defines more than 90 nodes, is significantly larger than the initial models. This model also served as basis for a maintainability model specifically geared to web applications [194].

Discussion The hierarchical models outlined above made a significant contribution to structuring the complex concept of quality. However, none of the existing approaches was able to define a comprehensive basis for assessing and improving the maintainability of software systems. The following section discusses prevalent shortcomings of the approaches. Most of these shortcomings have previously been identified by Kitchenham and other authors [5, 66, 134, 177, 178, 180].

1. *Customizability.* The models discussed above are concrete quality models that are not designed to be adapted to a specific situation. As noted in [177], it is unrealistic to expect that one quality model would fit the great diversity of software systems developed today. Consequently, such models are ill-fitted for most systems as they either lack important criteria or define irrelevant ones.
2. *Assessability.* Most quality models contain criteria that are too coarse-grained to be assessed directly [15]. An example is the *changeability* criterion defined by the ISO 9126 as »the capability of the software product to enable a specified modification to be implemented«. While this definition conveys an intuitive understanding of changeability, it is far from being precise enough to be actually assessed. The lack of assessability seriously reduces the usefulness of a quality model, since defining expected quality without being able to assess whether this definition is adhered to is of questionable use.

The extensions to the ISO 9126 standard [154–156] define metrics to evaluate quality characteristics but these were identified to be insufficient: Firstly, they are rather based on observations of interactions between the product and its environment than on observations of the product itself [134]. Secondly, some of the metrics are defined ambiguously [5]. Thirdly, the validity of

the metrics must be seriously doubted. For example, the sole metric the ISO 9126 defines for the subcharacteristic *changeability* is *change recordability* which is defined as the ratio between »number of changes in functions/modules having change comments confirmed in review« and »total number of functions/modules changed from original code«. It is unclear how this metric relates to the definition of changeability given above.

3. *Rationale*. Independently of this extreme example found in the ISO 9126, most existing quality models fail to give a detailed account of the impact that specific criteria (or metrics) have on software maintenance. Hence, they do not explain why a specific criterion is relevant. In particular, they do not describe how satisfying or violating a quality requirement affects the maintenance activities carried out on the software system. This does not only complicate reasoning about software quality but also makes it very difficult, in practice, to motivate engineers to comply to quality criteria when their rationale is unclear. In particular, the lack of rationale hampers reasoning about the quality model's completeness [178] and inhibits the identification of underlying causes of quality problems [201].
4. *Structuredness*. As Kitchenham et al. noted, hierarchical models typically suffer from a »somewhat arbitrary selection of characteristics and subcharacteristics« [177]. For example, they found that »it is not clear why portability is a top-level characteristic of ISO 9126 but interoperability is a subcharacteristic of functionality« [178]. Al-Kilidar et al. also found overlapping definitions of concepts in the ISO 9126. [5]. Another example is the *instrumentation* criterion used in McCall's model depicted in Fig. 3.3: While *consistency*, *conciseness*, ... are attributes of a system or its entities, *instrumentation* is a technique used to monitor runtime properties of a system. It remains unclear why this (and only this) technique has been included in the model at this location.

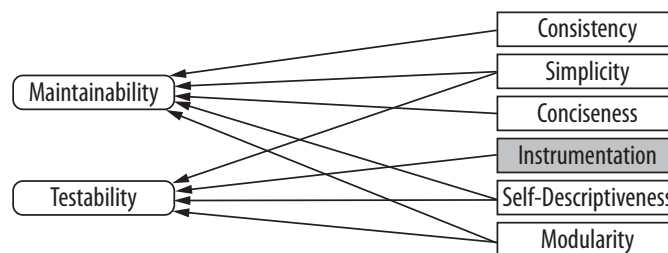


Figure 3.3: Extract of McCall's Quality Model [177, 205]

We consider this lack of structure to be an effect of two fundamental problems:

- a) As already discussed in 2.2.3, the definitions of quality attributes like *portability* and *functionality* are highly ambiguous. Thus, a hierarchical decomposition of these attributes must be expected to be ambiguous as well and is, hence, bound to be perceived as *arbitrary*. While it is sometimes claimed that the quality characteristics defined by the ISO 9126 facilitate communication concerning software quality [134], we agree with [5] that the common language proposed by the ISO 9126 is far from practical and, in fact, often complicates communication. The underlying problem is that the hierarchical models aim at defining complex terms like *portability* through terms like *adaptability* and *replaceability* which in itself are highly complex and call for precise definition.

- b) This underspecification cannot be overcome by simply adding more levels to the hierarchical decomposition. Rather, it is our experience (see [43]), that a further hierarchical decomposition does not help to clarify the model, but in contrast, leads to a highly entangled model that is even more disputable. We claim that the underlying problem is the lack of a clearly defined decomposition criterion that unambiguously defines how a complex term like *adaptability* is further decomposed. The problem is illustrated by the fact that even for the top level attributes various similar but distinctive hierarchical models exist.

In general it must be noted, that none of the existing hierarchical approaches explicitly defines a metamodel that would clarify the exact nature of the model elements and, foremost, the semantics of their interdependencies. Interestingly, the older models made at least some attempt to define the semantics of the edges in their model diagrams. For example Boehm et al. state that »the direction of the arrow [see Fig. 3.1] represents a logical implication: if a program is Maintainable it must necessarily be Understandable and Testable« [38]. The ISO 9126, on the other hand, leaves the interpretation of its model diagram completely to the reader. The lack of a precisely defined model semantics not only makes it difficult to interpret a model but hampers the aggregation of metric values along the hierarchical levels. Consistently, the ISO 9126 does not even discuss the topic of metric aggregation [5].

5. *Operationalization*. Most times, quality models are expressed in prose and graphics only. They accompany the development process in the form of documents but are not made an integral artifact that is tightly coupled with quality assurance activities. While most approaches define metrics that are supposedly used to assess a system's quality, they do not define how the approach can be integrated with other analytic quality assurance methods. One problem is that the models do not clearly separate criteria that can be assessed automatically from criteria that require manual assessments. This separation would support planning the required analytic quality assurance activities, i. e. application of tools and manual reviews. In particular, most approaches leave unclear how the models should be used for constructive quality assurance. They do not explain how the required quality criteria can be communicated to the developers and how they support achieving them. We doubt that asking developers to design software systems to be *changeable* achieves the desired effect.

Summary Hierarchical quality models may provide a superficial overview on the multifaceted nature of software quality. Due to their inflexibility and unstructuredness as well as their lack of operationalizability, however, they do not suffice as definitional basis for a mature discipline of maintainability engineering.

3.2.2 Quality Modeling Frameworks

Several researchers proposed *quality modeling frameworks* that enable users to build customized quality models. These frameworks foremost alleviate shortcoming 1, the lack of customizability, but also address other problems discussed above.

The *Squid* approach [35, 177] is a method to plan and control product quality during development. It uses hierarchical quality models, similar to the approaches discussed above, for defining quality.

Users can either define their own quality models, use the ISO 9126 quality model as-is or customize an existing model like ISO 9126. The approach advances on ISO 9126 as it focuses on the assessability of the defined models. Therefore, it explicitly distinguishes between abstract quality *characteristics* and quality attributes that are defined as *measurable properties*. For such attributes (metrics) the quality model designer must specify target values that enable the actual assessment. Furthermore, the approach provides a tool to identify anomalous components based on the measurements.

The EMISQ [234, 235] method is based on the ISO standard 14598 for product evaluation [150]. It defines an approach for the assessment of internal quality attributes like maintainability and explicitly takes into account the expertise of a human assessor. The method can be used as-is with a reference model that is a slight variation of the ISO 9126 model or customizations thereof. Consequently, the method's quality model is very similar to the ISO 9126. It defines quality characteristics and exactly one level of subcharacteristics that are mapped to quality metrics whereas one subcharacteristic can be map to multiple metrics and vice versa. The metrics are defined by well-known quality assessment tools like *PC-Lint*⁴ and *PMD*⁵. Hence, they include not only classic numeric metrics but also metrics that detect certain coding anomalies. A notable property of the EMISQ method is that its reference model includes about 1.500 different metrics that are mapped to the respective quality characteristics. The approach also provides tool-support for building customized quality models and for supporting the assessments.

The authors of the *FS (Factor-Strategy)* approach also found that the classic hierarchical models exhibit an »obscure mapping of quality criteria onto metrics« and have a »poor capacity to map quality problems to causes« [201]. The FS approach aims to alleviate these shortcomings by mapping quality characteristics to so-called *detection strategies* instead of mapping them to raw metric values. These detection strategies are combinations of multiple metric values that are evaluated on the basis of absolute and relative thresholds. For example, the following detection strategy defines classes as being poorly encapsulated if their *number of public attributes (NOPA)* is higher than 3 and belongs to the top 10% of the values measured for this metric. Alternatively, a poorly encapsulated class is defined by a *number of accessor methods (NOAM)* greater than 5 if this value belongs to the top 10% of the values measured for this metric [201]:

```
PoorEncapsulatedClasses :=
    (NOPA, HigherThan(3) and NOPA, TopValues(10%))
    or (NOAM, HigherThan(5) and NOAM, TopValues(10%))
```

The FS approach supports using a pre-defined quality model, like the ISO 9126, to map the detection strategies to, or building a custom quality model in a similar hierarchical fashion. It provides a set of pre-defined detection strategies to be mapped to the quality model but also describes how detection strategies can be deduced from informal descriptions of design flaws [200].

The *NFR Method (Non-Functional Requirements Method)* [91] is an approach for the specification of quality requirements early in the development process. The central idea of the approach is to use *experience-based* quality models that are customized for each project. If suitable project-specific

⁴<http://www.gimpel.com>

⁵<http://pmd.sourceforge.net>

customizations are reflected back to the experience base, they can be reused in future projects. The quality models itself are of hierarchical nature and, although this is not fully clarified in [91], are apparently based on a *goal-graph* notation similar to the one used in [210] and [280]. The goal-graphs' nodes describe goals like *high source code quality* or *high code naming and commenting quality* and the edges express the goal interdependencies, e. g. *contributes positively* or *is required for* [280]. Hence, goal graphs are structured similar to the classic quality model but express a goal-oriented perspective typically used in requirements engineering. The NFR method uses multiple workshops to carry out the customization (or tailoring) of the quality models and to derive metrics as well as appropriate threshold values to quantify quality attributes.

In contrast to the NFR method, that clearly has a top-down perspective on quality, the model presented by Dromey addresses quality modeling in a bottom-up manner [92,93]. In his models, system entities (called *components*) like classes, variables or loops are first-class citizens that are associated with so-called *quality-carrying* properties. To describe the *quality impact*, quality carrying properties are classified as *correctness*, *internal*, *contextual* or *descriptive* where each of the categories is associated with a general quality attribute as found in the ISO 9126. An example that describes quality criteria for the entity *variable* is shown in Fig. 3.4. The entity *variable* is associated with the quality carrying property *assigned* as usage of unassigned variables may lead to unpredicted results at runtime. In Dromey's model »assignedness« belongs to category *correctness* which is defined to have an impact on *functionality* and *reliability*. Similarly, the example expresses that variables should have *self-descriptive* names as this property has an impact on *maintainability* and software *reuse*. In addition to the example model for the quality of source code, Dromey describes how the same approach can be used to specify the quality of requirements documents and, also, outlines a process for developing quality models.

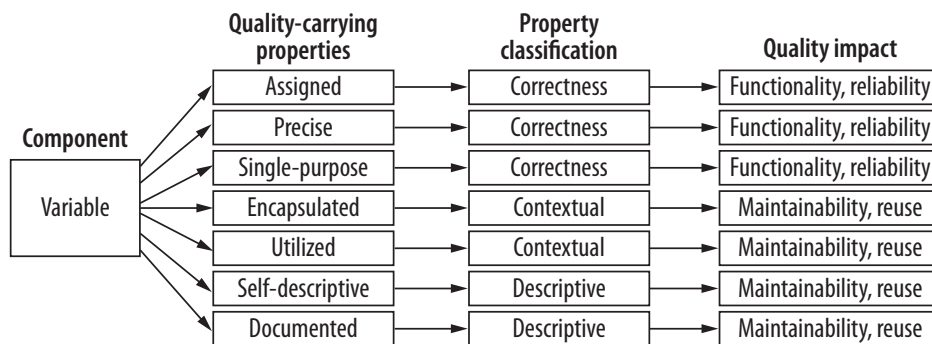


Figure 3.4: Extract of Dromey's Quality Model [93]

The QMOOD quality modeling approach [15] is based on Dromey's model and can be used to develop quality models to assess the quality of object-oriented designs. It extends on Dromey's work by identifying quality carrying properties for object-oriented design and associating them with a set of well-known as well as innovative metrics for object-oriented design. Based on these metrics, the approach provides a quantification of the quality carrying properties and describes how individual measurements can be aggregated to also quantify high-level quality attributes.

Discussion The quality modeling frameworks discussed above address a number of shortcomings identified for the classic quality models. However, the following discussion illustrates that none of the approaches remedies all deficiencies:

1. *Customizability.* All of the approaches are designed to support building customized quality models and, hence, satisfy the requirement of customizability.
2. *Assessability.* The author of the quality modeling approaches recognized that the non-assessability of previous models is a serious shortcoming and, hence, made assessability one of their central goals. The goal-graphs are an exception as they were explicitly not designed for evaluating existing systems but to »rationalize the development process in terms of nonfunctional requirements« [210]. For the other approaches, assessment is usually done on the basis of metrics used in conjunction with thresholds. Some approaches, e. g. EMISQ, FS, QMOOD, exclusively use product metrics like the *cyclomatic complexity* (see Sec. 3.4.1) while other approaches, e. g. NFR, focus on process metrics like »average number of hours required for system recovery« and Squid uses both types of metrics. Although, many quality properties, like naming quality, cannot be analyzed automatically, the FS and QMOOD approaches only take into account automatically assessable properties of a system. The EMISQ method is also based on automatic measurements but acknowledges that measurement results need to be thoroughly evaluated by a human expert.
3. *Rationale.* The approaches do a differently good job at providing rationales for the specified quality criteria, i. e. for explaining what impact a specific criterion has on software maintenance. Most approaches capture this only implicitly by explaining how a specific criterion relates to *maintainability*. While this provides an intuitive understanding, it makes not explicit how a criterion positively or negatively influences specific maintenance activities. For example, a typically construct modeled with Dromey's approach associates the *self-descriptiveness* of a *variable* with the quality attribute *maintainability*. While their might be an intuitive understanding for the impact of variables with self-descriptive names on software maintenance, the model does not capture this explicitly, let alone attempts to quantify the impact.

The EMISQ method is an exception as it supports the explicit documentation of the rationale for each metric included in the model. However, this is achieved by a prose description rather than in a structure manner.

4. *Structuredness.* For the concrete hierarchical quality models, the lack of a clearly defined decomposition criterion was identified as a major shortcoming. Unfortunately, the quality modeling frameworks hardly remedy this problem as they use a similar hierarchical structure as the classic models. This can be illustrated with a quality model for *efficiency* built with the NFR method [91]⁶ (see Fig. 3.5): Although the decomposition marked with *a* can also be found in ISO 9126, its semantics is unclear. What is the exact nature of the relation between *efficiency* and *usage time*? Decomposition *b* is even harder to understand, especially because it appears to be nonuniform. While *capacity* can be understood as a *quality-carrying* property in Dromey's sense, *type and position of devices* is hard to classify. Consequently, the meaning of the relations expressed by *b*₁ and *b*₂ are dubious. Decomposition *c*, however, appears to have a clearer

⁶NFR method publications do not provide examples for maintainability.

meaning. It describes the capacities of the different entities of the system and, hence, follows the structural decomposition of the system.

However, the central problem again is that the decomposition criterion for decompositions a , b , c is undefined and apparently nonuniform. Consequently, the semantics of the decomposition are unclear. Note, that the logical implication used in Boehm's model does not work here, either, as the edges are undirected and it is also not obvious how *resource utilization* would imply *type and position of devices*. Overall, it appears that the semantics of the relations between the model elements is not stronger than »has something to do with«.

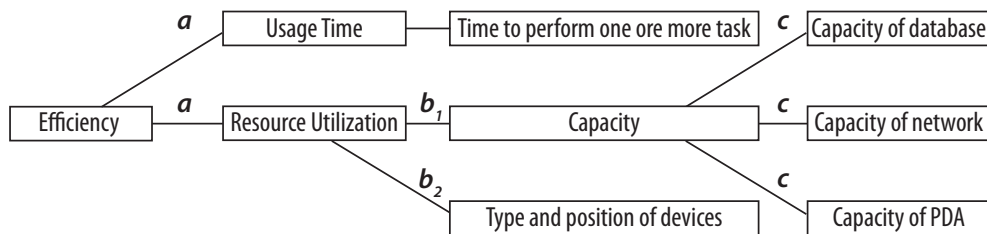


Figure 3.5: Extract of NFR Method Model for Efficiency [91]

While this example may represent an extreme case, the other approaches discussed above exhibit similar problems. Squid, EMISQ and FS use the same undefined decomposition as the ISO 9126. Dromey's approach is not directly affected by the problem as his models are, in fact, collections of unconnected fragments where each fragment has an comprehensible meaning. However, a hypothetical union of these fragments base on the quality attributes would exhibit the same shortcomings.

Like the classic quality models, these approaches do not explicitly define metamodels that would specify the syntax and semantics of the quality models. The Squid approach is one step ahead as it at least provides a semi-formally described metamodel [177]. An exception are goal-graph based approaches as syntax and semantics of goal-graphs are precisely defined. However, with their goal orientation these approaches have a strong focus on early development phases and do not define assessable models that could be operationalized for software maintenance. The lack of a clearly defined structure is especially problematic for customizable quality models. It makes extending a model very difficult, as it is unclear where a new quality criterion must be located in an existing model.

5. *Operationalization*. Similar to the classic quality model, the quality modeling frameworks do not provide ways for using the quality models for constructive quality assurance. It is left unclear how the quality models should be communicated to project participants, in particular to developers. This is all the more curious as e. g. Dromey states that the ability to provide »systematic guidance for building quality into software« is one of the central requirements for a quality model [92].

Due to their improved assessability, the quality modeling frameworks provide better means for being operationalized for analytic quality assurance. However, most of them do not explicitly explain how this can be done in practice. A central problem lies in the way the approaches deal with the collected metric values. To illustrate this one needs to be aware of the size of today's

software systems. Many software systems today have several thousand modules (classes) with a multitude of functions (methods) each. The quality modeling approaches define between a dozen and more than thousand metrics. Even if only module level metrics are take into account, an assessment of a whole system, hence, generates a collection of thousands or even millions of metric values. Clearly, a condensation of these masses is required to make them of practical use. This is usually achieved by aggregating metric values to higher levels. To achieve this, different strategies can be pursued: *aggregation w.r.t. quality decomposition* and *aggregation w.r.t. system decomposition*.

The problems current approaches exhibit regarding the first strategy are best exemplified by the extract of a Squid quality model shown in 3.6. For each leaf quality characteristic, the model defines one or more metrics for quantification. However, it remains unclear if and how the individual metric values should be combined, e. g. can *simplicity* be quantified by combining the measured values for *average module size*, *number of modules* and *cyclomatic complexity*? Furthermore, can *modularity* be quantified by combining *cohesion*, *coupling* and *simplicity*? It is to be noted, that it is not only unclear which aggregation operator would be appropriate but that the metrics value are on different scales and even of different scale type. This further complicates a sensible aggregation of data (see Sec. 3.4.1 for further discussion).

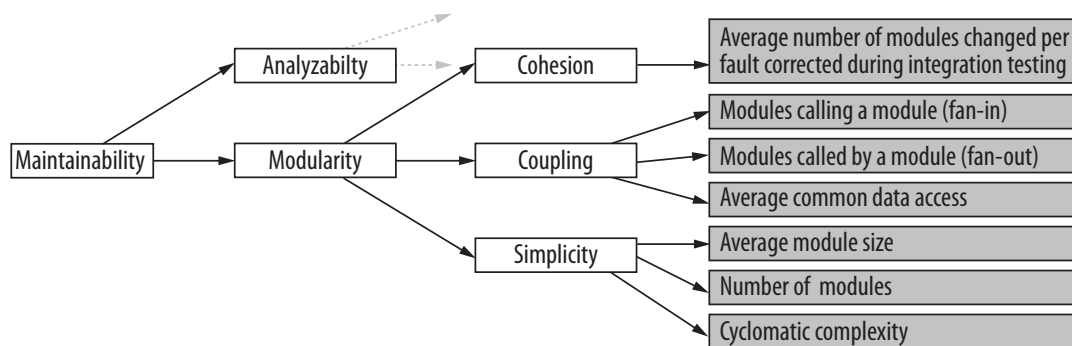


Figure 3.6: Extract of Squid Model [35]

Consequently, Squid and other approaches do not discuss aggregation at all. Hence, for assessments the hierarchical structure on top of the metrics serves only explanatory purposes. In contrast to his, the QMOOD method uses weighted sums to aggregate values. For example, it defines *extendibility* as:

$$\text{Extendibility} := 0.5 \cdot \text{Abstraction} - 0.5 \cdot \text{Coupling} + 0.5 \cdot \text{Inheritance} + 0.5 \cdot \text{Polymorphism}$$

where the properties are measured as shown in Table 3.1. Firstly, the rationale behind this aggregation is unclear. Why are exactly these metrics used, why are they summated and how are the factors defined? Secondly, the technical soundness must be seriously doubted. Metrics *Abstraction* and *Coupling* count classes whereas *Polymorphism* counts methods and *Inheritance* defines a ratio. It is unclear how a sum of these different entities should be interpreted. Moreover, the different metrics work at different ranges. The normalization approach described

in [15] to deal with this difference is incomprehensible. Furthermore, normalization does not remedy the fundamental problem that the rationale of the aggregation itself is dubious.

Property	Metric	Description	Range
Abstraction	Average Number of Ancestors (ANA)	Average number of classes from which a class inherits information	{0, 1, ...}
Coupling	Direct Class Coupling (DCC)	Number of classes a class is related to by attribute declarations or message passing	{0, 1, ...}
Inheritance	Measure of Functional Abstraction (MFA)	Ratio between number of methods inherited by a class and total number of methods accessible by member methods of the class	[0 ... 1]
Polymorphism	Number of Polymorphic Methods (NOP)	Number of methods that can exhibit polymorphic behavior	{0, 1, ...}

Table 3.1: Metric Definitions for the QMOOD Method [15]

A more sound approach is proposed by the EMISQ method. It translates metric values or rule violations to an ordinal scale with values *ok*, *critical* and *very critical* and uses the median to aggregate these *ratings* to higher levels.

The situation is equally bleak concerning the aggregation w.r.t system structure as none of the approaches details on this problem although some authors published such aggregated values. For example, the authors of the QMOOD method present a table of system-level metric values although they do not explain how their class-level metrics have been aggregated [15]. Obviously, one can use the arithmetic mean to deduce such values. However, this is not only doubtful w.r.t. to the scale type of the base metrics (see Sec. 3.4.1) but also of questionable utility as it does not help to identify quality defects.

Summary Quality modeling frameworks alleviate the inflexibility of the classic quality models and, to some extent, address their inaccessibility. However, the fundamental flaws concerning their structure and the ability to aggregate metric values still hamper their application as well-founded basis for maintainability engineering.

3.3 Constructive Approaches

There is a multitude of approaches for constructive quality assurance geared towards software maintenance. They range from personnel aspects like developer trainings over the usage of tools like configuration or change management systems to process models that define high-quality maintenance processes. The benefit of these measures is widely-acknowledged although there is an ongoing discussion about the relationship between process and product quality [178, 287]. However, none of these approaches entails an explicit specification of quality criteria. Rather, it is assumed that the usage of a certain tool or the application of a specific process aids software maintenance activities in general.

Concerning constructive approaches, this thesis focuses on methods that serve the purpose of communicating an explicit specification of quality to software engineers to support them in building

high-quality systems. Consequently, the following discussion of the state of the art focuses on existing methods for this purpose:

- *Guideline documents.* Guideline documents describe quality requirements and explicitly explain what developers should do or should not do to satisfy them.
- *Developer trainings.* In developer trainings an experienced trainer communicates quality requirements and ways to achieve them to the developers.
- *Language-characteristics.* Some quality requirements are so fundamental that they were respected by the designers of programming and modeling languages. For example, the Java language does not support unconditional branches (GOTO) as they support writing unstructured code [88]⁷. Hence, developers are constructively supported to avoid such pitfalls.
- *Tool-integrations.* Additionally, developers are supported by warnings that compilers generate for dubious use of language constructs. Nowadays, modern integrated development environments (IDEs) advance on this by integrating numerous checks for quality aspects that are reported to the developer in real-time.

Although doubtlessly important, developer trainings and programming language characteristics are beyond the scope of this thesis. Quality checks integrated in development tools reside on the borderline between constructive and analytic measures. While they can be perceived as a constructive measure, they are, in fact, more of analytic nature since they *analyze* already existing artifacts; although the time between artifact creation and analysis is infinitesimal small. Hence, the following sections focus on *quality guidelines*.

Today, most companies use guidelines to define quality criteria and to communicate them to the project participants [272]. As their focus is mostly on code, they are also called *coding standards* or *coding conventions*. They are introduced in nearly all standard software engineering literature and their usage is recommended by several international process standards, e. g. the IEEE as well as the ISO standard on software maintenance [145, 159] and the IEEE standard for software quality assurance plans (SQAPs) [146]. Some development methodologies like *eXtreme Programming* even made them one of their central principles [259].

Guidelines have been published for nearly all existing programming and modeling languages. Examples are standards for C [209, 275], C++ [135], Java [208], Fortran [207], ADA [249] and Matlab/Simulink [95, 196]. In addition, there are countless company-specific guidelines that are either developed individually or on the basis of publicly available ones. In some domains there exist guidelines that function as quasi-standards, e. g. the MISRA C guidelines in the automotive domain [209].

These standards vary in size and scope. For example, the Fortran standard [207] has 6 pages and exclusively focuses on source code whereas the C++ standard [135] has 88 pages and also includes documentation aspects. But foremost, the guidelines differ in their intent. While simple coding conventions mainly aim at consistency of code artifacts, more advanced guidelines aim at improving the quality of all development artifacts. Moreover, the guidelines emphasize different quality attributes. The MISRA guidelines for C have a clear focus on reliability, the guidelines discussed in [217] are meant to improve security and other guidelines stress portability [207, 275]. In general, however,

⁷In fact, the Java keywords *break* and *continue* support a constrained version of the GOTO jump [124].

the main concern of coding standards is readability and, hence, maintainability. For most programming and modeling languages this is foremost done by constraining the ways developers can use the language, e. g. by prohibiting the use of unconditional branches, and by presenting best-practices.

Interestingly, there is very little contribution from the research community to the topic of guidelines or coding standards as they are used in industry. While their existence and wide usage is recognized in most publications, there is hardly any work that deals with the characteristics, development and application of quality guidelines. Exceptions are [217] that describes how guidelines can be specifically developed for security issues and [189] that surveys the application of coding standards in programming courses.

Discussion For the purpose of this thesis, the discussion of guidelines distinguishes between definitional and constructive aspects, i. e. the quality criteria defined by the guidelines and the way they are communicated. Concerning the definitional aspects, the issues discussed for quality models also provide a well-suited framework for a structured discussion of the advantages and shortcomings of quality guidelines.

- *Customizability.* Guidelines are of a less rigid nature than (standardized) quality models and have been customized by software developing organizations ever since. However, this is usually done in an ad-hoc manner as there are no tailoring concepts defined for the guidelines.
- *Assessability.* Like quality models, guidelines sometimes define rules whose adherence is hard to assess. An example is the rule that require program entity names to be *self-descriptive*. While desirable, this rule is not concrete enough to be assessed [79]. However, due to their explanatory nature, guidelines in general are considerably more concrete than the classic quality models and, hence, better suited for assessments.
- *Rationale.* Guidelines often fail to motivate the required practices at all or provide only very generic explanations. For example, the Java Coding Conventions correctly demand that every *switch statement* should include a *default* statement but do not explain why [208]. The modeling guidelines published by the MAAB only annotate each guideline rule with one or more generic term like *readability* or *simulation* [196]. Well established and widely acknowledged guidelines, e. g. [135,209], clearly advance on this by providing detailed explanations for nearly all presented rules.
- *Structure.* The structure of guidelines is usually defined by classic document entities like chapters and sections only, although more advanced guidelines use templates to improve consistency [196]. Regarding the decomposition (chapter structure) most guidelines do not respect a clear decomposition criterion. Rather, they arbitrarily mix a decomposition based on language constructs (variables, functions, classes,...) with a decomposition based on quality attributes or some other method of decomposition based on arbitrary topics of interest.
- *Operationalization.* Next to the constructive aspects discussed below, guidelines are often not followed simply because it is not checked if they are followed or not [188]. This is all the more unfortunate as for some guideline rules compliance could be assessed automatically. However, there rarely is an explicitly defined connection between the quality guidelines that define quality and the analytic approaches that assess quality. An exception are the MISRA C guidelines as

multiple vendors offer tools that explicitly analyze »MISRA conformance« and cover a majority of the rules defined in the standard.

Concerning the constructive aspect of quality guidelines, it must be said that guideline documents are a well-suited method to communicate quality criteria to developers as their document form is the form developers are most familiar with. However, guidelines often do not achieve the desired effect as developers read them once, tuck them away at the bottom of a drawer and follow them in a sporadic manner only. Our experiences is that this is caused not only by the lack of rationale present in the guidelines but also by an insufficient tailoring of the guidelines for different target groups (see 6.1). For example, expert developers do not want to use guidelines that explain beginners' issues and require less additional explanations. Beginners or new project participants, however, cannot work efficiently with a guideline that is too condense. Moreover, guidelines often contain many rules on certain subsets of a language or framework that a developer or team does not use. Hence, these parts of the guideline are considered irrelevant and disturbing.

As guidelines do not have a properly defined tailoring mechanisms, they are either rejected due to these reasons or customized in an ad-hoc per-project or even per-developer manner. Because guidelines change or extensions are hard to distribute in such a situation, this inevitable leads to a number of inconsistent guideline documents. Since guidelines are often the only specification of quality criteria in the context of maintainability, the consequence is, in fact, different quality requirements within the same company or project.

Summary Quality guidelines are an integral part of maintainability engineering as they are needed to convey quality criteria to developers. However, as a definitional device they suffer from similar shortcomings as hierarchical quality models and as constructive device they are too inflexible in their current form. These deficiencies need to be addressed to make them work as an effective tool in constructive quality assurance.

3.4 Analytic Approaches

Even if successfully applied, constructive approaches are typically not restrictive enough to categorically rule out violations. Hence, analytic approaches still need to be used to ensure that the specified quality criteria are satisfied. The following sections discuss the state of the art of different analytic approaches.

3.4.1 Software Metrics

For several decades, software metrics have been used, amongst others, to measure system size, their quality and developer productivity. After the first software metric, *lines of code (LOC)*, was introduced in the 1960ies, more than 1,000 software metrics were proposed and discussed in more than 5,000 research papers [306]. This thesis does not propose any new software metrics and, hence, does not directly contribute to the field. However, software metrics are discussed here as they are the main mechanism for the definition of measurable quality attributes. They thereby provide the link between

the definition of quality and its assessment. Moreover, this discussion is required as it is sometimes difficult to distinguish advanced metric approaches from the quality models discussed above.

The following sections give a short introduction on measurement basics and then discuss software metrics in general as well as metrics that are specifically designed for software maintenance. The discussion has a focus on *product* metrics (as opposed to *process* metrics) as these are central for the contribution of this thesis. For a discussion of software metrics in general, see e. g. [63, 105, 116].

Measurement Basics As an exhaustive discussion of measurement theory is beyond the scope of this thesis, the following introduction to measurement basics is limited to the terms required for the discussion of the presented software metrics.

The IEEE defines a software metric as a »quantitative measure of the degree to which a system, component, or process possesses a given attribute« [143]. More formally, a metric or measure⁸ is a mapping M from an empirical relation system $\mathcal{E} = (E, R_E)$ to a formal relation system $\mathcal{F} = (F, R_F)$ where E are the entities of the real world, R_E their relations and F are formal entities and R_F their relations. For example, the LOC metric can be seen as mapping from the empirical relation system $\mathcal{E}_{\text{LOC}} = (\text{set of all programs}, \{\text{is longer than}\})$ to the formal relation system $\mathcal{F}_{\text{LOC}} = (\mathbb{N}_0, \{>\})$. The quality of a metric is judged by three criteria:

- *Objectivity.* Objectivity ensures that measurement is not influenced by the measurement environment.
- *Reliability.* Reliability ensures that measurement is not subject to measurement errors and is repeatable.
- *Validity.* Validity is the most complex criterion and is concerned with the question if the metric really measures what it intends to measure, e. g. are lines of code really a measure of system size? Measurement validity is a research discipline in itself and provides more detailed definitions for metric validity that go far beyond this informal description [50, 282]. However, for the purpose of this thesis, it is sufficient to restrict the discussion to the two following aspects of validity⁹:
 - *Content Validity.* Content validity describes the degree to which a metric accurately represents *all* aspects of the construct it attempts to measure. Hence, »content validity requires an inclusive definition of the domain of interest« [69]. For example, a metric LOC_A that measures the size of software systems by counting only the lines of code that include the letter »a« would not be considered content valid as major parts of the system are not taken into account. Content validity is the most important requirement for software metrics used in the context of quality assessments. Only if a metric accurately and completely captures the construct it attempts to measure, it can be reliably used to assess it. A metric that, without a detailed analysis, »appears« to be content valid, is often called *face valid* [282].

⁸In software engineering the term *metric* is commonly used for what other disciplines call a *measure*. Hence, this thesis uses the two terms synonymously.

⁹Different sources use inconsistent classifications of the types of validity. For example, some sources view *content validity* as an aspect of *construct validity* [282] while others regard them to be separate concepts [50, 69]. Hence, this thesis discusses the relevant types of validity without an attempt for a classification.

- *Predictive Validity.* Predictive validity is a subaspect of *criterion-related validity* and, hence, is defined with respect to an external criterion, i. e. the question is how well a metric predicts another aspect of the same entity. For example, the LOC metric is known to be a good predictor for the number of changes a component undergoes as larger components are typically changed more frequently. Note, that a metric does not have to be content valid to be predictive valid, e. g. the content invalid metric LOC_A from above can still be a good predictor for change frequency.

Another important aspect of measurement is the *level of measurement* or *scale type* of a metric. The level of measurement is a classification that is used to describe the nature of information contained within formal entities assigned to real-world entities. In 1946 Stevens identified four different types of scales [277]: *nominal scale*, *ordinal scale*, *interval scale* and *ratio scale*. The scales are ordered with respect to their expressiveness and allow differently powerful relations and operations to be used. For example, the nominal scale only allows to check values for equality, the ordinal scale allows to order values, the interval scale allows to sum values and the ratio scale allows multiplication. Consequently, the different scale types support different statistical operations, e. g. the nominal scale supports the mode, the ordinal scale the median and the interval scale the arithmetic mean. For software metrics, the level of measurement is important since »unless we are aware of the scale types we use, we are likely to misuse the data we collect« [229].

General Metrics A good part of the widely-known software metrics are used to measure the size of a software system. Examples are the LOC, *number of statements*, *number of methods*, *number of classes* or the *Halstead Volume* [130]. Next to this, a number of different metrics to measure source code *complexity* were proposed. Examples are *Halstead Difficulty* [130], McCabe's *Cyclomatic Complexity*¹⁰ [204] and the family of complexity metrics presented by Basili [21]. Moreover, numerous authors presented metrics designed for specific paradigms, e. g. the metrics suite for object oriented design proposed by Chidamber & Kemerer [56]. While many authors claim that their metrics can be used to measure *software quality*, most of them do actually focus on the more specific topic of *fault-proneness*. Concretely, they advocate metrics as predictors to identify fault-prone components [41, 211, 238].

Discussion While the utility of metrics in general is without doubt, there has been fierce criticism regarding a number of different issues:

- *Validity.* Many software metrics were not properly validated before publication. Hence, their content and predictive validity has been questioned by multiple researchers [19, 69, 104, 173, 179, 229, 258, 262]. For the purpose of this thesis, it is sufficient to highlight the shortcomings in content validity as it is central for a metric's capability to define and assess quality:

Many metrics attempt to quantify constructs that are intuitively recognizable but poorly understood. The prime example for this is *software complexity* for which countless metrics have been proposed although there is an ongoing discussion on what actually constitutes software

¹⁰Some authors even consider cyclomatic complexity a size metric as it essentially counts the number of program branches [21].

complexity [71, 123]. We share Fenton's believe that this attempt to circumvent a lack of understanding by providing a quantitative measurement is »one of the most common failings in software metrics work« [104]¹¹.

One underlying problem is that *measurement objectives* are not always clearly defined. In particular, it is left unclear if measurement activities aim at an *assessment* of a property or at a *prediction* [104]. As these objectives require different validity properties, it is hard to judge the quality of a proposed metric if the objective has not been clearly stated. With its vague definition of measurement objectives, Chidamber & Kemerer's suite of metrics for object oriented design is only one example [56]. A counter example is [246] that uses structural metrics to predict maintenance efforts and demonstrates how such a metric can be thoroughly validated.

The lack of content validity and clearly defined measurement objectives can be seen as a symptom of the broader phenomenon of »indiscriminate measurement« where metrics are designed and applied in a bottom-up manner, i. e. their rationale and intent is explained only after the measurement was carried out. Due to the lack of rationale it is unclear in which way system properties influence software maintenance. This makes it hard to convey measurement findings to developers.

Another consequence of »indiscriminate measurement« is the excessive focus on aspects that can be easily measured instead of aspects that are important to measure. As automatic measurements are significantly easier than manual measurement, this lead to a strong focus on things that can be measured automatically. Since many important quality factors of a software system, e. g. the naming of its identifiers or the appropriate use of data structures and algorithms, cannot be analyzed automatically, many *important* quality aspects are not covered by metrics.

- *Measurement Theory.* Multiple authors found that applications of metrics often do not respect the fundamental rules of measurement theory. In particular, it has been criticized that the metrics' *level of measurement* is not respected when applying transformations or other operations [104, 179]. One prominent example, is the application of the arithmetic mean for ordinaly scaled values that is widely used but invalid according to measurement theory. While it has been argued that this critique is at least partly based on a too strict interpretation of measurement theory in the context of software engineering [40], it remains true that a well- founded application of software metrics requires a more thorough observance of measurement theory.
- *Scope.* Another shortcoming of today's product metrics is their limited scope. While it has long been noted [247] that measurements should take into account pre-implementation artifacts like design documents, the majority of metrics is still dedicated to source code only. In theory, there are a number of metrics for design, in particular for object-orientation. However, they are often applied only in the implementation phase as the majority of metric tools works on code artifacts (see Sec. 3.4.4). Next to the neglected artifacts from early development phases, there is a number of implementation artifacts that are covered by metrics rather poorly. Examples

¹¹Other disciplines suffer from similar problems. For example, psychologists have developed measures for *intelligence* for more than 100 years although no commonly agreed on definition of intelligence exists. In 1923 E. Boring proposed to define intelligence as »intelligence is what the tests test« [39]. He thereby narrowed the concept of intelligence, but did so *deliberately* to support a substantiated discussion of intelligence tests. In software engineering, however, it appears that some researchers blindly accept software complexity being defined as »complexity is what the cyclomatic complexity measures«.

are, code written in domain-specific languages, configuration files, build scripts and, foremost, documentation.

Maintainability Metrics In addition to the general software metrics discussed above, a number of researchers presented metrics that were specifically designed to support software maintenance or proposed ways of applying general metrics for this purpose.

The earliest work on metric known to us that explicitly discusses maintainability aspects is Rubey & Hartwick's paper from 1968 [253]. This is one of the papers that is at the borderline between the quality models as discussed above and software metrics. For different quality characteristics it defines criteria and associates them with source code metrics whose values are between 0 and 100. For example, it defines criterion »the program logic is as simple as possible ($M_{6.2}$)« as one factor of *modifiability* and proposes the following metric to measure it:

$$M_{6.2} = \frac{100}{n} \sum_{i=1}^N \frac{F_i}{R}$$

where n is the number of instructions in a program, F_i is the number of programmer-accessible registers free after the i -th instruction and R is the total number of programmer-accessible registers.

In 1984 Berns presented a method for measuring the maintainability of Fortran programs [29]. As he assumes that maintainability can be equated with program difficulty the metric could be actually considered a complexity metric. However, it is specifically designed to aid software maintenance and is not meant to predict fault-prone components or steer testing efforts. The approach essentially assigns different Fortran language constructs with different weights and calculates the difficulty of a program by adding these weights. The approach considers interaction between program instructions that are believed to increase program difficulty. For example, the weight of assignment $A=B+C$ depends on the variable types and the scope of the variables defined earlier in the program. Furthermore, Bern's approach adds additional weights for »poor usages of Fortran«, e. g. the definition of a variable that is never used. His approach can be seen as an early predecessor to the *violation checkers* discussed in Sec. 3.4.4.

The best-known maintainability metric is the *maintainability index* proposed by a number of authors around Paul Oman in multiple publications [8, 61, 62, 226]¹². The maintainability index (MI) is defined as¹³:

$$\begin{aligned} \text{Maintainability} = & 171 - 3.42 \cdot \ln \text{avgEff} \\ & - 0.23 \cdot \text{avgECC} \\ & - 16.2 \cdot \ln \text{avgLOC} \\ & + 50 \cdot \sin \sqrt{2.46 \cdot \text{perCM}} \end{aligned}$$

¹²On the website of the Software Engineering Institute that previously advocated the maintainability index, it has recently been flagged as »legacy«.

¹³There are a number slight various of the MI. This version is cited from [62]

where *avgEff* the average Halstead effort per module [130], *avgECC* is the average extended cyclomatic complexity per module¹⁴, *avgLOC* is the average lines of code per module, and *perCM* is the average percent of lines of comments per module. This formula was derived using regression analysis where the dependent variable was the maintainability index and the independent variable a subjective assessment of maintainability that is not described in detail in the paper. It was later on adjusted to counter certain unwanted effects. For example the sine is used to provide a ceiling for the effects that comment lines have on the index. The authors do not specify concrete thresholds to interpret the index but give the following »rule of thumb«: »Components above the 85 maintainability index are considered highly maintainable, components between 85 and 65 are moderately maintainable, and the components below 65 are ›difficult to maintain«. « [62].

In [298] classic and object oriented metrics are used as predictors for change-proneness of components in order to support maintenance activities. Using Pareto analysis they found that a number of metrics, e. g. *coupling between objects (CBO)*, are good predictors for the number of changes a component undergoes where others like *depth of inheritance tree (DIT)* are not. However, none of the metrics was able to identify components with a high *change density*, i. e. number of changes with respect to component size. Consequently, the metrics under investigations were mainly identified as good »predictors« for component size.

Recently, Buse & Weimer presented a new metric for measuring the readability of source code which is generally considered important for software maintenance [48]. They performed a study where 120 participants subjectively rated the readability of 100 code snippets. They found that there is a »significant but not overwhelming« agreement on what readability is. They derive a readability metric based on a set of simple metrics, e. g. *line length*, *identifier length*, *number of spaces*. This is done using a machine learning approach as simple methods for establishing correlations are considered insufficient due to strong interactions between the basic metrics. They found that this metric is better in judging readability than the average human judge and that the metric is strongly correlated with change frequency and fault-proneness. As an additional result, they report that lengths of program identifiers does not influence the readability of a program.

Discussion The same criteria discussed for general metrics also apply for the maintenance-specific metrics:

- *Validity*. As attempts to define a content valid metric for program complexity failed so far, it is not surprising that metrics for maintainability, that is usually considered to subsume complexity, are not content valid either. In fact, most of the proposed metrics are not even *face valid* since even superficial analysis reveals that important aspects are not considered. For example, the maintainability index considers comment lines but does take into account the content of the comments itself. Similarly, the readability metric proposed in [48] integrates program identifiers but only analyze their length rather than desired properties like *self-descriptiveness*. In both cases, the desire for automatically measurable properties is believed to be the reason for these shortcomings. Another example for a metric that obviously captures only part of the construct it aims to measure, is Rubey & Hartwick's simplicity measure described above.

¹⁴It is unclear to which variation of McCabe's cyclomatic complexity the authors refer to. Presumably they use to a variation that takes into account Boolean operators like *and*, *or* that occur within branching conditions.

- *Measurement Theory.* As with the general metrics, work on maintainability metrics is often sloppy with respect to measurement theory. For example, it is not clear in how far the different scale types of the metrics used for the maintainability index have been considered and, particularly, it is unclear what the scale type of the maintainability index is. Berns does not explicitly discuss this problem but his maintainability assessment method acts more careful as it only adds weights [29].
- *Scope.* Most maintainability metrics focus exclusively on source code and, hence, do neither take into account artifacts of early development phases nor non-code artifacts of the implementation phase, e. g. documentation.

Summary Software metrics are an integral part of maintainability engineering as they serve the quantification of quality criteria and, hence, aid the unambiguous definition as well as the assessment of quality requirements. However, the above discussion shows that, to be applied effectively, metrics need to be supported by well-founded definitional measures in order to prevent indiscriminate measurement.

3.4.2 Metric Methodologies

In response to the indiscriminate application of software metrics, several researchers proposed *metrics methodologies* or *measurement methodologies* that support a substantiated application of metrics. Examples are the metrics program described by Grady & Caswell [125], the measurement approach defined by the SEI [223], the *ami* program [73] and, most notably, the *Goal-Question-Metric (GQM)* approach defined by Basili & Rombach [18]. As GQM is by far the most widely-used approach and the other approaches do not significantly advance on it, the following discussion focuses on GQM.

GQM is a »systematic approach for setting project goals [...] and defining them in an operational and tractable way« [18]. To apply GQM one needs to define project *goals* and refine them into quantifiable *questions*. For quantification, these questions are associated with one or more *metrics*. The approach supports the clear specification of goals by providing additional structure through the following four goal dimensions:

- *Object* — What is being examined?
- *Purpose* — Why object is being examined?
- *Issue* — Which attribute of the object is being examined?
- *Viewpoint* — From which perspective is the object being examined?

An example concerning the processing of change requests is given in Fig. 3.7. The example defines one goal refined into two questions that are associated with three and two metrics, respectively.

GQM has been used in various academic and commercial contexts to define goal-oriented measurement programs. With respect to software maintenance the following applications of GQM deserve particular attention:

Goal	<i>Purpose Issue Object Viewpoint</i>	Improve the timeliness of change request processing from the project manager's viewpoint
Question 1 Metrics		What is the current change request processing speed? <ul style="list-style-type: none"> ■ Average cycle time ■ Standard deviation ■ % cases outside of upper limit
Question 2 Metrics		Is the performance of the process improving? <ul style="list-style-type: none"> ■ $\frac{\text{Current average cycle time}}{\text{Baseline average cycle time}} \cdot 100$ ■ Subjective rating of manager's satisfaction

Figure 3.7: GQM Example [20]

Rombach et al. report on the application of GQM to characterize software maintenance at the Software Engineering Laboratory (SEL) [248]. They used a set of seven goals and twelve questions to investigate different aspects of the software maintenance phase, e. g. the distribution of change requests with respect to maintenance types, the variations between different projects and the impact of specific product properties on the maintenance activities. They do not report on the applied metrics in detail. Next to an improved understanding of maintenance at the SEL a central result was, that the definition of a measurement program is an iterative process that requires continuous revision due to changes in the measurement environment and an improved understanding of the examined object.

A related study was carried out by Basili et al. [16]. The goals were to develop a better understanding of software maintenance releases and to derive a model that predicts effort of future maintenance releases. To achieve this, they developed a GQM model with three goals; two directed at analyzing the current process for maintenance releases and one directed at the effort prediction model for maintenance releases. The central result of the study is that a structured measurement of the status quo, considerably helped to develop the predictive model.

Recently, Goldschmidt et al. used GQM to compare different data persistency frameworks with respect to performance and maintainability [122]. They defined one goal for maintainability that is refined by five questions where each question is associated two to four metrics. They used typically scenarios like »getting accustomed to the framework« or »add an additional persistent class« to define their questions. For example, one question is »How big is the effort to conduct all changes within the scenario?«. Two metrics associated with this question are »time to conduct the change in minutes« and »amount of files and/or models that need to be touched«. Using GQM the authors were able to perform a thorough and plausible comparison of the persistency frameworks, although they themselves do not draw any conclusions on the suitability of GQM for their study.

Discussion & Summary GQM is a major advancement on the »indiscriminate measurement« criticized above as it defines a top-down measurement approach that calls for clearly defined measurement objectives, explicitly includes manual metrics and does not require complex concepts like maintainability to be condensed to a single value [18]. Hence, a GQM-based measurement is likely to include *relevant* measures and not only measures that are easy to collect.

However, GQM-based measurements usually have a strong focus on the *process* and seldom explicitly take the product characteristics into account. This is also true for applications of GQM that are directed at product attributes. For example, maintainability related goals are usually quantified with respect to the effort needed to perform a specific task [16, 122, 248]. Note, that this is fundamentally different from the approach proposed by the quality models discussed above. These models purely describe product characteristic but do not clarify how these influence the maintenance effort.

As effort reduction can be seen as final goal of maintenance related measurements, focusing on it is a sensible and promising approach. For a practice of integrated maintainability engineering, however, this strong focus on analysis makes GQM a suboptimal candidate since it does not directly support constructive approaches. Obviously, the GQM approach could also be (miss-)used to build measurement models that include the various product characteristics included in the classic hierarchical models. However, due to the fixed number of levels (goals, questions, metrics) and the lack of a clearly defined decomposition criterion that would inevitable lead to models that are subject to the same criticism as the ones discussed above. Instead, it appears to be sensible to combine the goal-driven approach of GQM with a refined quality modeling method to avoid »indiscriminate measurement« while still providing the fine-granular information required for constructive approaches.

3.4.3 Reviews & Inspections

Reviews and inspections have long been recognized as an effective technique for finding software quality problems [4, 22, 100, 101, 109, 117]. While inspections are usually associated with finding software *defects*, they have also been applied to identify issues that are relevant for software maintenance but do not directly affect the functionality or reliability of the system [45, 203, 227, 268, 283].

There are no clear-cut definitions for *inspections*, *reviews*, *walk-throughs* and *audits* although most authors attribute different levels of formality and rigor to these techniques. For the purpose of this thesis the terms *review* and *inspection* are used synonymously to describe all manual verification techniques that are based on reading system artifacts. Fig. 3.8 illustrates the inspection process. In this process an *inspector* or team of inspectors analyses a product document, e. g. a code file. To do so, he checks if the product document satisfies the specification given by the *source document* and if a given set of *rules* is respected. This process is supported by *checklists* and finally leads to an *issue log* that lists the identified quality defects. The underlying idea is that the product document was created on the basis of the source document and the creation process respected a set of rules. For example, program code is written based on a prose specification and the rules are given by coding conventions. Checklists support the inspection process by formulating the rules in a manner suitable for a step-by-step inspection process. It is important, that they »must ultimately be derived from the rules [...]« [117]. In practice, however, there sometimes is a less clear distinction between rules and checklists and rules are made explicit only through the checklists. Consequently, these checklists serve not only analytic purposes but, in fact, *define* quality. Although Gilb advises against this practice [117], there is a rich body of checklists for different kind of artifacts in different languages that get reused across organizations and even domains [45].

Discussion There is relatively little work on the application of inspections for the particular purpose of identifying maintainability issues. However, previous work suggests that it may be worth-

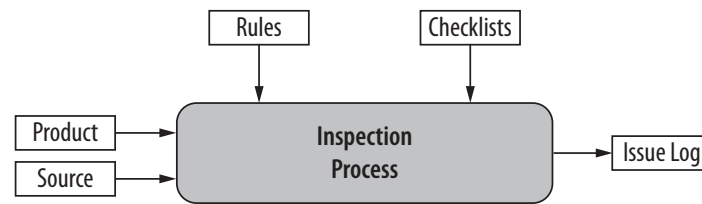


Figure 3.8: Software Inspection Process Inputs and Outputs [117]

while to not only include maintainability issues in »normal« inspections but specifically conduct inspections for this purpose [268]. Independently from the question if inspections focus solely on maintainability issues or consider them as *minor defects*, there is a number of shortcomings regarding the application of rules and checklists:

- *Definitional Aspects.* As rules and checklists are essentially definitions of quality, it is no surprise that they are affected by the similar shortcomings as the quality models discussed above. In particular, it has been noted that »checklist items should not be too general« [45] as this complicates analysis of conformance.
- *Consistency.* Another problem is that rules as well as checklists are usually stored as unstructured prose documents only. Hence, it is difficult to ensure completeness and consistency of checklists with respect to the rules. Furthermore, it is challenging to keep multiple checklists for different aspects consistent.
- *Automation.* In [45] Brykczynski states that »checklist items should not be used for conventions better enforced through other means (e.g., by the use of automated tools [...])«. This practice helps to reduce inspection efforts and reduce inspection omissions. However, this requires a structured approach that does not only clearly distinguish between automatically and manually inspected checklists but also ensures that each item is checked eventually. The unstructured format of today's rules and checklists does not facilitate such an approach.

Summary Inspections have been shown to be a very effective tool for the identification of quality defects and it appears promising to further increase their application for analytic purposes of maintainability engineering. However, the current practice of rules and checklists for defining quality criteria requires a more structured approach to be integrated in maintainability engineering.

3.4.4 Quality Analysis Tools

It has long been recognized that the high costs of manual inspections can and should be significantly reduced by using quality analysis tools [45]. Moreover, it was found that the enormous amounts of data generated during analysis can only be handled with appropriate tool support [247].

Consequently, commercial vendors as well as academia offer a plethora of quality analysis tools that credit themselves with the ability to accomplish this. The range goes from metric tools over violation

checkers and architecture assessors to application intelligence platforms and software quality dashboards. This thesis does not make a contribution to each individual tool category but focuses on tools that provide an integrated view of all collected quality data to support continuous quality control. Such tools are often called *quality dashboards* or *software cockpits*. However, due to the multitude and diversity of available tools, there is some confusion regarding the different types of quality analysis tools. To avoid a comparison of apples and oranges, Table 3.2 presents a categorization of the broad and heterogeneous tool landscape to support a structured discussion of their strengths and weaknesses. The table categorizes tools *Sensors*, *System Analysis Workbenches*, *Project Intelligence Platforms* and *Dashboard Toolkits*. Obviously, the boundaries of these categories are blurred. If one of the example tools can be argued to belong to more than one category, it is classified according to its primary use case.

	Sensors	System Analysis Workbenches	Project Intelligence Platforms	Dashboard Toolkits
Scope	quality analysis	quality analysis	project control & quality analysis	quality analysis & project control
Interaction Paradigm	autonomous	interactive	autonomous	autonomous
Usage Scenario	nightly-build, IDE integration	demand-driven	demand-driven & nightly-build	nightly-build
Analysis Object	development artifacts	code, architecture	metrics	project & process artifacts
Analysis Question	hard-wired	queries on system snapshot	queries on metric data	configuration of analysis topology
Result Represent.	lists	artifact-specific visualizations	lists, charts	list, charts & artifact-spec. visual.
Examples	JDepend, PMD, FxCop, NDepend, PC-Lint, Klocwork, JUnit	Sotograph, iPlasma	Hackystat, Team Foundation Server	ConQAT, XRadat, QALab, Sonar

Table 3.2: Categorization of Quality Analysis Tools

Sensors Sensors comprise anomaly detectors and metric calculators that perform fully automated analyses of development artifacts w.r.t. specific quality criteria. Due to their autonomous interaction paradigm that requires no user input, a common usage scenario is their application during automated nightly builds or as compile-time checkers in modern IDEs. The type of analysis question they answer is typically hard-wired. Analysis results are presented as tables or lists or as markers within an IDE. Examples¹⁵ include *JDepend*¹⁶, *PMD*¹⁷ and FindBugs [137] that perform guideline checks and bug pattern search for Java programs. *NDepend*¹⁸ and *FxCop*¹⁹ are representatives of comparable tools for the .NET platform. *PC-lint*²⁰ or *Klocwork*²¹ perform, amongst others, guideline checks and inspections of security vulnerabilities for C/C++. Another sensor example are *clone detection* tools that search source code for copy&pasted fragments of code [23, 172].

¹⁵If available, the publication that presented the tool is cited, otherwise the tool's homepage is referenced.

¹⁶<http://clarkware.com/software/JDepend.html>

¹⁷<http://pmd.sourceforge.net>

¹⁸<http://www.ndepend.com>

¹⁹<http://www.gotdotnet.com/Team/FxCop>

²⁰<http://www.gimpel.com>

²¹<http://www.klocwork.com>

System analysis workbenches System analysis workbenches support experts in the analysis of various development artifacts, including source code or architecture specifications, in order to answer analysis questions about specific quality aspects of a system, such as its architecture conformance or component structure. In contrast to sensors, they are interactive tools that are used on demand, during system inspection or review. They support interactive analysis by offering flexible system query languages and present results using specialized artifact- and task-specific visualizations, including graphs, charts and tree maps. *Sotoarc/Sotograph* [33], *iPlasma* [199] and *Moose* [96] are commercial respectively open source products for comprehension and reverse engineering of software systems. They provide analysis middle-ware in the form of a repository with a fixed metamodel into which systems under inspection are loaded for convenient access. *Sotoarc* supports modeling of a system's intended architecture and evaluation of the architecture conformance of its implementation. Furthermore, it can simulate restructurings to evaluate effects of architecture modifications. *iPlasma* offers, beside architecture analyses, a suite of object-oriented metrics and duplication detection. Furthermore, it provides a language to specify static analyses and a visualization framework and can thus be used as a basis for the development of further interactive analyses.

Project intelligence platforms Project intelligence platforms collect and store product and process related metrics of multiple sources to perform trend or comparative analyses. They are deeply integrated into a software development environment and collect metric data as it originates during development. Flexible query mechanisms often support generation of reports that show charts depicting the evolution of selected metric values over time. If these reports are generated in a frequent manner, they can serve as a project dashboard. Even though query creation has an interactive nature, project intelligence platforms operate autonomously and collect metric data without developer interaction. Project intelligence platforms are usually limited to metric values and allow ad-hoc queries on data from the project's past.

Hackystat [161] is an open source framework for collection and analysis of software development process and product data. It offers sensors that gather data during software development and transmit it to a central server for analysis, aggregation and visualization. Via a custom query language, reports can be specified to visualize, correlate or compare measured data. This way, hypotheses about the development process can be tested and impact of process changes on project performance can be evaluated. Although *Hackystat* offers several sensors that interface with static analysis tools that perform product quality analyses, its emphasis is on process measurement.

*Microsoft Team Foundation Server*²² is a commercial software product that aims to support collaborative software engineering. Besides source control and issue tracking functionality, it provides data collection and reporting services. Collected source control, issue tracking, build results, static analysis and test execution data is stored in a relational database system from which a report engine generates reports that monitor process metrics and visualize trends. The emphasis of the services is on process related data collection and reporting.

Dashboard toolkits Dashboard toolkits provide libraries of building blocks from which custom-made analysis dashboards, that collect, relate, aggregate and visualize sensor data, can be assembled by configuration. Building block libraries offer sensors for analysis of both product (e. g. code,

²²<http://msdn2.microsoft.com/en-us/teamssystem/aa718934.aspx>

architecture) and process (e. g. source control or issue management information) related artifacts. Additionally, blocks for presentation allow analysis results to be represented in a variety of formats, including general purpose lists or tables and specialized visualizations such as trees, graphs, charts or tree maps. In contrast to system analysis workbenches that are geared towards an interactive, on-demand explorative analysis of a system snapshot, dashboard toolkits are used for continuous quality analysis and monitoring of a project-specific set of questions. Their scope thus comprises both quality analysis and project control. In contrast to project intelligence platforms, that focus on operations on numerical metric values, dashboard toolkits can access sensor information on the level of development artifacts. They can thus exert greater control over sensor operations, which facilitates customization of analyses to project specific settings.

Dashboard toolkits can be differentiated by the degree of customizability they provide. Both *QALab*²³ and *Sonar*²⁴ offer pre-configured dashboards that present output from various sensors but offer very limited customization capabilities. QALab creates trend analysis charts displaying the evolution of the number of anomalies of a project. Sonar provides additional visualizations displaying aggregated single-project or cross-project quality information. Customization capabilities of both tools are limited to the choice of applied sensors. *XRadar*²⁵ is an open source code report tool for Java-based systems, which integrates XML reports calculated by different sensors via XSLT transformations. The results can be aggregated along the package hierarchy and also stored in the file system for plotting trend graphs of various metrics. Due to the expressiveness of XSLT transformations, aggregation and visualization of the imported analysis results can be configured more flexibly, than with QALab or Sonar.

Discussion While each tool is a valuable contribution to quality analysis, there still is a number of shortcomings that prevent a structured application of analysis tools to support maintainability engineering. The most critical shortcomings can be categorized as follows:

- *Integration.* Currently, the majority of tools operates virtually independently from definitional, constructive and other analytic quality assurance measures. Hence, assuring that the quality analysis tools measure what is defined by a quality model or quality guideline is tedious. Moreover, it is difficult to check if criteria that have not been evaluated by tools are duly taken care off in inspections.

The quality modeling approaches Squid [177], QMOOD [15], EMISQ [234], FS [201] discussed above provide partial tool-support to link analytic and definitional measures. Similarly, the software cockpit architecture presented in [28] discusses an explicit link to hierarchical quality models. However, the quality models used by the tools suffer from the shortcomings discussed above. Neither approach directly supports the integration of automatic and manual analyses although the EMISQ method allows for a human expert to evaluate automatic measurements.

- *Diversity.* The factors influencing product quality are diverse. Therefore a quality analysis tool should not be limited to a certain type of factors or artifacts it analyses. It must not only be able to analyze source code but should provide measures for other artifacts like documentation, models, build scripts or information stored in a change management system. As quality

²³<http://qalab.sourceforge.net>

²⁴<http://sonar.hortis.ch>

²⁵<http://xradar.sourceforge.net>

attributes can be discussed on many different levels, the tool should make no restrictions on the level of detail, the level of granularity, nor the type of analysis. It must, for example, be possible to analyze a source code artifact on representation levels as different as character stream, token stream, syntax tree or call graph.

However, most analysis tools today are limited to analyzing source code artifacts. Moreover, approaches that use a fixed metamodel to describe the analyzed objects, e. g. [33, 96, 199] are also limited to a specific level of abstraction.

- *Customizability.* Quality requirements are highly project-specific as the analyzed systems, the applied tools and processes, the involved technologies and the acting people differ. Even more so, these requirements are not constant but evolve over the course of a project. Hence, quality analysis tools must be highly customizable to support a project-specific tailoring of the analyses carried out and the way they are presented. Besides this, false positives generated by analysis tools are known to be a severe obstacle to the acceptance of quality analysis as they cause frustration among users. Here, customization is required to configure analysis tools to reduce the number of false positives [292]. For example, defects that impact code readability are not interesting for developers if they occur in generated code that is never read by humans. In such cases, analyses must be tailored to be aware of generated code.

Most tools offered today clearly lack the required customizability to adapt to project-specific settings and limit the number of false positives. For example, none of the tools known to us offers a mechanism for excluding generated source code from analysis if the code is mixed with manual code within the same file.

- *Autonomous Operation.* Tool supported assessments need to be carried out regularly (e. g. hourly or daily) to provide timely results and thereby prevent quality decay. To achieve this in a cost-efficient manner, analysis tools need to be able to work in a completely automated, non-interactive way. However, some of the most powerful analysis tools, e. g. Sotograph [33] or the *Insider* frontend for iPlasma [199], are designed as system analysis workbenches and as such require manual interaction.

Summary Due to the size of today's software systems quality analysis tools are a vital part of the analytic approaches applied in maintainability engineering. However, to make them work more effectively, they need to be better integrated with other analytic approaches and, foremost, with the definitional approaches that define what constitutes maintainability.

3.5 Summary

Today, the biggest obstacle towards a mature discipline of maintainability engineering are the approaches used to define maintainability and their integration with constructive and analytic quality assurance approaches.

The current approaches for specifying maintainability pale in comparison with the methods used for the specification of the functionality of a system. Requiring a system to have a *high maintainability* is comparable to specifying a CRM system by stating nothing more than that it »should support

customer relationship management«. Refining the requirement to a *high changeability* can be seen as analog to stating that the CRM system »should support CRM processes and store customer information«. To handle the growing complexity of software systems, model-based techniques have become more and more common to specify system functionality. In the same manner, model-based approaches should be used to specify maintainability. However, the current quality modeling approaches suffer from a number of shortcomings:

- *Assessability.* They do not define criteria for maintainability at a level that is suitable for an actual assessment. Hence, it is not possible to evaluate if a system complies to stated quality requirements or not.
- *Rationale.* They tend to omit the rationale behind the required properties of the system. This makes it difficult to describe impacts precisely and therefore to convince developers of the importance of the proposed quality criteria.
- *Structuredness.* They often use ambiguous decomposition dimensions which leads to inconsistent models and hampers the revelation of omissions and inconsistencies in these models.

Moreover, existing approaches are not firmly integrated in the software maintenance and quality assurance process. Due to this they cannot be directly used as basis for analytic and constructive quality assurance activities. On the one hand, it is unclear for existing approaches how their definitions of maintainability can be conveyed to all project participants so they can prevent maintainability defects from the beginning on. On the other hand, the role of the proposed approaches with respect to analytic quality assurance remains unclear. This is a precarious situation as it makes it difficult to use quality models as basis for a continuous quality control practice that counters the quality decay that software systems are known to undergo during their evolution.

To remedy these shortcomings, the following chapter presents a novel quality modeling approach that addresses the deficiencies identified for current modeling approaches and can also be tightly integrated with the software maintenance process.

»The measurement of quality is the price of non-conformance, not indexes.«

Philip B. Crosby

4 Defining & Controlling Maintainability

This chapter presents the main contribution of this thesis: A novel framework for defining, evaluation and improving *maintainability*. This framework includes a precisely defined metamodel for quality models as well as methods and tools to instantiate and operationalize the quality models for continuous quality control of long-lived software systems. To introduce this new framework, we describe the relevant processes for managing maintainability and explain the role of quality models in this context. We then formulate a set of requirements for quality modeling approaches that are based on the intended integration into the quality management processes as well as on the shortcomings of existing approaches discussed in the last chapter. We explain the underlying rationale of our approach, exemplify it with multiple examples and illustrate how it satisfies the stated requirements and thereby advances on previous approaches. After this we give a formalization of the quality metamodel (QMM) that constitutes the foundation of our modeling approach and explain how models based on it are operationalized for continuous quality control. The chapter concludes with a discussion of the benefits and drawbacks of the presented approach.

4.1 Maintainability Management

In Chap. 3 we argued that a major shortcoming of previous quality models is that they are not operationalized. They are expressed in prose and graphics only and accompany the development process in the form of documents but are not made an integral artifact that is tightly coupled with the quality assurance activities. To explain how this shortcoming can be overcome, a thorough understanding of the quality management process and its integration with the software maintenance process is necessary. Unfortunately, literature on quality management in software engineering and other disciplines uses a bewildering variety of different, often inconsistent, terms for different quality management activities. To avoid confusion, Fig. 4.1 illustrates the notion of quality management that underlies this thesis and clarifies the related terminology¹. The processes as well as the definitions of terms are based on international standards and recommendations [3, 143, 144, 147, 148] but also influenced by the quality management approaches developed by Juran [169], Crosby [67] and Deming [87].

Following the IEEE Standard 12207 [144] *quality management* is viewed as a supporting life cycle process for the primary life cycle process *software maintenance* that subsumes all coordinated activities to direct and control an organization with regard to quality. The quality management process is essentially based on the concept of *conformity*, i. e. the fulfillment of *quality requirements*. Hence, it is assumed that all quality requirements can be made explicit and that conformity can be assessed by comparing the actual *quality characteristics* of a product with the quality requirements. The core processes of quality management are:

¹To be consistent with existing terminology, the discussion of the processes generally uses the term »quality« although our considerations are limited to quality aspects relevant for software maintenance.

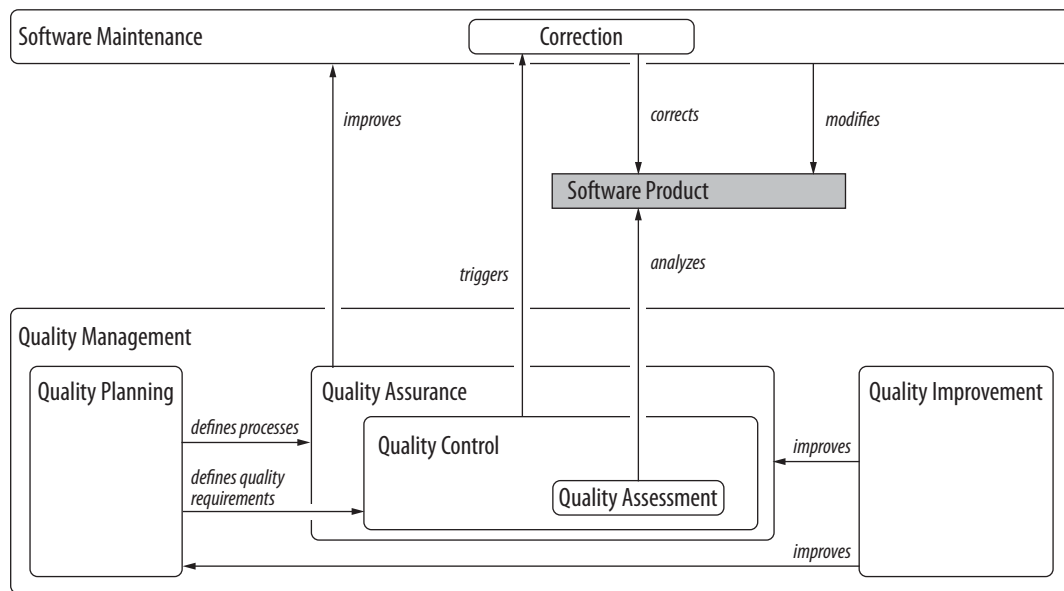


Figure 4.1: Quality Management Process & Terminology

- *Quality Planning.* The quality requirements of a product are defined by the quality planning process. Additionally, quality planning defines the quality assurance processes needed to control product quality and to enhance the primary life cycle process' ability to meet quality requirements.
- *Quality Assessment*². Quality assessment compares quality characteristics with quality requirements in order to evaluate conformity. Note that, according to Garvin [114], this quality management essentially adopts a *product view* as it assumes that conformity can be assessed by comparing product characteristics to requirements.
- *Quality Control.* Quality control is a process that consists of analyzing the actual quality of a product, comparing it to quality requirements, and taking necessary actions to correct the difference.
- *Quality Assurance.* If quality requirements are not communicated to the developers, the software maintenance process is very unlikely to generate conforming products. It is the responsibility of quality assurance to communicate quality requirements to the development process in order to enhance its ability to meet them. Quality assurance can additionally enhance this ability by improving the maintenance process itself, e. g. by providing appropriate tools. Hence, it is to be noted, that in contrast to frequently found definitions, quality assurance subsumes quality control but goes far beyond it as it helps to perform quality management in an efficient manner.
- *Quality Improvement.* Quality improvement is responsible for improving the quality assurance process and the quality planning process. It does not, however, improve the product itself.

²The ISO 9000 refers to this process as »inspection«. However, we use the term »assessment« to avoid confusion with the classic inspections discussed in the last chapter.

With respect to these processes, it can be clarified that the focus of this thesis is on *quality control* and the *quality assurance* activities that communicate the quality requirements to the developers in order to *prevent* quality defects. However, if appropriate, we also indicate how our quality modeling approach is integrated with other processes like quality planning.

Having established a precise definition of quality management for software maintenance, we can now explain how we envision a quality model to support the different processes. The interplay of the quality model, the main actors and their respective activities is shown in Fig. 4.2. For clarity's sake the relation to the above processes is not included in the figure but explained below.

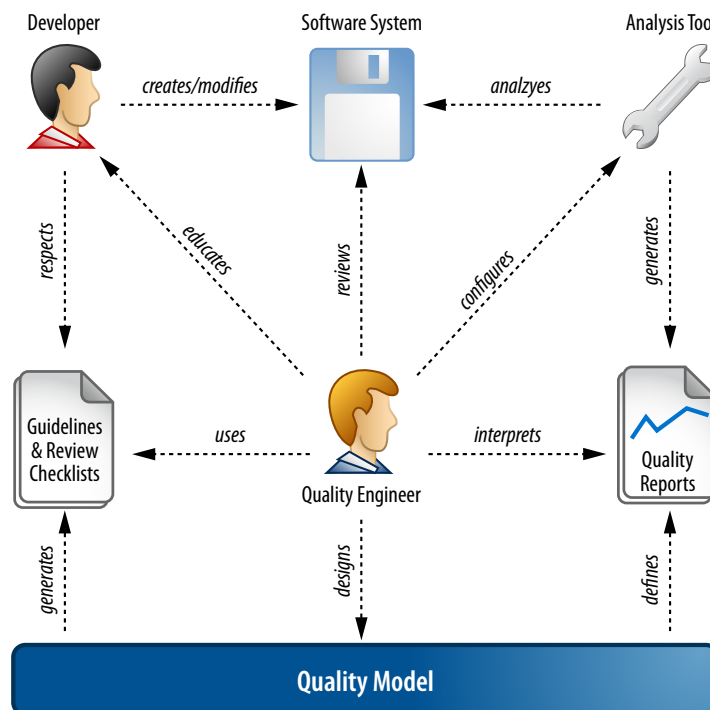


Figure 4.2: Operationalized Quality Model as Basis for Maintainability Management

A team or person responsible for the maintainability of software systems, the *quality engineer*, designs the quality model that explicitly captures the *quality requirements* used to determine conformity. Quality assurance is responsible for improving the maintenance process to enhance its ability to meet quality requirements. This is supported by *guidelines* that define what *developers* should do and what they should not do in order to improve the maintainability of the *software product*. Such guidelines are automatically generated from the quality model. The *quality control* process is based on manual *reviews* and the application of *quality analysis tools*. Manual reviews are supported by the automatic generation of review *checklists* from the model. The quality analysis tools generate quality reports that are tightly bound to the quality model and directly illustrate model conformity or non-conformity. Based on the review results and quality reports the quality engineer orders the developers to correct maintainability issues. Furthermore, he aids quality assurance by *educating* developers to improve maintainability in the long-term. As part of *quality planning*, the quality engineer adapts the quality model to match changing maintainability requirements.

In the proposed constellation the quality model assumes the role of a project- or company-wide quality knowledge base that centrally stores the definition of quality in a given context. As there is only a *single* instance of the model it guarantees a consistent definition of maintainability regardless of the number of quality engineers who work on it. As it presents a unifying, integrated view on the definition of maintainability it helps to identify missing criteria and thereby fosters completeness. Furthermore, the quality engineer is relieved from the tedious and inconsistency-prone work of writing quality guidelines and review checklists as they can be generated automatically from the quality model. Likewise, configurations for quality analysis tools can be derived from the model to ensure that analysis results reflect the criteria defined by the model.

4.2 An Activity-Based Model for Maintainability

The following sections give a detailed account on our quality modeling approach that supports the precise and unambiguous definition of maintainability which is a key requirement for maintainability management as outlined above. The design of the approach is based on the following list of requirements for quality models which in turn is based on the described way of operationalization and on the shortcomings of the existing approaches discussed in Chap. 3.

1. *Assessability.* The quality criteria defined by a quality model must be detailed enough to be *assessable*. Only quality criteria that are defined on a concrete, tangible level can be assessed during reviews and by analysis tools. High-level criteria like *changeability* are valid quality goals but do not offer means of assessment. In this context, assessment does not generally imply *automated assessments* since many important quality criteria inherently elude automatic analysis. An example is the quality of program identifiers, that needs to be assessed through manual reviews [79].
2. *Rationale.* A quality model must provide *justifications* for the quality criteria it defines, i. e. give a detailed account of the impact that specific quality characteristics have on software maintenance. In order to be respected by developers, a quality model must not only explain the *what* but also the *why*. Only quality criteria that are reasonably justified will be respected by developers whereas criteria ordered by decree are prone to be ignored [299]. This is also important if developers need to override certain conventions to solve an urgent problem. Only if they understand the consequences they can make informed the decision on the appropriateness of such a transgression.
3. *Structuredness.* A quality model describes a *decomposition* of the complex concept quality or, in this instance, maintainability. As the review of existing approaches in the last chapter showed, the factors influencing software maintenance are numerous and diverse. Since a quality model for a realistic system can easily contain several hundred individual factors, a quality modeling approach must provide a structure that allows to manage the plenitude of factors by defining the position of quality factor within the decomposition as unambiguously as possible. This not only supports the effective and efficient creation of quality models but also helps to reason about consistency and completeness of quality models.

Designing an approach that satisfies the requirements stated above comprises the following challenges: Firstly, it requires finding a metamodel that defines the legal model instances. The major

challenge in designing such a metamodel is to find a decomposition mechanism that is defined strictly enough to avoid ambiguities in model instances but still lightweight enough to be applied in practice. Our metamodel achieves this by clearly separating aspects that are typically intermingled in previous approaches and defining a decomposition that is exclusively based on the *is-a* and *part-of* relations that are well-known from various modeling techniques used in other contexts. Furthermore, the metamodel inherently ensures that the rationale of requirements described by quality models is made explicit and requires a definition of conformity to support quality assessments. Moreover, the metamodel is designed to not limit the scope of model instances, i. e. quality criteria may be expressed for software as well as infrastructure or other artifacts. By providing a formal definition of legal model instances, the metamodel serves as necessary basis for quality modeling tool support, e. g. quality model editors and guideline generators. The metamodel and its underlying rationale are presented in this chapter.

Secondly, to be of any worth such a metamodel needs to be instantiated to define maintainability in a specific project or company context. We present multiple instances of the model used in contexts as different as web application frameworks and model-based development with Matlab Simulink in the chapter on case studies (Chap. 6). Extracts of these models are used in this chapter to exemplify our quality modeling approach.

Thirdly, as maintainability needs be controlled in a continuous manner, the operationalization of the models is of paramount importance. In this chapter we propose different concepts for the operationalization of the quality models to generate quality guidelines and review checklists as well as to automate assessments of adherence to quality criteria. Specific applications of these concepts are discussed in the chapter on case studies (Chap. 6).

4.2.1 Modeling Rationale

This section describes the rationale that underlies the quality modeling approach presented in this thesis. While this rationale is based on economic considerations of software quality, we do not intend to describe a full-blown cost/benefit model for maintainability here. The reason for this is not only the limited scope of this thesis but also the current lack of understanding of how different quality factors do influence maintenance costs. Today, no one really knows the quantitative effect of a quality problem like code cloning for software maintenance. However, we detail on the economic considerations to motivate our modeling approach and, also, to pave the way for possible extensions of the model.

To satisfy the requirements stated above, a quality model needs to describe *maintainability* in a precise and unambiguous manner. But what is maintainability? How can it be defined, assessed and improved? As discussed in Chap. 3, there are numerous approaches to define maintainability in an *extensional* manner, i. e. by listing product attributes that contribute to maintainability. Due to the reasons discussed before, none of the approaches could eventually be established and none of them can be directly used as basis for an operationalized quality model. To build such a model we therefore chose to start from frequently cited definitions, that define maintainability in an *intensional* manner, i. e. by focusing on the effect maintainable systems have on the maintenance process. Examples are:

- *Maintainability: The effort needed to make specified modifications to a component implementation*³.
- *Maintainability is characterized by the average effort in staff-hours per system and maintenance task.* [246]

These definitions nicely illustrate that the desire for *high maintainability* is actually a desire for *low maintenance efforts* and thereby open up an economic perspective or, more precisely, a *cost* perspective. They define a system *A* to be more maintainable than another system *B*, if *A*'s average effort per maintenance task is lower than the *B*'s. A similar notion is also used in previous works that adapted GQM approaches to software maintenance related issues. In these works, maintainability related goals are also quantified with respect to the effort needed to perform a specific task [16, 122, 246, 248].

The direct applicability of these intensional definitions is limited as they can only be used to compare existing systems with similar maintenance tasks and known maintenance efforts. However, we will show that maintenance efforts are an ideal means to form the basis of a comprehensive quality model for software maintenance.

The Price of Non-Conformance To use this economic perspective for quality modeling we break down maintenance costs C_m as

$$C_m = C_d + \overbrace{C_c + C_{nc}}^{\text{Quality Costs } C_q}$$

where C_c are the *conformance costs*, C_{nc} are the *non-conformance costs* as discussed in Sec. 2.2.4 and C_d are *direct costs*, i. e. costs not related to quality, e. g. understanding of the change requests themselves. Following Philip Crosby's central insight that the »measurement of quality is the price of non-conformance« [67] we now define a system to be highly maintainable if its non-conformance costs are low and vice versa.

To illustrate this, we assume the following fictitious situation. Two software systems *A* and *B* implement exactly the same functionality but are not verbatim copies of each other; both were developed independently. The two systems are maintained by two organizations O_A and O_B which are 100% clones of each other (including staff, processes, tool, ...) and both systems undergo the exact same change requests. In this case, the direct costs C_d as well as the conformance costs C_c for both system are equal. Hence, a difference of the maintenance efforts for systems *A* and *B* is caused by differing *non-conformance costs* that are due to a difference in the maintainability of systems *A* and *B* (Fig. 4.3)⁴.

³SEI Open Systems Glossary (<http://www.sei.cmu.edu/opensystems/glossary.html>)

⁴In the PAF model the conformance and non-conformance costs are often viewed as being independent from each other. In reality, they are not. It is to expect that a high quality system has lower conformance costs and vice versa, e. g. because reviews can be performed quicker. However, for the sake of the above explanation we also subscribe to the independence of the two costs types and assume that differences in conformance costs that are caused by non-conformance are part of the non-conformance costs.

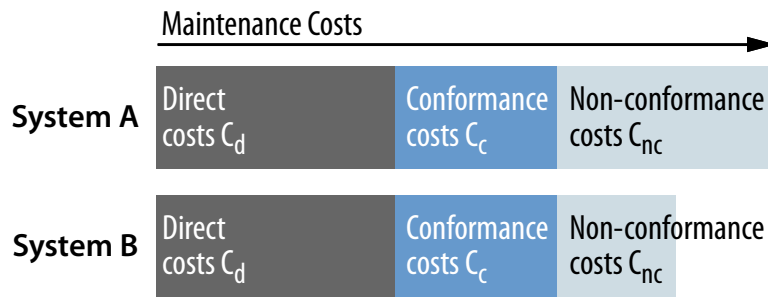


Figure 4.3: Lower Non-Conformance-Costs, Higher Maintainability

In the context of maintenance, we therefore propose to use quality costs as basis for the definition of quality. As a consequence, the model presented in this work does not strive for a definition of the abstract concept of *maintainability*, but describes the influence of the various contributing factors (cost drivers) on the *maintenance efforts*, or, more precisely, on the non-conformance costs. While this distinction may not look important at first sight, we will show that it is crucial in overcoming the vagueness of the term *maintainability* as well as its limitations in scope.

With respect to Garvin's five views on quality, one is tempted to categorize this way of addressing quality as the *value-based view*. However, there is a subtle but important difference. The value-based view, as defined by Garvin, considers the trade-off between quality and the cost (or price) required to build (or buy) quality products. The view proposed here does not take into account such a trade-off but simply defines quality through non-conformance costs, i. e. a product with low non-conformance costs has a high quality and vice versa. This is not to say that the trade-off highlighted by the value-based view would be irrelevant. Contrariwise, we are convinced that a thorough analysis of this trade-off is of paramount importance for economical software engineering. However, this requires a well-founded definition of quality, or maintainability in this case, first.

Focus on Activities We not only want to compare identical systems whose maintenance costs are known but build a model that defines maintainability by specifying concrete product properties. Hence, we need to find a structuring mechanism that allows the construction of quality models that are capable of describing all relevant criteria and their influence on the non-conformance costs. In Chapters 2 and 3, however, we saw that there is a large number of very diverse factors that influence maintenance costs. We hence need an appropriate means for untangling these factors.

Our solution to this problem is inspired by a technique called *Activity-Based Costing (ABC)* that was developed in the manufacturing industry during the 1980ies to remedy a serious accounting problem recognized at that time. The problem was that manufacturing companies that produced multiple products were unable to determine the cost of a single product as product line approaches, support operations, marketing activities and other overhead functions made it virtually impossible to trace expenses to specific products. To capture these non-product-specific costs, companies simply added the same amount of overhead cost to each product and thereby failed to notice that some products were highly profitable while others were actually disprofitable.

To solve this problem, Cooper and Kaplan [64] proposed the Activity-Based Costing (ABC) technique that is nowadays widely used in the manufacturing industry. The general idea behind ABC is to *trace* the costs for each product by determining the costs drivers instead of arbitrarily *allocating* costs to products. To achieve this, ABC proposes to use the *activities* that are carried out to find the actual cost drivers. Concretely, ABC advises accountants to identify all activities that contribute to the development, production, distribution and support of a product and then determine the cost drivers for each activity.

We are convinced that the same approach can be used to determine the drivers of maintenance non-conformance costs. Therefore, we propose to use maintenance activities as the major structuring mechanisms for quality models. Consequently, our quality metamodel treats maintenance activities as first-class citizen and defines models that explicitly relate cost drivers to activities. Activity-based methods have been discussed in the context of software quality before, e. g. by Jones [163] or Mandeville who postulated that »the costs of quality must be measured at an activity level« in a paper that unfortunately received little attention [197]. Similarly, previous work on software maintenance recognized the important role of maintenance activities [181,271,273]. For example, Granja-Alvarez & Barranco-Garcia use activities as basis for their software maintenance cost model and define the total maintenance cost as the sum of the costs for understanding, modifying and testing [126].

Nevertheless, maintenance activities were never explicitly used for modeling quality. Interestingly, many previous approaches, however, implicitly take them into account. For example, the well-known hierarchical quality models discuss maintenance activities in a somewhat disguised form. An example is given in Fig. 4.4 which shows the *maintainability* branch of Boehm's *Software Quality Characteristics Tree* [38]. The nodes in the gray boxes refer to activities whereas the uncolored nodes describe system characteristics (albeit very general ones). So the model could also be read as: When we *maintain* a system we need to *modify* it and this activity of *modification* is (in some way) influenced by the *structuredness* of the system. While this difference may not look important at first sight, we claim that the mixture of activities and characteristics is, at least partially, responsible for the unsatisfactory structure of existing models as this inhomogeneity inhibits the definition of a clearly defined decomposition criterion.

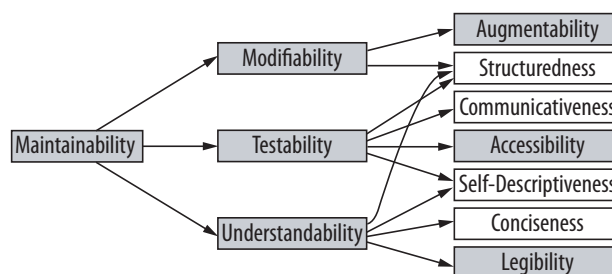


Figure 4.4: Software Quality Tree (cf. Fig 3.1)

As the actual maintenance efforts strongly depend on both, the type of system and the kind of maintenance activity, it should be obvious that the need to distinguish between activities and characteristics becomes not only clear but imperative. This can be illustrated by the example of two development organizations, where company *A* is responsible for adding functionality to a system while company

B's task is merely to fix bugs of the same system just before its phase-out. One can imagine that the success of company *A* depends on different quality criteria (e. g. architectural characteristics) than company *B*'s (e. g. a well-kept bug-tracking system). While both organizations will pay attention to some common attributes such as documentation, *A* and *B* would and should rate the maintainability of *S* in quite different ways because they are involved in fundamentally different *activities*.

4.2.2 Overview

This section gives a high-level overview of our activity-based approach to explain its basic concepts. To facilitate the introduction of the novel modeling approach, we start from the well-known hierarchical model shown in Fig. 4.4 and explain how our approach advances on it by means of suitable examples. After that, we will give a formal definition of the underlying metamodel.

A consequent separation of maintenance activities and system characteristics leads to a 2-dimensional structure as shown in Fig. 4.5. The figure represents a restructuring of Boehm's software quality tree where the maintenance activities are shown on top and the system characteristics on the left. The activities form a tree based on a *part-of* relation. Hence, *modification* can be considered a *subactivity* of *maintenance*. Activities *augmentation*, *access* and *reading*⁵ are depicted in braces only as Boehm's model does not describe characteristics that influence them. The matrix is used to illustrate the relation between the elements of the two dimensions. In this case, the relation is unweighted and does not explain *how* a characteristic influences an activity. We will later see how this can be refined to describe the actual nature of the influence.

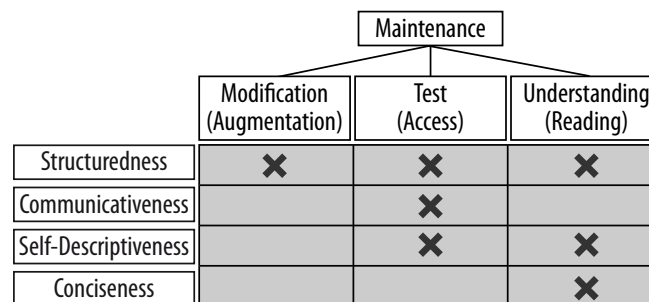


Figure 4.5: Separation of Activities and Characteristics

The separation of activities and characteristics addresses the lack of structuredness identified for previous quality modeling approaches as the relationship between the model elements is clearly defined: activities form a sub-/super-activity relation and characteristics are related to activities through their impact on them. This impact relation also addresses the lack of rationale exhibited by other quality modeling approaches as the explicit relation to a maintenance activity inherently explains *why* a certain characteristic is relevant for software maintenance. The structural separation aids reasoning about the completeness of the quality model as it can be matched to the existing maintenance process to ensure that all relevant activities are included. However, the same cannot be said for the

⁵We assume that Boehm uses the term *legibility* synonymously to *readability* although *legibility* is usually stronger connoted with presentational aspects of reading, e. g. typefaces, while *readability* refers to documents' content.

characteristics on the left. It still cannot be verified if all relevant characteristics are included or not. Moreover, the most important requirement postulated for quality models, their assessability, is still not satisfied as it remains unclear how e. g. *structuredness* or *conciseness* can be assessed.

We claim that the central problem is that most previous models attempt to describe these characteristics on the level of the *whole* system. However, real software systems are usually of significant size and consist of several thousand artifacts as diverse as programs, documents and models in many different languages. On a finer granular level this further extends to elements like classes, methods and variables. Obviously, characteristics like *structuredness* apply to many of them but mean different things for each of them. For example, the structuredness of a method is surely defined differently from the structuredness of an UML class diagram. Hence, it is inconceivable how a quality model that does not take into account these different types of artifacts can be used to comprehensively capture the relevant quality requirements.

Facts To overcome this problem we propose to include the various system artifact types in the quality model and make explicit their relation to characteristics and maintenance activities. This allows to decompose characteristics like structuredness that are hard to assess for the whole system into separate constructs that describe e. g. the structuredness of a method or of an identifier: [Method | STRUCTUREDNESS] or [Identifier | STRUCTUREDNESS]. We call these constructs *facts*. The part of a fact that describes the artifact type is referred to as *entity* and the characteristic is called *attribute*⁶. This separation allows to explicitly describe attributes of different artifacts types on a level fine-granular enough to achieve assessability. For example, we can define that a method is structured if it has only a single exit point and an identifier is structured if it matches a certain format. The separation of entities and attributes is similar to the one used in Dromey's quality model [92] which separates *components* from *quality-carrying properties*. However, the separation alone provides no means to reason about completeness of a quality model. On the contrary, due to the large number of different artifact types, relevant ones can be easily missed.

To remedy this problem we not only include the different system artifact types (entities) but also explicitly capture their decomposition by including the *part-of* relationship between them in the model, e. g. a Method is part of a Class which is part of a Package which is part of a System. It needs to be stressed that this decomposition does *not* describe the actual decomposition of the system artifacts but a decomposition of the artifact *types* (entities). Hence, the quality model contains the entities Class and System but no entities Class X and System Y. Put differently, the entities and their relations describe a rough approximation of the metamodel of the system. We chose the *part-of* relation as the main mode of decomposition for the entities as it is well-known by software engineers and furthermore has the beneficial nature of defining a tree (or forest) on the set of entities. This tree-like decomposition allows to reason about the completeness of the model by traversing it in a top-down manner for the identification of missing entities. Fig. 4.6 shows a simple example quality model consisting of 7 activities, 8 entities and 4 attributes. This example does not build on Boehm's quality tree as this was obviously not designed with our modeling approach in mind and, hence, is suboptimal for illustrating it. The entities of the example are taken from one of the case studies discussed in Chap. 6. The activities tree is loosely modeled on the IEEE 1219 standard maintenance process [145].

⁶The choice of terms is based on a well-known paper by Kitchenham [179] that states that entities »are the objects we observe in the real world« and attributes are »the properties that an entity possesses«.

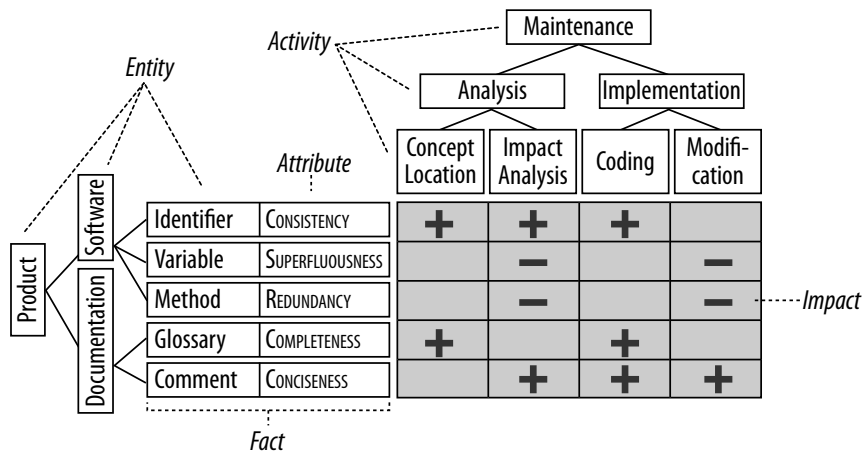


Figure 4.6: Maintainability Matrix Example

Due to the large number of entities, we found it beneficial to additionally use a second mode of decomposition based on a generalization (*is-a*) relation next to the (*part-of*) relation. This decomposition enables serves as basis for an inheritance mechanism for attributes and thereby greatly helps to build concise quality models. For example, we can define the entity Local Variable to be part of entity Method and also to be a specialization of entity Variable. An attribute defined for entity Variable is then automatically inherited to the subordinate entity Local Variable, e. g. the fact [Variable | LOCALITY] that expresses the desire for variables with a minimal scope is inherited to [Local Variable | LOCALITY]. When extending the quality model with an entity that specializes an existing one, it is ensured that important attributes are not overlooked. Hence, the inheritance mechanism does not only avoid redundancies in the quality models but also fosters model completeness. For clarity's sake, this second mode of decomposition is not shown in the examples above.

Impacts In contrast to Fig 4.5 the example above does not express a Boolean relation between facts an activities but uses a two-valued scale to express if an impact is positive or negative. As the formalization of the metamodel will show, more elaborate scales can be used to express different strengths of the relations. Using the notation introduced for facts we can elegantly express the impact a fact has on an activity:

$$[\text{Entity } e \mid \text{ATTRIBUTE } A] \xrightarrow{+/-} [\text{Activity } a]$$

Examples are [Identifiers | CONSISTENCY] $\xrightarrow{+}$ [Concept Location] describes that consistently used identifier names have a positive impact on the concept location activity and [Variable | SUPERFLUOUSNESS] $\xrightarrow{-}$ [Code Reading] that describes that unused variables hamper the reading of the code. To overcome the problem of unjustified quality guidelines each impact is additionally equipped with a detailed plain text description.

Assessment To assess the conformity of a given system to a quality model, it needs to be analyzed how the actual system characteristics relate to facts defined in the model. To allow this, each fact is equipped with a description that explains how conformity is assessed. Depending on the fact and the available means of analysis this can e. g. be a plain text description of conformity or a definition of metric thresholds. We deliberately chose not to make the definition of conformity explicit in the model as experience has shown that the assessment techniques used in practice are so various that a uniform description of the common denominator would not bring any benefit. To counter a lack of precision in the textual conformity descriptions, the metamodel furthermore requires each fact to be annotated with its *assessment type*. We distinguish three assessment types:

1. Facts that can be assessed or measured with a tool. An example is an automated check for *switch*-statements without a *default*-case ([Switch Statement | COMPLETENESS]).
2. Facts that can be automatically assessed to a limited extent but require additional manual inspection. An example is redundancy analysis where cloned source code can be found with a tool but other kinds of redundancy must be left to manual inspection ([Source Code | REDUNDANCY]).
3. Facts that require manual activities; e. g. reviews. An example is a review activity that identifies the improper use of data structures ([Data Structures | APPROPRIATENESS]).

Experiences have shown, that the explicit definition of the assessment type aids reasoning about the assessment descriptions and thereby helps to identify ambiguous descriptions. The categorization of the facts depends on the given context. For example, a general purpose quality model would base it on the current state of the art while a company-specific model would base it on the actual availability of tools.

Assessing a system of significant size can produce an overwhelming amount of data as each fact needs to be assessed not only once but for all artifacts it applies to. If the quality model contains facts that apply to artifacts that the system contains a plentitude of, e. g. methods, the number of individual assessments can go up to a hundred thousand or more. This is can be best explained by Fig. 4.7: Although the minimalist quality model defines only three facts, the assessment of the system results in 8 individual assessment results, 2 for fact [Class | STRUCTUREDNESS] that prescribes the order of fields and attributes within the class, 2 for the fact [Field | LOCALITY] that requires fields to be defined *private* and 4 for fact [Method | LOCALITY] that requires the method's visibility to be defined as minimal as possible.

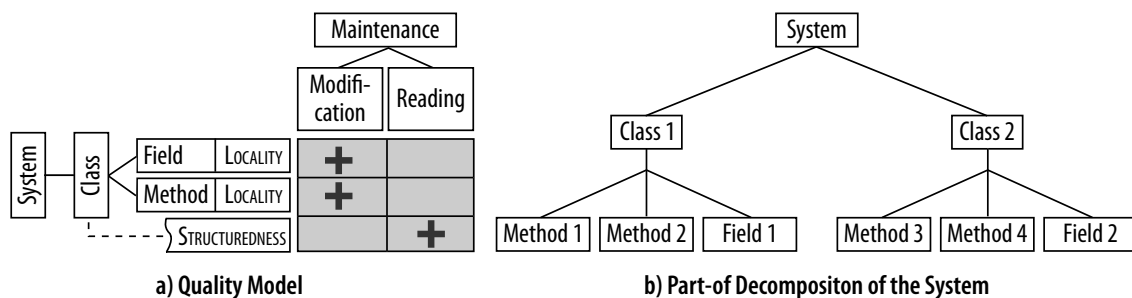


Figure 4.7: Assessment Data Aggregation

How this amount of data needs to be handled depends on the goal and mode of the assessment. If the assessment, for example, is carried out as part of a tight quality control process to identify the relatively few non-conforming artifacts that need to be corrected, a simple filter can be used. If the assessment, on the other hand, is used to create a quality profile of a previously unknown system, techniques to aggregate the assessment results are imperative to condense the amount of data. Aggregation is basically a mapping $Agg : M^n \rightarrow M$ that fuses several input values from one set to one output value of the same set [49]. Obviously, this requires the different facts to be measured on the same scale (the same set M). Only provided a uniformity of scales, aggregation can be carried out in a sensible manner, i. e. without succumbing to the same pitfalls as the maintainability metrics discussed in the last chapter. This is challenging due to the diverse nature of facts that need to be set off against each other. For example, one needs to integrate the assessment results for the facts defined above with others like [Program | REDUNDANCY] that relates to code cloning and [Variable | SUPERFLUOUSNESS] that prohibits variables that are defined but never used. One possible solution to this is to rate each fact an ordinal scale like $\{1, \dots, 6\}$.

However, uniformity of scales alone does not solve the aggregation problem as the application of standard aggregation operators [49] to all measured fact-artifact combinations condenses the assessment data too much. For example, when the ordinal scale suggested above is used, applying the arithmetic mean to all assessed facts on all artifacts will return something close to 3 for almost all systems whereas the max operator will usually return 6. This problem can be addressed by not flatly aggregating all fact-artifact combinations but by making use of a clearly defined decomposition. This allows a stepwise aggregation of data until a level is reached that is concise enough to be interpreted but still not too condensed to be meaningless.

An obvious choice for an aggregation dimension is the actual decomposition of the system as this allows to determine the state of quality of different parts of the system (Fig. 4.7b). A benefit of the stepwise aggregation along a defined decomposition can be illustrated here: It allows to use different scales and different aggregation operators for different levels of the hierarchy. For example, the assessment for the facts [Field | LOCALITY] and [Method | LOCALITY] could use a Boolean scale that only indicates if a field or method violates the quality requirement expressed by the fact or not. On the class level, however, this information could be converted to a ratio between the number of offending fields and methods and the total number of fields and methods. However, this requires a sophisticated definition of scales and aggregation operators for different levels and even artifact types.

From an economic perspective, one would ideally want to aggregate assessment data along the activities tree of the quality model to determine which additional effort is caused by non-conformity. Theoretically this could be done by first condensing the artifact-based assessment data to the entities of the quality model and then relating them to the activities' effort or by directly defining a relation between artifact assessments and activities. However, as we currently lack a thorough understanding of how different quality factors influence maintenance costs, this cannot be achieved in its entirety. Nevertheless, the case study in Sec. 6.3 demonstrates that such quantitative considerations can be carried out for particular cases with a limited scope.

Due to the variety of different artifact types included in quality models but also due to the different goals of aggregation, the requirements for aggregation are too diverse to be satisfied with a generic aggregation mechanism. Hence, we decided not to include such a mechanism in our quality modeling approach. Instead, our quality control framework ConQAT (Sec. 5.2) provides a variety of aggregation

mechanisms, including the ones discussed above, and allows to apply them in a customized way to match the context-specific requirements.

Quality Planning While the process of quality planning, i. e. the derivation of quality requirements and the according design of a quality model, is not at the focus of this thesis, we still give an idea how our quality modeling approach supports it. Obviously, already existing quality requirements for maintainability, e. g. in the form of guidelines, can be transferred to an activity-based quality model. This process is outlined in the example in Sec. 4.2.4. Beyond this, the activities as well as the entities tree of the quality model aid the effective and efficient creation of quality models. The activities tree can be used if an organization identifies an unsatisfactory execution of certain activities, e. g. debugging. It can then break further down this activity and reason about the facts that influence the relevant activities in a structured way. This can be viewed as top-down or goal-oriented approach to quality. Through the entities tree our model also supports a bottom-up approach. To explain this, one should recall that the entities tree describes the different artifact types of a system and, hence, a rough approximation of the system's metamodel. This circumstance allows the design of new quality models, e. g. for particular programming or modeling languages, to be guided by the metamodel or grammar of the language. In fact, the entities tree of the quality model can be directly modeled on the metamodel of the considered language. Starting from the entities tree, one can then reason about the relevant attributes and affected activities. An example for this design approach is discussed in Sec. 6.1 where we describe the design of a quality model for the Matlab Simulink modeling language.

Scope As discussed in Sec. 2.1.4, the factors influencing maintenance costs are not only product-related but include organizational as well as personnel factors. They are determined by a plethora of other factors which include the skills of the engineers, the presence of appropriate software processes and the availability of proper tools like debuggers. By putting *maintenance effort* instead of *maintainability* in the focus of the quality modeling efforts, we inevitably raise the question for the appropriate scope of a quality model. As these factors are expected to have as much of an influence as product-related factors, we consider it reasonable to include them, too. Since the impact of such factors can also be structured on the basis of activities, our quality modeling approach provides the opportunity to include them if needed. Examples are $[\text{Debugger} \mid \text{EXISTENCE}] \xrightarrow{+} [\text{Fault Diagnostics}]$, that describes that the existence of a debugger has a positive influence on the activity fault diagnostics.

4.2.3 The Quality Metamodel QMM

Although most quality models conform to an implicitly defined metamodel they usually lack an explicitly specified metamodel that precisely defines the set of legal model instances. In contrast to this, our model is based on the explicit quality metamodel QMM. The benefit of an explicit metamodel is twofold: First, it ensures a consistent structure of quality models. Second, it is a necessary basis for tool support as described in the next chapter. The metamodel consists of the elements discussed above: entities, attributes, facts, activities and impacts. An overview of the metamodel is presented as an UML class diagram in Fig. 4.8. The following sections give a formal definition of the metamodel based on sets, functions and predicates. The creation and maintenance of models that conform to this metamodel is supported by the quality model editor QMM.editor presented in Sec. 5.1.

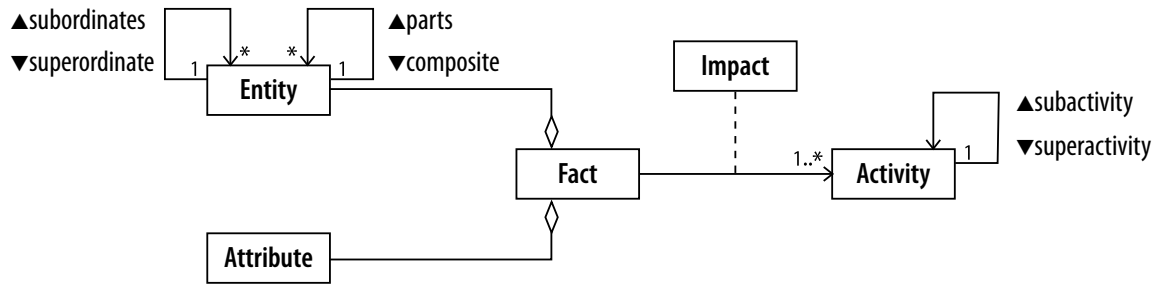


Figure 4.8: The Quality Metamodel QMM

Entities The entities of a quality model are defined by a set E . To support the development of comprehensive quality models, the metamodel uses two relations to impose a structure on the set of entities: the *composition* function \mathcal{C} and the *generalization* function \mathcal{G} . These functions are required as the entity set of realistic models usually consists of more than 500 elements and, hence, are hard to handle without a clearly defined structure. For example, a quality model for systems implemented in Java needs to express quality criteria for the various entities present in Java systems, e.g. Class, Method, Constructor, Package, Variable, Statement.

For the given example, the composition function \mathcal{C} is used to express that classes are parts of packages: $\mathcal{C}(\text{Class}) = \text{Package}$. The generalization relation \mathcal{G} is used to express that a constructor is a special kind of method: $\mathcal{G}(\text{Constructor}) = \text{Method}$. The quality metamodel uses the (partial) composition function \mathcal{C} as the primary mode of decomposition and defines it in a manner that ensures that the structure imposed on the entities set is a *forest*. We do not require the function to define a tree with a single root entity to support the bottom-up construction of quality model. Function \mathcal{C} is defined as:

$$\begin{aligned} \text{(Composite)} \quad \mathcal{C} : E \rightsquigarrow E \\ \mathcal{C}(e) = \text{composite element of entity } e. \end{aligned}$$

where $E \rightsquigarrow E$ denotes a partial function from E to E . The domain of the function is referred to as $\text{dom}.\mathcal{C}$ ⁷. The preimage of \mathcal{C} describes the parts of an entity and is defined as:

$$\text{(Parts)} \quad \mathcal{C}^{-1}(e) = \{x \in \text{dom}.\mathcal{C} \mid \mathcal{C}(x) = e\}$$

Based on the part function \mathcal{C}^{-1} we define the constituents function $\mathcal{C}_{\infty}^{-1}$ that returns the parts of an entity and recursively the parts of all its parts:

⁷The term »domain« is ambiguously used in the context of partial functions. We consider the domain of \mathcal{C} to include all elements $e \in E$ for which \mathcal{C} is defined.

$$\begin{aligned}
\text{(Constituents)} \quad \mathcal{C}_\infty^{-1} : E &\rightsquigarrow \mathcal{P}(E) \\
\mathcal{C}_\infty^{-1}(e) &= \bigcup_{i \in \mathbb{N}} (\mathcal{C}^{-1})^i(e)
\end{aligned}$$

where $(\mathcal{C}^{-1})^i(e)$ describes the i -times repeated functional composition and \mathbb{N} does not include 0. Furthermore, we assume that function \mathcal{C} can be extended to sets. Hence, $\mathcal{C}(\{e_1, \dots, e_n\}) = \{\mathcal{C}(e_1), \dots, \mathcal{C}(e_n)\}$ for $e_1, \dots, e_n \in E$. Likewise, \mathcal{C}^{-1} is also defined for sets.

To ensure, that the graph defined by the function c is acyclic, we require the following condition to hold:

$$\text{(Acyclicity)} \quad \forall e \in E : e \notin \mathcal{C}_\infty^{-1}(e)$$

The generalization function $\mathcal{G} : E \rightsquigarrow E$ defines a secondary mode of decomposition that serves as basis for the inheritance mechanism described below. The generalization function \mathcal{G} and its preimage \mathcal{G}^{-1} is defined in analogy to the decomposition function \mathcal{C} . For $\mathcal{G}(e)$ the value of the function is referred to as the *superordinate* of e and for $\mathcal{G}^{-1}(x)$ the values are referred to as *subordinates* of x . We also define a function \mathcal{G}_∞^{-1} in analogy to \mathcal{C}_∞^{-1} and refer to its values as *descendants*. Also, for \mathcal{G} acyclicity must hold.

Attributes Attributes are properties that entities possess. Attributes are defined by a set A . Example attributes are CONSISTENCY, REDUNDANCY and COMPLETENESS. The quality metamodel does not define additional relations to structure attributes for two reasons. Firstly, with the case studies described in Chap. 6 we made the experience that quality models rarely have more than 25 attributes. A set of this size can be comfortably managed without the need for a decomposition. Secondly, We are not aware of a criterion that would allow an unambiguous decomposition of fundamental attributes like CONSISTENCY.

Facts Entities describe the relevant artifact types of the system and attributes define properties. Their combinations, the facts, describe quality characteristics and have been identified to play an important role in the design of quality models as they are the items typically used in the discussion of quality. Moreover, facts describe the characteristic that need to be assessed in order evaluate a given system (or situation). Hence, the metamodel treats them as first class citizens and calls them *attributed entities* or *facts* (Fig. 4.8). The set of facts F is a subset of all possible combinations of entities and attributes:

$$\text{(Facts)} \quad F \subseteq E \times A$$

Facts are usually written as [Entity | ATTRIBUTE]. Example facts are [Class | SUPERFLUOUSNESS] that describes unused classes and [Goto Statement | EXISTENCE] that describes the existence of *Goto* statements in a program. For the set of facts F the following condition must hold:

$$\text{(Inheritance)} \quad \forall e \in E, \forall a \in A : (e, a) \in F \rightarrow (\forall x \in \mathcal{G}^{-1}(e) : (x, a) \in F) \quad (4.1)$$

This condition ensures that each entity adopts the attributes of its superordinate and is similar to the inheritance mechanism used in object orientation. It is, hence, referred to as *attribute inheritance*. The conditions helps to avoid repetitive attribute definitions in quality models and fosters the completeness of quality models. The rationale behind this is the following: If a quality model designer, includes, e. g. the fact [Program Element | SUPERFLUOUSNESS] in the quality model, it is likely that subordinates of the entity, e. g. $\mathcal{G}(\text{Class}) = \text{Program Element}$ or $\mathcal{G}(\text{Variable}) = \text{Program Element}$ should be equipped with the same attribute. If $(e, a) \in F$, we refer to $(\mathcal{G}(e), a)$ as the *super fact* (if $e \in \text{dom}(\mathcal{G})$). Vice versa $(x, a), x \in \mathcal{G}^{-1}(e)$ is referred to as a *subfact* of (e, a) .

Facts fundamentally define the quality requirements expressed by the quality model. To ensure that facts are actually assessable, each fact is equipped with an *assessment description* via function \mathcal{D}_{FA}

$$\begin{aligned} \text{(Assessment Description)} \quad \mathcal{D}_{FA} : F &\rightarrow P \\ \mathcal{D}_{FA}(f) &= \{\text{assessment description of fact } f\} \end{aligned}$$

where P is the set of all textual descriptions. This description does not only contain the required steps for analyzing quality characteristics but also the definition of conformity. Depending on the quality characteristic and the available means of analysis this can e. g. be a plain text description of conformity or a definition of metric thresholds. We deliberately chose not to include a formalization of the conformity definition in the metamodel as experience has shown that the assessment techniques used in practice are very diverse. A formal description of the common denominator would, hence, be either of low expressiveness or highly complex. According to our experience, the potential gains of the formalization do not warrant such an increase in model complexity. As discussed in Chap. 8 it may, however, be beneficial to extend the model with a formalization of the conformity definition for specific applications like certification.

To counter a lack of precision in the textual conformity descriptions, the metamodel furthermore requires each fact to be annotated with its *assessment type*. The assessment type describes if a fact can be assessed in a fully automatic manner, is supported by tools but requires manual aid or must be assessed completely manually. This categorization helps to reason about the assessment descriptions and is, furthermore, used by tools like guideline generators. The assessment type is expressed by assessment type function \mathcal{T} that associates each fact with an assessment type:

$$\begin{aligned} \text{(Assessment Type)} \quad \mathcal{T} : F &\rightarrow \{\text{MANUAL}, \\ &\quad \text{SEMI-AUTOMATIC}, \\ &\quad \text{AUTOMATIC}\} \end{aligned}$$

Activities Activities describe maintenance activities. They are defined by a set T and the superactivity function $\mathcal{A} : T \rightsquigarrow T$ that defines a *part-of* relation between activities. In contrast to the entities, the activities set rarely contains more than 50 elements in realistic quality models. Hence, case studies have shown that the second mode of decomposition defined for the entities is not needed here as the single function provides sufficient structuring for the set of activities.

The superactivity function \mathcal{A} and its preimage \mathcal{A}^{-1} is defined in analogy to the decomposition function \mathcal{C} . For $\mathcal{A}(t)$ the value of the function is referred to as the *superactivity* of t and for $\mathcal{A}^{-1}(x)$ the values are referred to as *subactivities* of x . We also define a function $\mathcal{A}_{\infty}^{-1}$ in analogy to $\mathcal{C}_{\infty}^{-1}$. Also, for \mathcal{A} acyclicity must hold.

Impacts A quality model is fundamentally defined through the (partial) impact function \mathcal{I} , that maps tuples of facts and activities (f, t) to an impact value and thereby describes which impact a fact has on an activity. The function is defined as:

$$\begin{aligned} \text{(Impacts)} \quad \mathcal{I} : (F \times T) \rightsquigarrow I \\ \mathcal{I}(f, t) = i \end{aligned}$$

where I is a set that characterizes the impact. Obviously, this is a partial function as an impact is not defined for all tuples of facts and activities. However, we require each fact to have at least one defined activity. Hence, the following condition must hold:

$$\text{(Dangling Facts)} \quad \forall f \in F : \exists t \in T : (f, t) \in \text{dom.}\mathcal{I} \quad (4.2)$$

As introduced earlier, impacts are usually written as:

$$[\text{Entity } e \in E \mid \text{Attribute } a \in A] \xrightarrow{\mathcal{I}((e,a),t)} [\text{Activity } t \in T]$$

The exact nature of the impact set I is deliberately underspecified as a suitable choice depends on the mode of operationalization of the quality models. Possible choices for I are $\{-, +\}$ that expresses a positive or negative impact or a one element set like $\{\text{TRUE}\}$ that is used to indicate the presence of an impact without specifying its nature. More elaborate choices are e. g. sets like \mathbb{Z}_3 or \mathbb{Z}_5 to express an ordinal scale and the set $[0..1]$ to express a percental impact on the effort associated with an activity. We elaborate on the choice of the impact set in the chapter on case studies (Chap. 6) where we use different examples for different purposes.

Note, that condition 4.2 is of relevance for the inheritance mechanism induced by condition 4.1. As each fact must have an impact on at least one activity, it is obviously required that subfacts of a fact have at least one impact, too. However, it is not required that a subfacts must have an impact on the same activities as the super fact. Note, that by choosing an impact set that explicitly models a *neutral* impact, e. g. $I = \{-, \circ, +\}$, a model designer can be enabled to effectively override impact definitions of super facts with the non-impact.

Activity Inference Next to the attribute inheritance, the quality metamodel also defines an *activity inference* mechanism that was established to support the efficient design of quality models. This mechanism ensures that impacts that are defined for activities are also defined for their subactivities. For example, a quality model expresses that all program elements that are defined but never used, have a negative impact on the activity Program Comprehension as superfluous program elements may confuse the reader:

$$[\text{Program Element} \mid \text{SUPERFLUOUSNESS}] \xrightarrow{-} [\text{Program Comprehension}]$$

This relation also implies that subactivities of Program comprehension, e. g. Concept Location and Code Reading are affect by the attributed entity:

$$\begin{aligned} ((\text{Program Element}, \text{SUPERFLUOUSNESS}), \text{Concept Location}) &\in \text{dom.}\mathcal{I}, \text{ and} \\ ((\text{Program Element}, \text{SUPERFLUOUSNESS}), \text{Code Reading}) &\in \text{dom.}\mathcal{I} \end{aligned}$$

Again, this is a weak form of inference that only requires that subactivities are impacted by the same attributed entities as their superactivities but does not require that the impacts are the same. The activity inference mechanism is formally defined as an implication:

$$(\text{Inference}) \quad \forall f \in F, \forall t \in T : (f, t) \in \text{dom.}\mathcal{I} \rightarrow (\forall y \in \mathcal{A}^{-1}(t) : (f, y) \in \text{dom.}\mathcal{I}) \quad (4.3)$$

Working Sets When working with large quality models, a mechanism to define views on selected parts of a quality model is of paramount importance. The metamodel supports this with so called *working sets*. A working set w is a tuple (E', A', T') where $E' \subseteq E$, $A' \subseteq A$ and $T' \subseteq T$. The following conditions must hold for E' and T' to ensure that working sets always contain whole subtrees of the entities and activity forests:

$$\begin{aligned} \forall e \in E' : \mathcal{C}^{-1}(e) &\subseteq E' \\ \forall t \in T' : \mathcal{A}^{-1}(t) &\subseteq T' \end{aligned}$$

A working set w is used to restrict the impact function

$$\begin{aligned} \mathcal{I}|_w : (F' \times T') &\rightsquigarrow I \\ \mathcal{I}|_w(f, t) &= \mathcal{I}(f, t). \end{aligned}$$

where $\mathcal{I}|_w$ denotes the restriction of \mathcal{I} to w and $F' = \{(e, a) \mid e \in E' \wedge a \in A' \wedge (e, a) \in F\}$. For example, one can define a working set that includes only the model elements that express impacts on the activity Program Comprehension

$$w_{\text{Program Comprehension}} = (E, A, \{\text{Program Comprehension}\})$$

or only impacts that relate to the entity Variable

$$w_{\text{Variable}} = (\{\text{Variable}\}, A, T)$$

This definition of working sets allows to combine working sets by using set operations on the entity, attribute and activity sets. For example, two working sets

$$\begin{aligned} w_{\text{Variable}} &= (\{\text{Variable}\}, A, T) \\ w_{\text{Class}} &= (\{\text{Class}\}, A, T) \end{aligned}$$

can be combined to

$$w_{\text{Variable} \cup \text{Class}} = (\{\text{Variable}, \text{Class}\}, A, T)$$

Summary Based on the ingredients introduced above a quality model M can finally be defined as a 4-tuple

$$M = (\mathbb{F}, \mathbb{A}, \mathcal{I}, I)$$

where \mathbb{F} is a tuple $(E, \mathcal{C}, \mathcal{G}, A, F)$ that describes the models facts through the set of entities E , the composition function \mathcal{C} , the generalization function \mathcal{G} , the attributes set A , and the facts set F . \mathbb{A} is a tuple (T, \mathcal{A}) that consists of the activities set T and the super activity function \mathcal{A} . \mathcal{I} is the impact function and I is the impact set.

While the 4-tuple defined above describes the basic structure of a quality model, the modes of operationalization presented later in the chapter require the following extensions to the metamodel. For clarity's sake they are not formally included in the metamodel. Most modes of operationalization require all model elements to be equipped with human readable names and detailed descriptions. The association of model elements and descriptions are defined by multiple functions that map model elements to descriptions. For example function \mathcal{D}_E below maps entities to descriptions and function \mathcal{N}_E maps entities to their name. Name and description functions are defined for all other model elements in analogy.

$$\begin{aligned} \text{(Entity Description)} \quad \mathcal{D}_E : E &\rightarrow P \\ \mathcal{D}_E(e) &= \{\text{description of entity } E\}, \end{aligned}$$

$$\begin{aligned} \text{(Entity Name)} \quad \mathcal{N}_E : E &\rightarrow P \\ \mathcal{N}_E(e) &= \{\text{name of entity } E\}, \end{aligned}$$

4.2.4 Example

To summarize the description of the quality metamodel and to illustrate how the modeling approach addresses the shortcomings of previous approaches, this section gives a comprehensive example. This example demonstrates how the approach can be used to model quality criteria for a FOR loop in the C programming language. The quality criteria to be integrated are taken from well-known and widely-used MISRA [209] and Ellementel [135] guidelines for C:

1. »The statements forming the body of FOR statement shall always be enclosed in braces.« [209]
2. »Floating point variables shall not be used as loop counters.« [209]
3. »Only expressions concerned with loop control should appear within a FOR statement.« [209]
4. »Numeric variables being used within a FOR loop for iteration counting should not be modified in the body of the loop.« [209]
5. »The choice of loop construct (FOR, WHILE or DO-WHILE) should depend on the specific use of the loop.« [135]

From these requirements we infer that a FOR loop obviously has a head, a body and a counter variable. Hence, we define: $\mathcal{C}^{-1}(\text{FOR Loop}) = \{\text{Head, Body, Counter Variable}\}$. We model the relation between the entities with the composition function \mathcal{C} as head, body and counter variable are *parts* of the FOR loop. Note, that this decomposition roughly matches the grammar of the C language but is, of course, highly simplified. For simplicity's sake, we assume that the relevant activities for modeling the quality criteria are program comprehension and modification which are both subactivities of maintenance: $\mathcal{A}^{-1}(\text{Maintenance}) = \{\text{Program Comprehension, Modification}\}$. Based on these preliminaries, we can model the criteria as follows:

1. The MISRA standard requires the body of a FOR loop to be enclosed in braces as this »avoids the danger of adding code which is intended to be part of the conditional block but is actually not« [209]. As this expresses a positive impact on the modification activity, it can be modeled as $[\text{Body} \mid \text{WELL-FORMEDNESS}] \xrightarrow{+} [\text{Modification}]$ where the attribute well-formedness was introduced to capture the requirement. Other guidelines consider the complete bracing also relevant for program comprehension. Hence, another impact can be added to the model: $[\text{Body} \mid \text{WELL-FORMEDNESS}] \xrightarrow{+} [\text{Program comprehension}]$. The fact $[\text{Body} \mid \text{WELL-FORMEDNESS}]$ can be checked automatically (and even corrected) by tools, hence $\mathcal{T}((\text{Body}, \text{WELL-FORMEDNESS})) = \text{AUTOMATIC}$.
2. The MISRA guidelines furthermore require that no floating-point variables are used as counter variables. Interestingly, the reason for this is not directly related to maintenance but rather to reliability. Rounding and truncation errors may lead to inaccuracies and, in fact, to unexpected results if the number of iterations varies from one implementation to another. With respect to software maintenance this can be a problem for portability, i. e. porting the software to another platform can produce unexpected errors⁸. Hence, we add the activity Porting and use the attribute APPROPRIATENESS to describe $[\text{Counter Variable} \mid \text{APPROPRIATENESS}] \xrightarrow{+} [\text{Porting}]$. This

⁸The use of floating point variables for loop counters can have other ramifications beyond the porting problem. For brevity's sake, these are not discussed here. See <https://www.securecoding.cert.org> for details.

fact can be analyzed automatically, too. Therefore $\mathcal{T}((\text{Counter Variable}, \text{APPROPRIATENESS})) = \text{AUTOMATIC}$.

3. According to the MISRA guidelines the head of a for loop serves the only purpose of initializing, incrementing and testing the loop counter. While the guidelines lack a clear description of the rationale behind this rule, we assume that additional statements in the loop head distract the reader and define $[\text{Head} | \text{PURPOSIVENESS}] \xrightarrow{+} [\text{Program Comprehension}]$. This fact cannot be assessed fully automatically but appropriate tools can identify heads that possibly violate the criterion, hence $\mathcal{T}((\text{Head}, \text{PURPOSIVENESS})) = \text{SEMI-AUTOMATIC}$.
4. Furthermore, the MISRA guidelines recommend not to modify the counter variable within the body of the loop. Again, we assume that this has negative consequences for program comprehension and define $[\text{Body} | \text{INTRICACY}] \xrightarrow{-} [\text{Program Comprehension}]$. We used a construction with a negative attribute here as we describe a single known anomaly with this impact. Write access to the counter variable can be analyzed automatically in some cases but is hard to do for all cases, hence $\mathcal{T}((\text{Body}, \text{INTRICACY})) = \text{SEMI-AUTOMATIC}$.
5. The Ellemtel guidelines require that the choice of loop construct (FOR, WHILE or DO-WHILE) should depend on the specific use of the loop. It, hence, asks for the appropriate loop construct to be used for a specific task. We assume that the appropriateness of the loop construct eases program comprehension and supports modification as well as porting and define: $[\text{FOR Loop} | \text{APPROPRIATENESS}] \xrightarrow{+} [\text{Maintenance}]$. The appropriateness of the loop construct cannot be analyzed automatically. Hence, $\mathcal{T}((\text{FOR Loop}, \text{APPROPRIATENESS})) = \text{MANUAL}$.

For demonstration purposes, this example uses 4 different attributes for 5 facts. Experience shows, that in realistic models the size of the attribute set A is usually no greater than about 25 elements and, hence, significantly smaller than the entities set which typically contains several hundred elements. Fig. 4.9 summarizes the impacts defined by the example quality model. The example demonstrates how the quality metamodel QMM allows to express quality criteria in a concise and consistent manner. The following sections explain how the modeling approach addresses the shortcomings identified for previous approaches in Chap. 3:

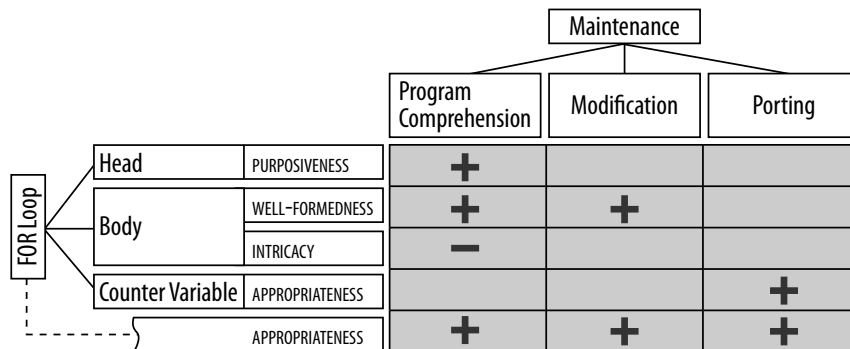


Figure 4.9: Example Model for the FOR Loop

- *Assessability.* The facts formulated above, e. g. [Body | WELL-FORMEDNESS] or [FOR Loop | APPROP.], are fine-granular enough to be assessed although this cannot necessarily be done automatically in all cases. At first sight, the metamodel does not prevent the definition of facts that are not assessable. For example, one could define a fact [System | SIMPLICITY] that is no more precise than criteria defined by the classic hierarchical models. However, our quality modeling approach counters this in two ways. First, it requires each fact $f \in F$ to be equipped with an assessment and conformity description via function \mathcal{D}_{F_A} and to be categorized w.r.t. to the assessment type via function \mathcal{T} . As these can hardly be provided for the fact [System | SIMPLICITY], a model containing this fact would be invalid. Second, the well-defined structuring mechanism provided by the metamodel allows a straightforward decomposition of such a fact, e. g. in [Subsystem | SIMPLICITY] or [Class | SIMPLICITY] and, hence, helps to break it down into assessable facts.
- *Rationale.* The quality model enforces the clarification of the rationale behind the quality criteria by making explicit the impact on maintenance activities. Hence, it prohibits the definition of criteria that like some of the examples above lack a description of the rationale or express it only vaguely.
- *Structuredness.* Finally, the quality metamodel provides a clearly defined decomposition and thereby fosters consistency and completeness of the quality models:
 - *Consistency.* Consistency is an issue, in particular, if multiple definitions of quality, e. g. multiple guidelines, are used. This can be exemplified with the following rules taken from the same guidelines as the examples above: »the break statement shall not be used (except to terminate the cases of a switch statement)« [209] and »use break to exit a loop if this avoids the use of flags« [135]. Obviously, these criteria contradict each other. Expressing them with our quality modeling approach, can reveal the contradiction if both criteria are expressed as facts of the entity Break Statement. Clearly, other ways of modeling the two rules can be thought of that would conceal this contradiction. However, experience shows the facts tree's decomposition that follows the decomposition of the artifact types is well-suited to reveal such issues. Next to this, the example makes a strong case for having *one* integrated definition of quality instead of multiple parallel ones.

Beyond the consistency regarding the content, the quality modeling approach also fosters the uniformity of the terminology. For example, it is quickly discovered that the MISRA guidelines call the counter variable *loop counter* whereas the Ellementel guidelines use *loop variable* and *iteration variable*. Particularly, the fact that different terms are used within one guideline document, highlights the practical relevance of the consistency issue.

- *Completeness.* To illustrate how the QMM aids quality model completeness, we assume that the quality model that describes the FOR loop is integrated with a larger quality model. This larger model defines an entity Program Element that we declare a superordinate of the FOR loop: $\mathcal{G}(\text{FOR Loop}) = \text{Program Element}$. As unneeded program elements confuse the reader, the model defines the impact [Program Element | SUPERFLOUSNESS] $\xrightarrow{-}$ [Program Comprehension]. Being a subordinate of the program element, the FOR loop automatically inherits the fact [FOR loop | SUPERFLOUSNESS], a criterion that is captured only implicitly by the guideline documents above. Although this is certainly valid, the model should be refined to express a particular form of superfluosness only exhibited by loops: loops that are iterated only once. This refinement is achieved by overriding the description of the respective

fact $\mathcal{D}(\text{[FOR loop | SUPERFLOUSNESS]})$. This is a criterion contained in neither of the guidelines above. Discovering it is supported by the inheritance mechanism of the quality model. As this criterion is true for all loops, the model can be further generalized by defining this criterion for a newly introduced entity *Loop* that is a subordinate of *Program Element* and the superordinate of *FOR Loop*. Having defined this new fact, the model designer must also add the missing facts to ensure that condition 4.2 is met. In this case, the impacts are simply copied from the superfact: $[\text{Loop | SUPERFLOUSNESS}] \xrightarrow{-} [\text{Program Comprehension}]$ and $[\text{FOR loop | SUPERFLOUSNESS}] \xrightarrow{-} [\text{Program Comprehension}]$.

The example demonstrates how the quality modeling approach supports the construction of well-structured and assessable quality models that clearly capture the rationale behind the individual quality criteria. However, such a model only is of real value if it is operationalized in the quality assurance process.

4.3 Operationalization

The following sections demonstrate how QMM-based models can serve as basis to *quality control* and the *quality assurance* activities that communicate the quality requirements to the developers in order to *prevent* quality defects.

Quality Control Lehman and other authors showed that software systems undergo a quality decay if no countermeasures are taken (see Chap. 2 for details). Without exception, this affects all of the commonly known quality attributes like reliability, functionality, efficiency, portability, usability and maintainability. However, maintainability is often affected most seriously as a decline in maintainability is not immediately visible to customers and, hence, often perceived as less critical. To counter this problem, a quality control process is required to analyze and pro-actively improve maintainability in a continuous and timely manner.

Quality control in non-software disciplines has been described by leading quality experts with the analogy of a feedback loop as often found in control systems [87, 169, 263]. The central idea is that quality cannot be evaluated only once but needs to be monitored continuously. Deviations from quality requirements are addressed by correcting the product. The feedback loop in its most basic form is often referred to as *Plan-Do-Check-Act-Cycle (PDCA)*⁹.

We propose to use the same analogy to describe the quality control process for software maintainability. Fig. 4.10 describes the feedback loop by using the terms of classic control theory: The *system* under control is the software system and the system's *output* is its maintainability. The system's maintainability is *disturbed* by modifications of the system that are caused by changes in the problem as well as the solution domain. The *desired* maintainability of the system is specified by a quality model. Each time the control loop is processed, the *quality analysis* determines the current state of maintainability. Based on the deviation between the current maintainability and the desired maintainability,

⁹Other frequently used terms are *Shewart-* and *Deming-cycle*, named after its inventor or its most prominent champion respectively.

the *quality engineer* triggers correction by asking the software *developers* for remedial actions. The developers improve the system, thereby change its maintainability and the process starts again.

An important question concerns the cycle time of the feedback loop. Here it is important, to find an equilibrium of timeliness and effort that ensures that quality decay is avoided without devoting all resources to quality control. While an actual *continuous* control, e. g. after every check-in to the version management system, may be realizable for automatically assessed facts, it's infeasible for manual reviews. Therefore, in practice, usually multiple cycles are used to assess different facts at different rates.

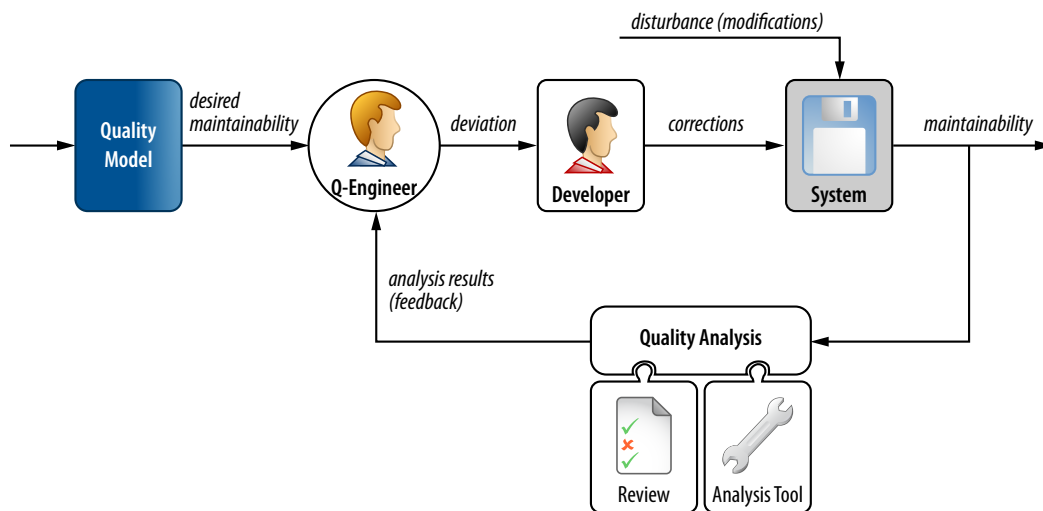


Figure 4.10: Quality Control Loop

Quality Assurance If the quality control process is strictly performed as described above, it is very unlikely that the system will ever reach a satisfying maintainability. The reason is that developers are asked to perform corrections of specific quality defects but the actual target quality requirements are never communicated to them. Hence, the quality engineer should not only ask for specific corrections but additionally educate developers to help prevent future quality defects. The main means for achieving this are quality guidelines that capture the quality requirements in a form developers are familiar with.

Continuous quality control enables the early identification of quality problems, when their removal is still inexpensive, and aids in making adequate decisions as it provides an overview on the current status of a software system. As a side effect, continuous and timely feedback enables developers and maintainers to improve their skills and thereby helps to avoid future quality defects. However, quality control for software maintainability is hardly applied in practice. We claim that this is caused by shortcomings of existing definitional approaches to specify quality requirements and their insufficient integration with constructive and analytic approaches. The central problem is that most definitional approaches define quality requirements whose conformity cannot actually be assessed. Moreover, the approaches are not well-structured enough to provide explicit relations to construc-

tive approaches and between different analytic approaches. In summary, this makes it tedious and costly to enact quality control and to ensure that it covers all relevant aspects.

The following sections illustrate how quality models as they are introduced in this thesis address these shortcomings and can, hence, be used as a basis for quality control and for the communication of quality requirements to the developers. This is achieved by automatically generating quality guidelines as well as review checklists from quality models by a tight coupling of quality analysis tools with the quality model.

4.3.1 Manual Reviews

Manual reviews are a powerful means for analyzing software artifacts' conformance to quality requirements. The checklists that are used to carry out the reviews, however, frequently suffer from a number of shortcomings regarding definitional aspects and also regarding the consistency between multiple checklists. Moreover, review activities are typically not well integrated with other, possibly automated, analytic approaches. The following sections show how generating review checklists from quality models based on the presented metamodel helps to overcome these deficiencies.

	Entity	Description	Assessment
Entity name: $N_E(\text{For Loop})$	FOR Loop (...)	A FOR loop is used only when the loop variable is increased by a constant amount for each iteration and when the termination of the loop is determined by a constant expression.	MANUAL Check if ...
Assessment Type: $T(\text{For Loop, SUPERFLUOUSNESS})$		A FOR loop is superfluous if its body is not executed at all or only once.	MANUAL Check if ...
Composite entity name: $N_E(C(\text{Counter Variable}))$	Counter Variable (FOR Loop)	Floating point variables shall not be used as loop variables as rounding and truncation errors may make the number of iterations unpredictable.	AUTOMATIC Done by PC-Lint
Assessment description: $D_E((\text{Body, INTRICACY}))$	Body (FOR Loop)	The counter variable should not be modified in the body of the loop.	SEMI-AUTOMATIC MockAnalyzer detects simple cases. Other ...
Fact description: $D_F((\text{Body, WELL-FORMEDNESS}))$		The statements within the body of a FOR loop shall always be in a block (enclosed within braces), even if they are a single statement.	AUTOMATIC Done by PC-Lint

Figure 4.11: Generated Review Checklist

Fig. 4.11 shows an excerpt from a review checklist that has been generated from the example quality model for FOR loops. The checklist lists all model entities alongside the facts defined for the entities. For each fact (table row) it shows the assessment type and the assessment description. While this is only one example how checklist generation can be implemented, it demonstrates how the resulting document addresses the shortcomings discussed in Sec. 3.4.3:

Definitional Aspects Gilb requires, that checklists »must ultimately be derived from the [review] rules [...]« [117]. While this is hard to ensure for classic, hand-written checklists, it is automatically guaranteed when checklists are generated from a quality model. This ensures consistency and completeness of checklists w.r.t. the quality requirements stated by the quality model. Furthermore, it has been noted, that »checklist items should not be too general« [45] as this complicates analysis of conformance. Checklists generated from our quality model address this as each fact is annotated with an assessment description that explains how the checklist item needs to be evaluated.

Consistency It is considered good practice to keep review checklists as short as possible, ideally a single page [45]. As all relevant criteria can hardly be expressed on a single page this usually leads to multiple checklists that cover different aspects. While this facilitates the review process, it poses the risk of inconsistencies between the checklists. Again, consistency can be automatically ensured when checklists are generated from a quality model. The model's working sets can be used to generate checklists that address different quality aspects. For example, one could define a working set w_{Porting} that contains only elements that affect the portability of a system and generate a checklist that contains only the facts relevant for this aspect.

$$w_{\text{Porting}} = (E \times A \times \{\text{Porting}\}) \quad (4.4)$$

Automation In [45] Brykczynski states that »checklist items should not be used for conventions better enforced through other means (e.g., by the use of automated tools [...])« as this helps to reduce inspection efforts and reduce inspection omissions. Our quality metamodel supports this as each fact is associated with an assessment type via function \mathcal{T} . This information can be used in multiple ways. For example, one could simply omit facts from checklists that are checked automatically or include the result of the automatic assessment in the generated checklists. In the next section we will show that this can be particularly beneficial for facts that are assessed semi-automatically as the checklist may include hints that support the manual assessment.

Summary The explanations above show, that the deficiencies of classic review checklists, i. e. completeness, assessability and consistency, can be overcome by generating checklists from quality models based on the metamodel QMM. Moreover, the metamodel supports the a tight integration of checklists with the automated quality assessments discussed in the next section.

4.3.2 Automated Assessments

Many quality attributes relevant for software maintenance can be assessed with quality analysis tools. However, the majority of existing tools operates virtually independently from the definition of quality. Hence, assuring that the quality analysis tools measure what is defined by a quality model is tedious. Moreover, it is difficult to check if criteria that have not been evaluated by tools are duly taken care off in inspections. The following sections explain how quality models based on the presented metamodel are ideally suited to be truly integrated with quality analysis tools.

Ideally, quality analysis would be supported by an analysis tool that is aware of a concrete quality model and directly relates the analysis results to it. It could then generate quality reports that follow the structure of the quality model and, hence, allow a straightforward assessment of conformity. Furthermore, the analysis tool could identify quality criteria that are not assessed automatically and require manual review. However, neither does such a tool exist nor is it realistic to expect one to be developed. The reason for this is the diversity of analysis problems discussed in the last chapter. Due to the high complexity and specificity of analysis tasks there never will be one single analysis tool that is capable of carrying out all required assessments. To exemplify this, think of a real-world software system consisting of ten thousands of artifacts in multiple different languages as different as XML, C, Java, Matlab Simulink as well as domain-specific languages that need to be analyzed with respect to a multitude of different quality aspects.

As no single integrated tool will be available, another means to relate the assessment results of multiple analysis tools to a quality model is required. To achieve this, we propose the application of an integration tool that processes the analysis results of all analysis tools as well as the results of manual reviews and relates them to the quality model. Based on this, it generates quality reports as well as other documents, e. g. prefilled review checklists that include the results of all automatically analyzed criteria. Fig. 4.12 illustrates the role of this integration tool that we refer to as *Q-Relator*.

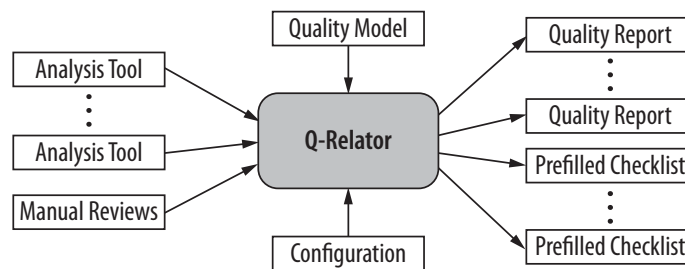


Figure 4.12: Relating Analysis Results to a Quality Model

To realize such an integration, the *Q-Relator* must be configured so it knows which analysis results relate to which elements of the quality model. This requires the relevant elements of the quality model to be unambiguously identifiable. The QMM inherently satisfies this and allows the *Q-Relator* to be configured accordingly. For example, a configuration could state the fact `[Body | WELL-FORMEDNESS]` is assessed by rule `ForLoopsMustUseBraces` of the static analysis tool PMD. Using this configuration the *Q-Relator* can then process the results of multiple analysis tools and generate quality reports that are directly related to the quality model. To illustrate this, Fig. 4.13 shows three different example views on the results generated by ConQAT, our implementation of the *Q-Relator* (discussed in detail in Sec. 5.2).

Each view focuses on a different aspect: Fig. 4.13a) lists system artifacts that inhibit the porting activity, Fig. 4.13b) shows artifacts that contain FOR loops which modify the counter variable within the FOR loop body and Fig. 4.13c) shows the familiar matrix notation, although it is used for assessment purposes here. As the remainder of this thesis will illustrate, a multitude of different views and aggregations are required to satisfy the needs of project participants as diverse as developers, quality engineers and project managers. However, the proposed approach always ensures that each view is

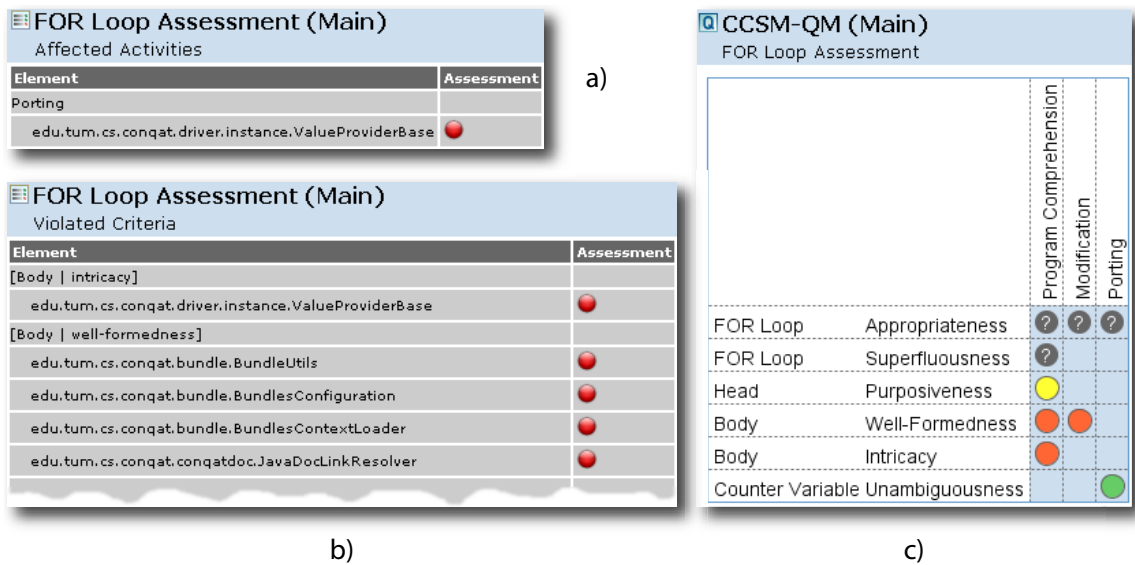


Figure 4.13: Quality Reports Generated by the Q-Relator

based on the same definition of quality as this is made explicit by the quality model processed by the Q-Relator. The explicit relation to the quality model enables the seamless integration of automated and manual analysis techniques. For example, the Q-Relator can be used to generate a prefilled review checklist that already contains all assessment results that were carried out by the analysis tools and thereby enables the reviewer to focus on the quality criteria that require manual assessments (see rows 3 and 5 in Fig 4.14).

This is especially valuable for facts that require semi-automatic assessment. Semi-automatic assessments are used if tools can only support a manual review by preparing the manual assessment but cannot fully automate it. This, in fact, applies to almost all analysis tools that generate »false positives« as these list *candidates* but leave the final decision to the human reviewer. One example are clone detection tools that aim at detecting redundancy by searching for duplicated code. Another example for semi-automatic assessments are problems where only specific cases can be handled automatically. A typical example is the fact [Body | INTRICACY] that requires that the counter variable of a FOR loop must no be modified within the loop body. It requires a very sophisticated analysis tool to correctly assess this fact in all cases as a full dataflow analysis is needed. However, simple cases where the variable is modified directly within the body of the loop can be detected quite simply. As this case is expected to be the most frequent, it is worth applying an automated analysis although it needs to be completed with a manual review. Due to the existence of precisely defined quality model both types of assessments can be integrated by the Q-Relator in a prefilled review checklist that supports the manual reviewer by relieving him from the task to check trivial cases. An example is shown in row 4 of Fig 4.14.

Summary The explanations above show that the central shortcoming of existing analysis tools, their integration with definitional and other analytic approaches, can be overcome by applying a

Entity	Description	Assessment	Result
FOR Loop (...)	A FOR loop is used only when the loop variable is increased by a constant amount for each iteration and when the termination of the loop is determined by a constant expression.	MANUAL Check if ...	
	A FOR loop is superfluous if its body is not executed at all or only once.	MANUAL Check if ...	
Counter Variable (FOR Loop)	Floating point variables shall not be used as loop variables as rounding and truncation errors may make the number of iterations unpredictable.	AUTOMATIC Done by PC-Lint	No violations found. ✓
Body (FOR Loop)	The counter variable should not be modified in the body of the loop.	SEMI-AUTOMATIC MockAnalyzer detects simple cases. Other ...	No simple violations found. Manual checking still required.
	The statements within the body of a FOR loop shall always be in a block (enclosed within braces), even if they are a single statement.	AUTOMATIC Done by PC-Lint	Violations in: line 32, line 148 ✗

Figure 4.14: Prefilled Review Checklist

soundly structured quality model that allows to define explicit relation between different quality analysis approaches. Besides the lack of integration, a number of other shortcomings of analysis tools were discussed in Chap. 3. We address these in Chap. 5 where we present ConQAT, an implementation of the Q-Relator.

4.3.3 Guidelines

An important task of quality assurance is to communicate the quality requirements to the developers and other project participants. As discussed in the previous chapter, quality guidelines are the central means to do this. However, classic guideline documents suffer from a number of shortcomings concerning definitional as well as constructive aspects. The following sections show how generating guideline documents from quality models based on the presented metamodel helps to overcome these deficiencies.

Fig. 4.15 shows an excerpt from a guidelines document that has been generated from the example quality model for FOR loops discussed in Sec. 4.2.4. The example guideline uses the model entities and their composition function \mathcal{C} as the basic structure of the guideline. For each entity $e \in E$ it lists all defined facts with their descriptions and their impacts. While this is only one example of how guideline generation can be implemented, it demonstrates how the resulting document addresses the shortcomings discussed in Sec. 3.3:

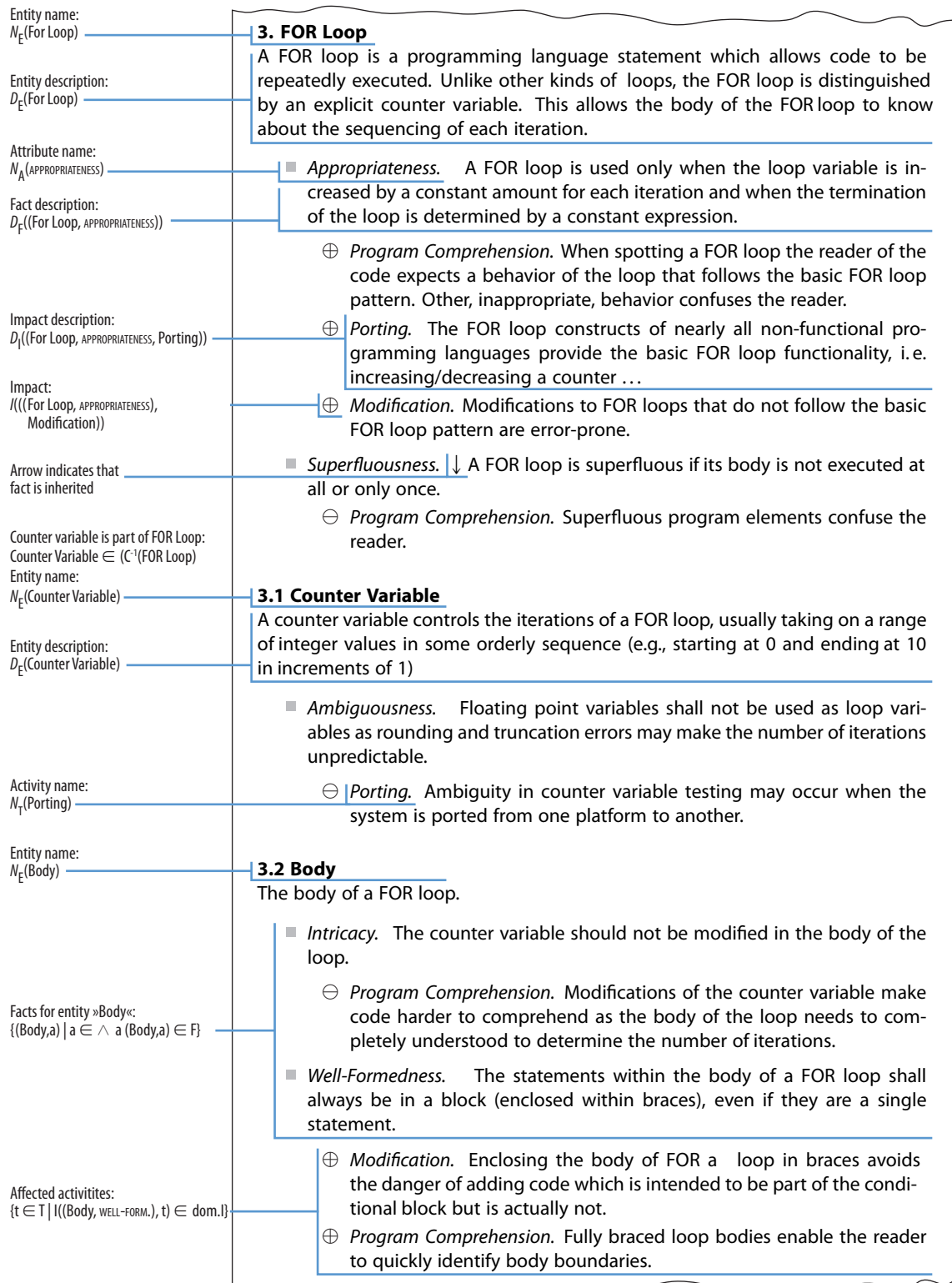


Figure 4.15: Generated Quality Guideline

Definitional Aspects Classic quality guidelines often exhibit a somewhat arbitrary structure. This is addressed by generated guidelines as they follow the clearly defined decomposition imposed by the quality metamodel. The guideline shown in Fig. 4.15, for example, uses the entities tree to structure the guideline. Hence, its chapter and section structure is unambiguously defined by the composition function \mathcal{C} , the facts defined for the entities and the affected activities. The inclusion of the affected activities automatically counters the lack of rationale present in many classic guidelines. The reader can instantly see why a rule is included in the guideline. Although the descriptions for assessing the individual criteria are not included in the example guideline, one can be sure that it contains only assessable criteria as the metamodels requires an assessment description for each fact. If needed, this description could of course be included in the guideline. Classic guideline documents are usually not well integrated with the analytic quality assurance approaches as there is no defined relation to them. With quality guidelines generated from our quality model, this relation is automatically defined. This allows, e. g., to relate quality assessment results directly to sections in the guidelines.

Constructive Aspects There are also constructive aspects of quality guidelines that have been criticized. Usually the reason for this is that guidelines are insufficiently tailored for their target audience. While the guideline shown in Fig. 4.15 may be well suited for a beginner or a new project participant, a seasoned developer will perceive it as too verbose as he wants to have only a concise list that reminds him of the quality criteria. Such a list can easily be generated by customizing the guideline generator, e. g. by making it leave out the descriptions of the entities and the list of impacts (see Fig. 4.16). While this provides a better suited *view* on the same quality model, consistency is still ensured as all project participants work with the same quality model.

Besides the level of presented details, it has been criticized that quality guidelines often contain information that is not relevant for certain groups of developers. One example are quality criteria specific to certain language constructs or libraries that may or may not be used in a particular project. While they are valuable for project participants that work with such a library, they are perceived as disturbing by participants who do not. Experience showed that this kind of tailoring is so frequent, that the customization of the guideline generator to specifically include or exclude certain model elements is not a viable way. Hence, such tailorings are best realized with *working sets* defined by the quality model. For example, one can define a working set that includes all language constructs but pointers as these are not used in certain systems. The generation of the guideline can then be performed for the specified working set:

$$w_{\setminus \text{Pointer}} = \left((E \setminus (\{\text{Pointer}\} \cup \mathcal{C}_{\infty}^{-1}(\text{Pointer}))), A, T \right)$$

Since working sets are defined as restrictions on the impact function \mathcal{I} , it makes no difference for the generator if it works on the impact function \mathcal{I} itself or on a working set. As working sets can be combined using set operations on the elements of their domain, the same generator can be used to generate guidelines for all working sets.

Summary The explanations above show, that the definitional shortcomings of quality guidelines, i. e. lack of structure, lack of rationale, can be overcome by generating guidelines from the quality

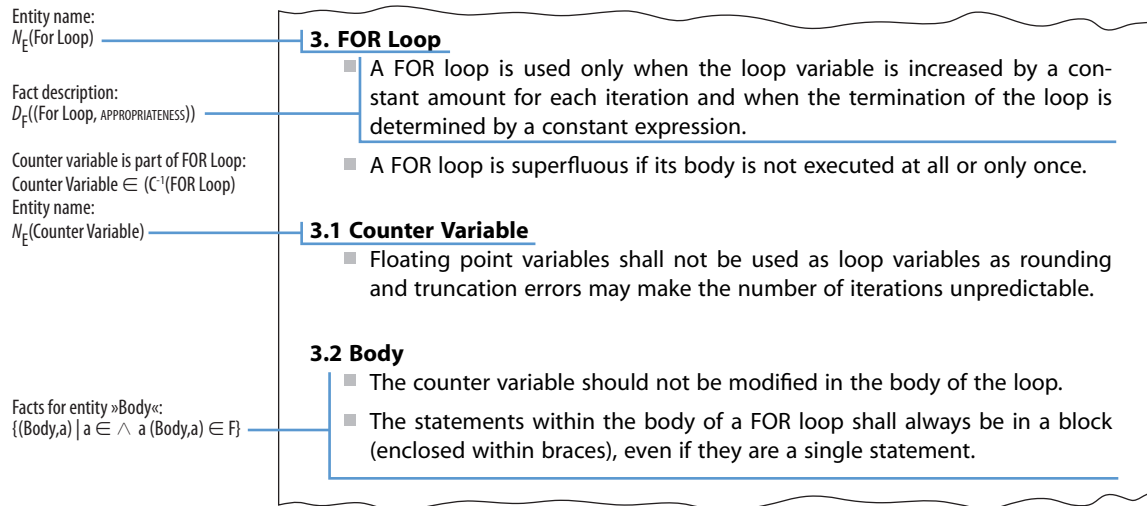


Figure 4.16: Simplified Quality Guideline

models based on the metamodel QMM. Moreover, working sets as well as customized guideline generators help to tailor the guideline documents for the intended target audience while still ensuring consistency.

4.4 Summary

In the chapter on the state of art (Chap. 3) we concluded that today the biggest obstacle towards a mature discipline of maintainability management are the approaches used to define maintainability and their integration with constructive and analytic means of quality assurance.

To remove this obstacle, the quality modeling approach presented in this thesis addresses all shortcomings identified for previous approaches:

- *Assessability.* Previous approaches do not define criteria for maintainability at a level that is suitable for an actual assessment. Hence, it is not possible to evaluate if a system complies to stated quality requirements or not. The quality metamodel QMM requires each fact to be equipped with a precise assessment description and a classification of the assessment type (manual, semi-automatic, automatic). This helps to avoid the definition of non-assessable facts. Furthermore, the structure imposed by the metamodel supports the decomposition of non-assessable into more tangible, assessable ones.
- *Rationale.* Previous approaches often omit the rationale behind the required properties of the system. This makes it difficult to describe impacts precisely and therefore to convince developers of the importance of the proposed quality criteria. Our modeling approach is based on the relation between quality facts and the activities of the software maintenance process. For each fact, this relation expresses its relevance for software maintenance and thereby explains its

rationale. As the maintenance activities provide a breakdown of the total maintenance effort, this can also contribute to a discussion of quality economics.

- *Structuredness.* Previous approaches often use ambiguous decomposition dimensions which leads to inconsistent models and hampers the revelation of omissions and inconsistencies in these models. Our approach overcomes this shortcoming by rigorously separating aspects that are typically intermingled: activities, entities and attributes. This separation creates separate hierarchies with clearly defined decomposition criteria. In contrast to previous approaches, our modeling approach is based on the explicitly defined quality metamodel QMM. The use of an explicit metamodel further fosters the conciseness, consistency and completeness of quality models as it forces the model designer to stick to an established structure and supports him in finding omissions. The rigid structure of the quality model instances enables us to provide a rich set of tools for editing and maintaining quality models (see next chapter).

Another major shortcoming of previous quality models is that they provide no real operationalization to support constructive and analytic means of quality assurance. This is a precarious situation as long-lived software systems are known to undergo a quality decay and should, hence, be subject to continuous quality control. Due to the rigid structure imposed by the quality metamodel QMM, our quality models overcome this problem and lend themselves to serve as basis of a quality control process. They support this process not only by providing a precise definition of what constitutes maintainability in a given context but also by the automatic generation of review checklist and the integration with quality analysis tools. Moreover, they support quality assurance by the automatic generation of quality guidelines that are used to communicate the quality requirements to developers and other project participants.

However, it needs to be noted, that modeling quality is essentially *modeling*. Hence, all the challenges one faces when modeling, i. e. creating an appropriate abstraction, apply for modeling quality, too. In particular, every sufficiently expressive modeling language offers multiple different, perhaps differently suitable, ways to express the same concepts. Consequently, it is as likely that someone builds poor quality models with our modeling approach as it is that someone misuses any other modeling or programming language.

We were aware of this problem when designing this modeling approach and, consequently, took great care to find a metamodel that is rigid enough to prevent obvious misuses but is still simple enough to be applied in a straightforward manner. This design process was based on a careful analysis of the shortcomings of previous approaches as well as on the experience we gathered with multiple case studies in different contexts. Hence, we are confident that our approach provides the necessary solid basis for modeling and controlling of maintainability. To substantiate this, Chap. 6 reports on our experiences with the approach in various commercial and academic contexts. The next chapter, however, introduces the tools that are necessary to put quality modeling and control into practice.

»Concepts and tools, history teaches again and again,
are mutually interdependent and interactive.«

Peter F. Drucker

5 Tool Support

It has been repeatedly discussed in this thesis, that the operationalization of quality models is of paramount importance. This chapter describes the tools that were developed to achieve this. This includes QMM.editor, an editor to create and maintain QMM-based quality models that are used to explicitly define the target quality. Moreover, we present ConQAT, a toolkit to build quality dashboards that supports the continuous control of quality factors during software maintenance.

5.1 Quality Model Editor »QMM.editor«

Realistic quality models often contain several hundred model elements. Hence, comprehensive tool support is necessary to design and maintain these models in an efficient and effective manner. Moreover, tool support is required to ensure that quality models really conform to the quality metamodel QMM introduced in the last chapter and to generate checklists and guidelines from quality models. Our earlier studies have shown that handling realistically sized models is plainly infeasible if no adequate tools are provided. Hence, significant effort was spent on the development of the graphical model editor QMM.editor that supports all relevant modeling tasks. The following section gives an overview of the editor and its implementation of the quality metamodel QMM.

5.1.1 Metamodel Implementation

The quality model editor QMM.editor is directly based on the quality metamodel QMM. Hence, all model element types discussed in the last chapter can be created and modified in the editor. However, the following extensions, concretions and simplifications of the QMM were introduced. These deviations from the formally defined metamodel are driven by the experiences we made in the application of the editor. We found that such concretizations are fundamental for operationalizing quality models. The following list gives a detailed account of all deviations from the formal metamodel:

- *Impacts.* The metamodel allows to use arbitrary impact sets I . As we primarily use the impact set $I = \{-, +\}$ in our cases studies we, the QMM.editor implementation is currently restricted to this impact set.
- *Ids.* A central feature of the QMM is the unambiguous identifiability of all model elements. This is required to relate constructive and analytic means of quality assurance to specific elements of a quality model. The formalization straightforwardly provides this identifiability by using sets. In the concrete implementation of the QMM, however, designated identifiers are required to provide identifiability for persisted models. The implementation of the QMM realizes this with two different mechanisms:

- *Textual Ids.* Each model element in a quality model has a textual identifier. This identifier is not required to be globally unique but must be locally unique so it is ensured that model elements can be identified by path-like expressions like *system/for_loop/body*. Such expressions are called *fully qualified names*.
- *Numeric Ids.* Textual identifiers have the advantage of being readable by humans and, additionally, the fully qualified names carry information about the location of a model element within a model. For certain purposes these two advantages, however, turn out to be disadvantages. First, the fully qualified names tend to be rather lengthy for elements in large models and, hence, are sometimes inconvenient to use. Second, the fully qualified names are sensitive to structural changes of the model, e. g. moving an element from one place in the model to another changes its fully qualified name. To be able to unambiguously identify model elements regardless of their location in the model, each model element is additionally equipped with a globally unique numeric id. This id is assigned at element creation and never changed thereafter; it is also called the *permanent id*.
- *Languages.* In many industrial contexts it is required to express the quality models in multiple languages. As building two quality models that differ only in the language of their descriptions is tedious and error-prone, the QMM.editor allows to define the textual descriptions of all model elements in *multiple* languages. With respect to the formal metamodel, we, hence, define multiple description functions for each model element type and language. Likewise, the assessment description function \mathcal{T} is defined for multiple languages.
- *Textual Markup.* As model element descriptions are often of significant length, the QMM.editor does not limit them to plain text but provides a simple Wiki-like syntax to define basic text formatting like bullet lists and enumerations.
- *Rooted Entities Tree.* The QMM.editor requires the forest defined by the entities and the composition function \mathcal{C} to be connected and to have a single root element. Experience has shown that this eases navigation of the entities tree. As an artificial root element can be used, this poses no actual limitation with respect to the QMM. This constraint does *not* apply for the generalization forest.
- *Sources.* To further document the rationale behind model elements, it is beneficial to annotate them with the origin they derive from. Hence, the QMM.editor introduces the additional model element type *source* that describes references like articles, books or websites. Each model element can be annotated with multiple sources.
- *Assessment Type.* The QMM defines the three assessment types MANUAL, SEMI-AUTOMATIC and AUTOMATIC for facts. However, experience showed that quality model designers often want to express that a fact could theoretically be assessed automatically but the organization currently lacks the required tools. To express this, we introduced the assessment type POTENTIALLY AUTOMATIC. Moreover, quality model designers often are not experts in automatic analyses and, therefore, do not know if a fact can be assessed automatically. To express this, we introduced the assessment type UNKNOWN. However, the quality model editor warns the user about the existence of facts with assessment type UNKNOWN.
- *Attribute Inheritance.* The QMM defines predicate 4.1 to model the inheritance of attributes. In the QMM.editor this not implemented as a predicate to check model consistency but as a

mechanism that automatically creates the inherited facts. For example, if one defines the fact [Program Element | SUPERFLUOUSNESS] the model will automatically contain all subfacts.

- *Activity Inference.* A similar mechanism is used to implement the activity inference predicate 4.3. Each impact defined for an activity is automatically inferred for all its subactivities.
- *Dangling Facts.* Through condition 4.2 the QMM requires each fact to have at least one defined impact. In contrast to attribute inheritance and activity inference this condition cannot be automatically satisfied as the impact cannot be derived by the editor. Therefore, another mechanism is needed to ensure that this condition is met. This is problematic because the step-wise design of a quality model usually includes states that violate the condition. Hence, the editor cannot generally prohibit models that violate the condition. The editor resolves this dilemma by accepting models that violate the condition but warns the user about the detected incompleteness.
- *Other Completeness & Consistency Issues.* The same mechanism is used to warn the user about other potential problems regarding completeness and consistency. Examples are model elements that lack a description in a specified language or facts that have assessment type UNKNOWN.
- *Working Sets.* In the current implementation of the QMM.editor, working sets can be defined only by specifying the entities and attributes that belong to a working set. The set of activities included in a working set currently cannot be restricted.

5.1.2 Overview

The editor's main window consists of several views that are used to edit the different element types of the metamodel QMM (Fig. 5.1). The following list gives an overview of the views and explains how they relate to the quality metamodel (numbers and letters of the list below refer to elements shown in the screenshot in Fig. 5.1).

1. *Model Explorer.* The model explorer shows all entities, activities and attributes of a model. Double-clicking an element opens it in the editor view (6) to edit its properties. The model explorer comprises the following parts:
 - a) *Entity Composition Tree.* The entity composition tree shows all entities in a collapsible tree. The tree structure is defined by the composition function \mathcal{C} . Context menu entries allow to create new entities as parts of an existing one and to equip an entity with an attribute to create a new fact.
 - b) *Activities Tree.* The activities tree shows all activities in a collapsible tree where the tree structure is defined by the superactivity function \mathcal{A} . A context menu entry is provided to create new subactivities of existing activities.
 - c) *Attributes.* The attributes view lists all attributes of the quality model. A context menu entry to create new attributes is provided.

2. *Matrix View.* The matrix view visualizes the impact function with the matrix-based notation introduced in the last chapter. Matrix elements can be selected to open them in the model explorer view.
3. *Generalization View.* The generalization view is used to visualize the tree induced by the generalization function \mathcal{G} . It is synchronized with the model explorer view to display the subordinates of the element currently selected there. Context menu entries are provided to create new subordinates of existing entities.
4. *Facts View.* The facts view shows all facts defined for the currently selected entity. For each fact a list of impacts is shown. A context menu entry allows to create new impacts.
5. *Sources.* The sources view lists all available sources that can be associated with the model elements to document their origin. Menu entries are provided to create new sources.
6. *Editor.* The editor view is used to edit the model elements. The view is subdivided into multiple sections where some are common for all model element types and some show information specific for the element type, e. g. the assessment type for facts or the impact value for impacts. The common sections for all model element types are:
 - a) *General Information.* This section shows the id, the fully qualified name, the permanent id as well as creation and modification time of the model element.
 - b) *Details.* The details section shows details that are specific for the type of the edited model element. For example, the details section in the screenshot shows the composite elements as well as the superordinates of the edited entity. Moreover, it provides controls to change the superordinate and the composite. For clarity's sake, the details section is accompanied by another type-specific section for some element types. For example, the *attributes* section in the screenshot shows an overview of the facts associated with the entity and illustrates the inheritance of the attributes.
 - c) *Languages.* The QMM.editor allows to define multiple descriptions of each element to support multi-lingual quality models. The language-specific description of the elements are edited on separate editor tabs that can be selected at the bottom of the editor. The list box (6c) is used to add or remove support for a specific language to or from the edited model element.
 - d) *Sources.* The QMM.editor allows to associate each model element with multiple *sources* to document where the element derives from. The list box (6d) is used to add and remove sources.
7. *Problems.* The problems view reports inconsistencies and incompleteness of the model. Examples are lacking descriptions or impact definitions.
8. *Working Sets View.* The working set view shows working sets defined for a quality model. Menu entries are provided to create new working sets.

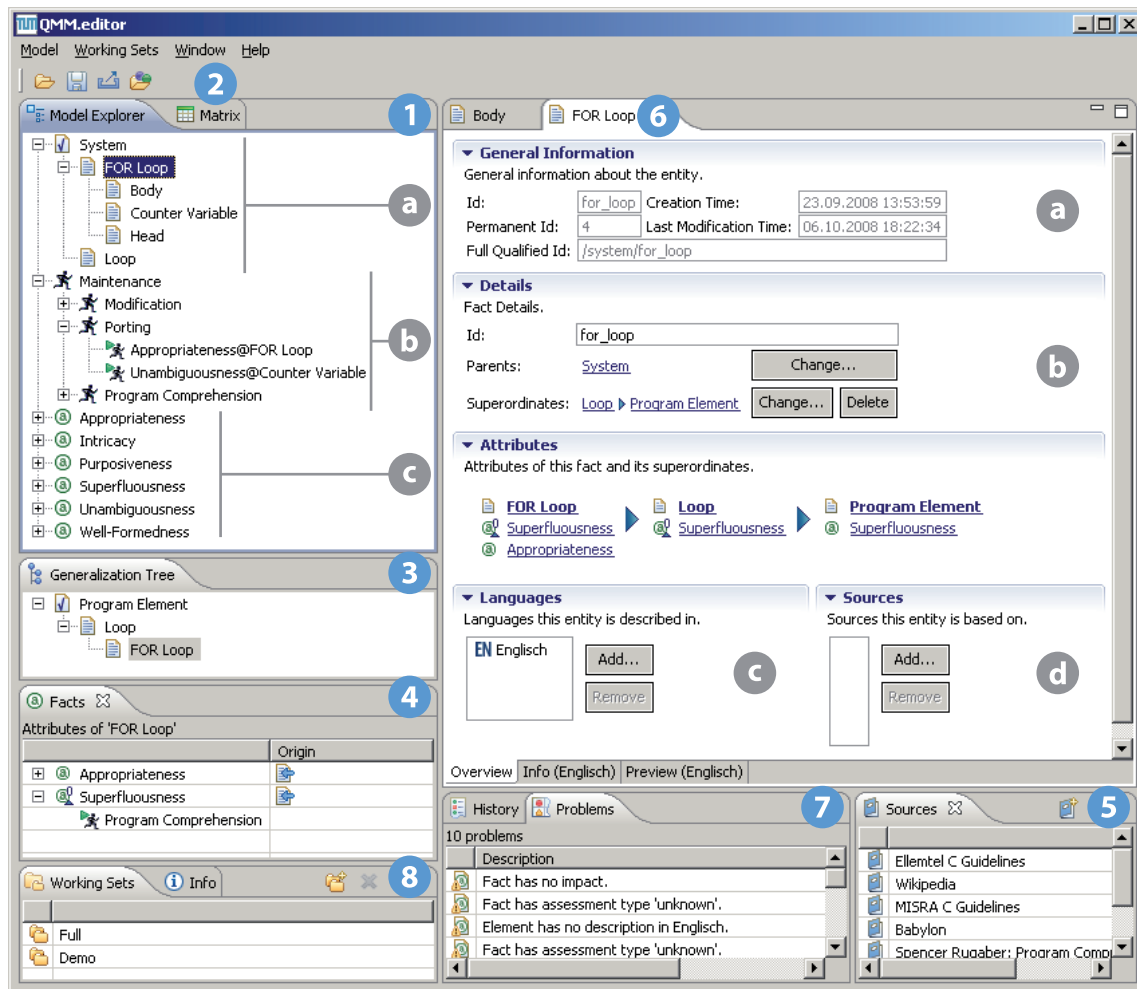


Figure 5.1: Screenshot of the Quality Model Editor

5.1.3 Model Design & Maintenance

The QMM.editor is designed to support quality model creation in a top-down as well as bottom-up manner. Hence, model designers can start modeling with any model element, may it be an entity, an activity or an attribute. To support this, the QMM.editor allows to create incomplete or inconsistent models but informs the designer about such problems. To support the iterative development of models, the model designer must not provide all required textual descriptions when an element is created but can build the basic structure first and then annotate the elements with the descriptions. If required, this can be done in several languages.

The actual construction of the quality model is mainly guided by context-menu entries that provide access to the relevant operations. For example, the context menu of an entity provides an operation to create a new part of this entity and an operation to associate the entity with an attribute in order to define a new fact. Similarly, the context menu of a fact provides an entry to define an impact on

an activity. If reasonable, these operations are supported by graphical dialogs, e. g. the target activity for a new impact can be selected in a graphical representation of the activities tree.

Refactoring Quality models need to be extended and updated as quality requirements change or new ones arise. To efficiently support this, the quality model editor provides several refactorings as they are known from modern integrated development environments. For example, model elements and their ids can be changed at any time. Moreover, model elements like entities and activities can be freely moved within their respective trees. For all changes, the editor ensures that relations to other model elements are maintained. Refactorings that would lead to invalid states, e. g. duplicate local ids, are prohibited by the editor.

Working Sets Due to the size of realistic quality models, working sets are an indispensable tool for managing them. Hence, the editor uses working sets not only to limit the scope of generated guidelines and checklists as discussed in the last chapter but also to provide views on the currently edited model. To use this feature, a model designer can pick one of the previously defined working sets from the working set menu and thereby make the editor hide all model elements that do not belong to the working set from all views. This enables him to focus on the a specific part of the model but still provides the flexibility to change to other parts within one mouse click only.

5.1.4 Checklist & Guideline Generation

Next to the design and maintenance of quality models, the QMM.editor is also used to generate review checklists, quality guidelines and possibly other documents from a quality model. To support this the editor provides an extension interface that allows other developers to implement new exporters. Currently, the editor provides the following exporter implementations:

- *Entity Tree Exporter.* This exporter exports a visualization of the entities tree in several different graphic formats. The entity tree export is useful for gaining an overview on the model and for presenting it, e. g. in developer trainings.
- *Activity Tree Exporter.* The same exporter is provided for the activity tree.
- *Matrix Exporter.* The matrix exporter exports the matrix visualization of the impact function to a bitmap or vector graphic format. This exporter, too, is mainly used for preparing presentations for developer trainings.
- *Latex Checklist Exporter.* This exporter exports a simple review checklist in the Latex format.
- *MSR MEDOC Exporter.* The MSR MEDOC exporter is currently the most advanced exporter. It exports a quality model as a quality guideline document in the MSR MEDOC format¹. The generated guideline document is structured in two parts where the first part contains the exported fact in a concise, checklist style form and the other part contains all relevant further

¹The MSR MEDOC format is an XML application to describe generic textual documents. It was defined by the MSR MEDOC working group whose members are mainly from the German automotive industry (see <http://www.msr-wg.de/medoc> for details). The format was chosen as one of our main industrial research partners uniformly uses it for all its documents. The documents in the MSR MEDOC format can be converted to PDF with XSLT.

explanations. In PDF files that are generated from the MSR MEDOC format, the two parts reference each other with hyperlinks. This allows developers and reviewers to focus on the concise checklist and only consult further explanations if needed. Examples of guidelines generated by this exporter and are shown in the chapter on case studies.

5.1.5 Implementation & Architecture

The QMM.editor is built upon the Eclipse Rich Client Application Platform² (RCP), as it provides a comprehensive framework for building graphical editors in a very efficient manner. Figure 5.2 illustrates the basic architecture of the QMM.editor. The figure includes only components that were specifically developed for the editor and omits standard libraries as well as the RCP framework itself; Eclipse plugins are shown in violet, other Java libraries in grey. The central components of the application are:

- *QMM Core.* The QMM Core is Java library that contains the implementation of the quality metamodel QMM, the persistence mechanism for quality models and a number of model-related utility functions. The implementation of the metamodel is a straightforward translation of the formal metamodel discussed in the last chapter to Java classes. Hence, the UML class diagram in Fig. 4.8 provides a suitable overview of the implementation classes. Following the command pattern [111], the QMM Core provides a set of classes to create new model instances and to modify existing ones. The persistence mechanism for quality models uses XML files to store and load quality models. The structure of these XML files is defined by an XML Schema. The QMM Core is not implemented as an Eclipse plugin as it would not benefit from the framework and is also used by non-Eclipse applications, e. g. by the quality control toolkit discussed in the next section.
- *Text.* The Text library implements a basic Wiki-like markup functionality for texts that is used to format the description texts of the model elements. We did not use any of the existing Wiki markup rendering engines as these are usually focused on transforming the markup to HTML whereas we must support a number of different output formats like LaTeX and MSR MEDOC. The Text library is not implemented as an Eclipse plugin, either, since it is used in multiple non-Eclipse contexts.
- *QMM Editor.* The QMM Editor is the central component of the quality model editor and provides all the functionality discussed above bar the export of checklists and guidelines. The vast majority of the component's code is purely concerned with the user interface of the editor and does not provide any additional logic beyond what is implemented in the core. An exception is the auditing functionality that checks quality models for completeness and consistency. This currently resides in the editor component but will be moved to the core in the future. The component provides an extension point³ that other plugins can use to contribute exporters to the editor.
- *Exporters.* This extension point is used by the exporters that were already introduced above. To use the extensions points exporters must implement a specific Java interface. This interface

²<http://www.eclipse.org/rcp>

³An *extension point* is the standard Eclipse mechanism to define interfaces between plugins.

is intentionally kept as lightweight as possible to not restrict the type of exporters it describes. Hence, exporters cannot only be used to export checklist and guidelines but to generate any kind of information from a quality model.

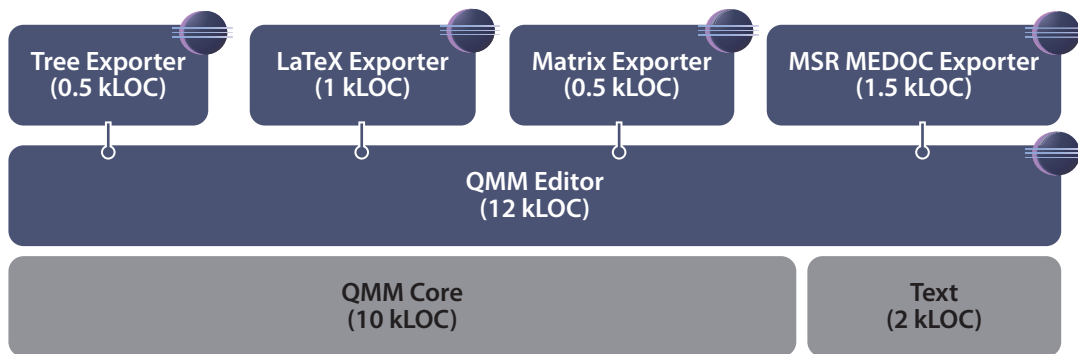


Figure 5.2: Quality Model Editor Architecture

5.1.6 Summary

While the QMM.editor provides all required functionality to design and maintain QMM-based quality models, it has neither the maturity nor the functionality that is required for day-by-day use of non-experts. For example, it currently lacks proper search functionality to find model elements by searching their descriptions as well as functionality to copy model elements from one model to another. Nevertheless, we view the editor as being beyond a mere proof-of-concept implementation and the fact that multiple researchers as well as one industrial partner use the editor seems to confirm this opinion.

5.2 Quality Control Toolkit »ConQAT«

Due to the diversity of factors that influence maintainability comprehensive tool support for quality assessment is indispensable to establish an effective and efficient quality control process. Tools are not only needed to carry out automated quality assessments but also to provide an integrated view of all collected quality data in order to assess the quality of a system's artifacts with respect to the defined quality requirements. Tools that provide such an integrated view are usually referred to as *software quality dashboards* or *software quality cockpits*. However, as the discussion of existing tools in the chapter on related work points out, there currently is no quality dashboard application available that satisfies the requirements of a truly integrated quality control process as advocated in this thesis.

To address this, we designed the *Continuous Quality Assessment Toolkit* ConQAT. ConQAT provides a rich and extensible library of building blocks that is used for the rapid development of customized quality dashboard applications. As such, ConQAT's main focus is not on the provision of new quality analysis methods but on the seamless integration of existing ones. However, as multiple examples show, ConQAT also provides a comprehensive basis for building novel analyses. In fact, we found that the border between reusing existing analyses and developing new analyses often blurs when an integrated toolkit is used. While ConQAT can be tightly coupled to QMM-based quality models (see Sec. 5.2.8), it can also be operated in conjunction with other quality models or even without an explicit definition of quality requirements. While we do not recommend this, we still leave the option open to not limit the contexts ConQAT can be used in.

5.2.1 Requirements

If quality dashboards are used to support continuous quality control as described in the last chapter, they must meet the requirements discussed below. The list of requirements is based on the shortcomings of existing analysis tools that were discussed in chapter on the state of the art. It is augmented by further requirements that were identified as part of our experience with quality control in several industrial and academic contexts.

Integration To provide an integrated overview of the current state of quality of a system, a quality dashboard should be able integrate multiple automatic quality analysis tools as well as the results of manual reviews. The combined results must be put into relation with the stated quality requirements.

Diversity The factors influencing product quality are diverse. Therefore a quality dashboard may not be limited to a certain type of factors or artifacts it analyses. It must not only include analyses for source code but should provide measures for other artifacts like documentation, models, build scripts or information stored in a change management system. Furthermore, each of these artifact types can be analyzed at different abstraction levels. Source code is a good example to illustrate this as each of the following abstraction levels serves as basis for different types of analyses:

- *Character Level.* Certain analyses like the determination of the file size or the search for specific textual patterns are carried out on the character level of a source code file.

- *Line Level.* The line level is e. g. used to determine the lines of code or to carry out a line-based clone detection.
- *Token Level.* A source code file can be tokenized by a lexical analyzer (scanner). These tokens serve as basis for analyses that need to differentiate different token types like identifiers, keywords, literals or comments. Examples are analyses that detect redundant literals or search for textual patterns in specific tokens like comments. An especially powerful analysis best carried out on the token level is clone detection.
- *AST⁴ Level.* Many analyses require knowledge of the syntax tree of a source file. Examples are the detection of anti-patterns or bug patterns or analyses that detect the misuse of certain language constructs.
- *Dependency Graph Level.* Other analyses require a representation of the source code that is above the syntax tree as they focus on the dependencies between program elements like methods and classes. Examples are the detection of unused methods or classes. Dependency graph-based analyses often do not only inspect single source files but rather look at subsystems or even the whole system.
- *Byte Code Level.* Many modern programming languages are not directly compiled to machine code but a *byte code* that is then executed in a virtual machine. While the byte code is not really an abstraction level of a source code file, it represents a relevant representation of it, as many analyses can be carried out on the byte code more easily and also more efficiently than on source code.

As all the above analyses can be relevant for the assessment of maintainability, a quality dashboard application must not be predisposed to a specific abstraction level.

Customizability Quality requirements are highly project-specific as the analyzed systems, the applied tools and processes, the involved technologies and the acting people differ. Even more so, these requirements are not constant but evolve over the course of a project. Hence, a quality dashboard must be highly customizable to support a project-specific tailoring of the analyses carried out and the way they are presented. Besides this, mechanisms to reduce the high number of false positives typically reported by analysis tools are of paramount importance as too high number of false positives are known to frustrate users. One needs to be aware that the continuous application of quality analyses significantly raises the bar for the quality of the analyses results. While false positives can be dealt with relatively easily for one-time assessments, we found that developers hardly tolerate more than 5% of false positives if they are required to work with the analysis results on regular, e. g. daily, basis.

In general, there are two ways to deal with false positives: *prevention* and *management*. Prevention aims at avoiding false positives altogether. This can be achieved by carefully selecting the analyses relevant in a certain context, i. e. by not blindly executing default analysis configurations that come with most tools. Furthermore, the analysis target must be chosen thoroughly, e. g. by excluding all generated code from analyses like clone detection. As practice shows, even this seemingly simple approach is technically challenging as generated code is often intermingled with handwritten code

⁴Abstract Syntax Tree

within source files. Another possibility to deal with false positives is to allow them in first place but provide mechanisms to mark them as irrelevant and henceforth exclude them from analysis results. We refer to this technique as black-listing. Again, this is technically challenging as black-listing should still work if elements the false positives were reported against are moved from one location to another. Depending on the type of analyses, very different technical solutions have to be applied.

Another type of customization regards the management of *tolerations*. In most systems continuous quality control was not applied from the beginning on but is introduced at some later point in the software life cycle. Hence, the initial application of quality analysis usually generates an enormous amount of findings even if false positives are treated correctly. As most organizations cannot cope with all findings at once, mechanisms are required to define tolerations that exclude particular finding from the analysis results and thereby enables an organization to clean up a system step by step. It needs to be stressed that, while the techniques to define these tolerations may be similar to ones used for dealing with false positives, the motivation to do so is fundamentally different.

Autonomous Operation Tool supported assessments need to be carried out regularly (e. g. daily or hourly) to provide timely results. To achieve this in a cost-efficient manner, analysis tools need to be able to work in a completely automated, non-interactive way.

Aggregation & Visualization Automated quality analysis of large software systems generates an enormous amount of analysis data. To not overwhelm users, analysis results need to be aggregated to a comprehensible level and presented in an appropriate manner. As discussed in the description of our quality metamodel in the last chapter, different types of aggregations are required for different purposes. Hence, a quality dashboard must provide different types of aggregation that can be flexibly composed to achieve the desired results. The provided aggregation mechanism must not be predisposed on a specific type of aggregation operator or a particular metric scale.

Even highly aggregated analysis results need to be conveyed to the user in an effective manner. Hence, the quality dashboards must provide powerful visualization mechanisms beyond tables and charts, e. g. graphs for structural information or tree maps to visualize distribution of anomalies within a system.

Dedicated Views The assessment results must be accessible for all project participants, e. g. on a website or via a specific client. However, due to differing interests, it must be possible to provide a customized view for each stakeholder. Project managers, for example, are mainly interested in a high-level overview that enables them to spot problems without going into details. Developers, on the other hand, require views of finer granularity that allow them to inspect analysis results for the artifacts they have been working on.

Trend Analysis Many quality defects are hard to identify by investigating a single snapshot of a system but can be discovered by tracking changes over time. Hence, the dashboard must be capable of storing and presenting historical data to foster the identification of trends. Moreover, various metrics used today are hard to interpret on an absolute scale but are well-suited if relative measures are used. It is, for example, not entirely clear what the quality implications of a code cloning ratio of 16% are,

whereas most quality engineers would agree that it is important to ensure that the cloning ratio does not increase over time⁵.

Extensibility As no tool can innately support the whole spectrum of all possible artifacts it must provide an extension mechanism that allows users to add further analysis or assessment modules as needed. Examples of artifacts that demand such extensions are models used in model-driven development or programs written in newly created domain-specific languages. Moreover, the extensibility should not be limited to adding new sensors but should also cover the addition of new aggregation mechanisms, filters, visualizations or other functionality that is not directly related to a specific sensor.

Performance As quality controlling is particularly relevant for large scale systems that rule out comprehensive manual assessments, a quality controlling tool must be able to cope with the analyzed system's size in acceptable time.

5.2.2 Design Considerations

To the best of our knowledge none of the tools available completely satisfies the requirements pictured above. We therefore designed ConQAT from scratch. Our design considerations are led by the requirements above and by the experiences we made with existing analysis tools as well as our own prototypes. A central decision was not to build a monolithic dashboard application but rather a toolkit or framework that provides building blocks to rapidly develop dashboard applications that are tailored to the specific needs of the given context. The requirement for extensibility made it evident that our toolkit must provide a flexible extension mechanism that allows to add new building blocks. These extension blocks can carry out various analyses that were previously unthought of but should nevertheless be composable with other blocks.

The main challenge here was the design of an architecture which is rigid enough to allow the efficient combination of different analyses while being flexible enough to integrate the plethora of different kinds of analyses. Detailed analysis of different extensible architectures pointed out that there is in fact a *spectrum of flexibility*. However, there's always a trade-off between the flexibility and the expressiveness of the extensions. On one end of the spectrum, one finds architectures that are extremely rigid. They define a very stringent extension interface and thereby limit the extensions' expressiveness. Nevertheless, they allow a flexible composition of the extensions and permit a rich infrastructure in the architectural core of the system. On the other end of the spectrum, one finds architectures that define a very unspecific interface and integrate their extensions only loosely. This enables extensions to be much more powerful but limits composition possibilities and inhibits a rich common infrastructure.

To obtain a better understanding of this spectrum, we developed two prototypes close to both ends of it. The one on the *rigid* end basically supported a mapping from compilation units to numerical

⁵The cloning ratio measures how much of the source code has been copied at least once. It thus provides an estimate of how likely changes will have to be performed in multiple places due to code duplication.

metric values⁶. Obviously this mechanism allows very efficient composition of different analysis modules but limits the range of analysis types. It doesn't support metrics which yield anything but a numerical value (without cumbersome workarounds) and makes analyses with a granularity different from compilation units impossible. The prototype at the other end of the spectrum was more or less a web portal which allowed the extensions to contribute HTML pages with their analysis results. It should be clear that this approach allows almost unlimited possibilities for the extensions but makes a meaningful composition of different extensions nearly impossible.

5.2.3 Architecture

Our analyses and experiments showed that finding the »right spot« on this spectrum was impossible due to the multifaceted nature of quality factors. Whenever we came up with a seemingly suitable set of interfaces, a new requirement for a specific quality analysis revealed another deficiency. Though this could be attributed to a lack of skills on our part we are convinced the problem is caused by the great number of diverse analysis types a system like this must support. We therefore opted for a solution that avoids picking a fixed spot on the *flexibility spectrum* and thereby limiting the system's versatility. Central idea of the selected solution is to specify interfaces that are general enough to support literally every kind of analyses and let users of the framework work out more precise interface definitions for components that allow a meaningful composition.

These considerations finally led to the design depicted in Fig. 5.3. The central element of ConQAT's architecture are *processors* that are interconnected in a pipes-and-filter oriented style. These processors highly diverse functionality and work like functions that accept multiple inputs and produce a single output. The *Driver* component is responsible for configuring the processor network and passing information from one processor to another. Processors may access external data like the file system or databases either directly or using one of the provided *Libraries*.

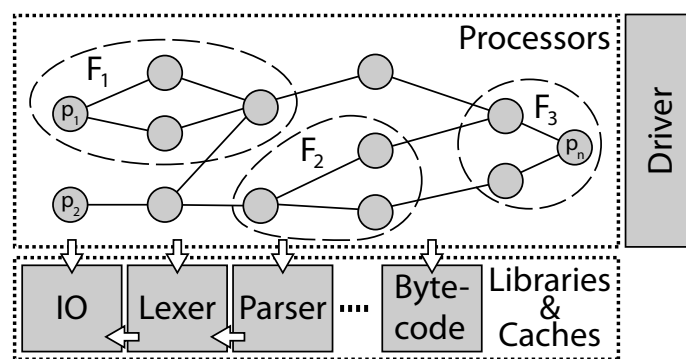


Figure 5.3: Architectural Overview

A simple example for composing an analysis of multiple processors is depicted in figure 5.4. Purpose of this analysis is to determine the average length of methods and to assess it with regard to a threshold. The analysis is composed of 7 processors that perform highly diverse and dedicated

⁶This prototype is still available at <http://www4.cs.tum.edu/~ccsm/svat/> but no longer maintained.

tasks. Processor *Scope* analyzes the file system and records the directory structure for all source code files that match a certain naming pattern. This tree-like data structure is forwarded to processors *LoC* and *#Methods* which determine the lines of code of each source file respectively compute the number of methods whereas the latter uses a parser or byte code analyzer (provided as library). Processors *#Methods* and *LoC* both annotate the original data-structure with integer values describing the results of their analyses and hand them to processor *Div*. This is a very simple processor which solely computes the average method length for each source file. Processor *Assessment* assesses the average method length with regard to a predefined threshold and rates each source file on simple traffic light scale with either GREEN, YELLOW or RED. Processor *Aggregator* aggregates these assessments from the leaves to the root of the tree, i. e. nodes that have RED child nodes are themselves rated RED. Finally processor *Output* writes the results to a file with a suitable format like HTML.

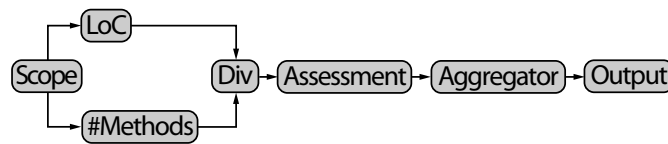


Figure 5.4: Processor composition example

The type of data exchanged between processors is purposely unspecified to allow greatest possible flexibility. As ConQAT and its processors are implemented in Java, it is actually defined as *java.lang.Object*. Nevertheless, processors must define their concrete interfaces by means explained below. Our hope was that during the continuing extension of the tool, *families of processors* with matching concrete interfaces would emerge as indicated in figure 5.3 by the dashed »clusters« denoted with F_i . Fortunately our assumption was confirmed very quickly: After implementing a couple of processors, the desired *families* emerged. Examples are processors that perform calculations on scalar values as typically done when processing the results of metric analyses. These processors have no knowledge of the origin of the values they process and can thereby be flexibly combined and reused in all situations that demand basic calculations. Another example are processors that deal with the »traffic light assessments« we typically use. Besides assessing the results produced by other processors, they are specialized in aggregating and filtering assessed results.

Experience shows that interfaces within processor families remain stable after a certain tuning phase due to their relatively limited scope. This allows flexible organization of analyses by composing processors in different ways. An obvious example are the processors that deal with scalar values. By implementing a set of processors that perform basic calculations, more complex calculations can be performed by composing processors. Note that formally the computability expressed by composition is limited as we don't allow recursive calls to the processors. In practice this proves to be of no significance since each processors may implement every computable function. Equally important are the interfaces between different families of processors. This is best illustrated by the following example. There is a family of processors that perform code audits like checking code format conventions or finding dubious pieces of code like empty blocks. These processors create lists with audit warnings for each source file. A simple interface between these processors and the ones described above is a processor that counts the number of warnings for each source file. This number may then act as input to further processors which perform calculations on it or assess it with regard to predefined rules.

Over time this approach led to the modularization which we weren't able to design from scratch due to the great diversity of requirements. Such an evolutionary approach demands measures of control to ensure success and avoid undesired developments. Problems that typically arise and which we experienced as well are »bloating« functionality of single processors and redundancy as two or processors implement the same or overlapping functionality. We counter these effects with precisely the same continuous quality control measures we advocate in this thesis. This involves clone detection, static checks for architecture violations combined with manual reviews. From the very beginning, we used ConQAT in a bootstrapping manner on itself to integrate these activities.

Central to these activities was the identification of commonly used functionality and moving it to libraries that can be accessed by all processors. These libraries form a central point of entry to the analyzed system's artifacts and thereby allow the implementation of efficient caching strategies. As it is very likely that different processors will use e. g. the AST of a particular compilation unit, the AST will be cached for future use and needs to be built only once. All ConQAT libraries use caching mechanisms that greatly reduces analysis time. To further improve performance the libraries are built on top of each other (if reasonable). For example, the parser library uses pre-cached tokens from the lexer library. All caches are implemented in a memory-sensitive way and support dynamic uncaching if the system is short of memory.

5.2.4 Configuration

The evolution of the architecture must be supported by a solid technical basis that inhibits uncontrolled growth of the interfaces. Typically one would expect that our decision to loosely specify the interfaces between the processors would result in a mess of explicit cast operations and the accompanying inevitable cast errors. Indeed the problems arising from the unspecified interfaces initially made our approach look infeasible. After implementing about 15 different processors we realized that the problems of non-explicit interfaces and the required explicit type casts introduced too many sources of errors to achieve a well maintainable system. We therefore developed a novel solution which we regard powerful and elegant. As we consider it essential for the success of ConQAT, this solution is presented in detail along with ConQAT's configuration mechanism.

We found that the mechanism that allows users and extenders of ConQAT to configure composed analyses from simple building blocks (the processors) is crucial. In early prototypes this configuration was simply done by hard-coding the configuration with Java. As even the most simple re-configuration of the system demanded modification of source code, re-compilation and re-distribution of the whole system it became evident that this approach is not an option for a system whose central requirement is flexibility. We therefore moved to a solution that employs a declarative XML configuration file to describe the interconnection of processors. This resembles the mechanisms typically used by extensible architectures like the Eclipse platform. To understand how this configuration mechanism work, we first provide details on the interface of the processors.

Processor Interface As our processors are basically functions, their interfaces could be described by a single method:

```
Object process(Object[] parameters);
```


This interface precisely displays the problem discussed before: it is in fact untyped. As implementations can hardly perform any real work on objects with type *Object* they need further knowledge of the actual type of the parameter objects. Processor composition is further hampered by the fact that the result type of a processor is unspecified. With the new features of Java 5 the latter problem can be solved relatively easily using covariant return types. Covariant return types allow implementers of an interface to refine the return types of methods by using a subclass of the original return type, e. g.:

```
Integer process(Object[] parameters);
```

Unfortunately, the former problem can't be solved as easily since covariant method parameters are unsafe and therefore not supported in Java. The central idea of our solution to this problem is to omit input parameters in the interface and leave their definition up to the processors. This is achieved using Java's annotation mechanism⁷.

Two example processor implementations are shown on the right hand side of Fig. 5.5. Both processors implement the parameterless method *process* and define their result types by using covariance. Processor *FileSystemScope* is responsible for scanning a given directory path for all Java source files and creating an *IFFileSystemElement*-object that describes the resulting directory tree. The task of processor *LOCAnalyzer* is to annotate each leaf element of this tree object with the number of lines of code the corresponding source file has. Obviously this processor needs an object of type *IFFileSystemElement* to work on. Therefore it defines a method

```
void setRoot(IFFileSystemElement root);
```

and annotates it as a *@AConfigElement*. This annotation informs the ConQAT runtime system that the annotated method is meant to provide an input parameter. Additionally one may use the annotation to specify further details e. g. if this parameter is mandatory or not.

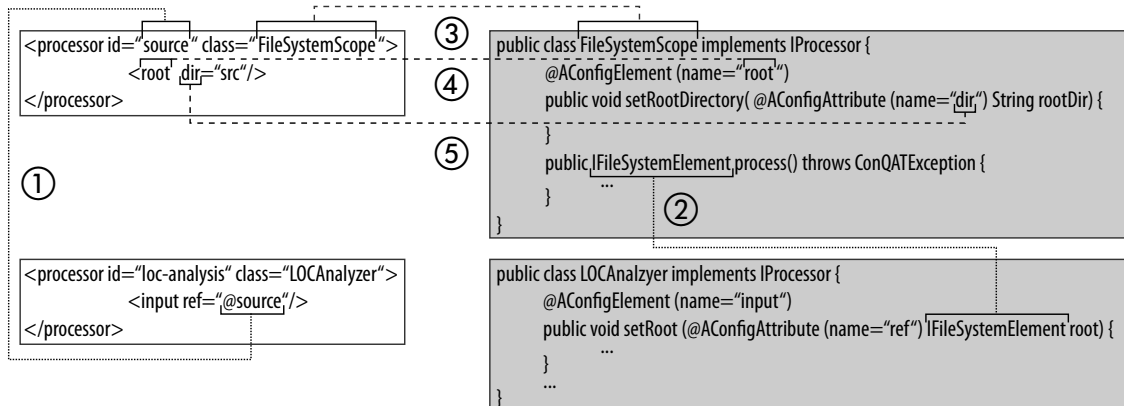


Figure 5.5: Mapping from Configuration File to Implementation

⁷<http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>

Type Safety To fully grasp the benefit of this approach, one must understand how the connection of different processors is configured with ConQAT. A typical configuration file is shown on the left hand side of Fig. 5.5. It defines the processors named »source« and »loc-analysis« where the latter one is connected to the former one by referencing its name (»@source«). This connection is indicated by line ①. This configuration implies a corresponding connection between the implementation of the two processors as shown by line ②. Now the advantage of this mechanism becomes evident: ConQAT's run-time system can make sure that two connected processors have matching interfaces by using Java's reflection mechanism. In fact, ConQAT refuses to run an analysis if the interfaces of connected processors do not match. The advantage of this approach is that type safety needs to be ensured only once before running the first analysis. We call this approach »load time type checking« (opposed to compile time or runtime type checking).

Besides this, the figure also shows that there is a defined mapping between the configuration file and the processors' implementations. Line ③ shows that the processor implementation is referenced by specifying the class name in the configuration file. Lines ④ and ⑤ exemplify how XML-elements are mapped to the corresponding input parameter methods: By using annotations, class *FileSystemScope* states that it expects only one configuration element called »input« that has exactly one attribute »root« with type *String*. In contrast to parameter »@source«, which describes a reference to the output of the processor *source*, »src« is an immediate parameter since strings can be provided in the configuration file itself. In fact, annotations in processors are not only used to ensure type-safety and allow a defined mapping to the declarative configuration, file but also provide a basis for automated generation of processor documentation.

Blocks ConQAT configurations often contain repeated elements. A simple example is a configuration that describes the analysis depicted in Fig. 5.4 for multiple different systems or subsystems. Designing such an analysis results in duplicating large parts of the configuration (Fig. 5.6). To prevent this, ConQAT provides an abstraction mechanism called *block* that allows to group processors. This modularization does not only reduce configuration redundancy but provides means to build reusable groups of logically coherent processors. To support reuse ConQAT also provides mechanisms to document such blocks accordingly. The block mechanisms is hierarchical so that blocks again can contain blocks.

Blocks are configured in the same manner as »normal« ConQAT configurations are. The only difference is that they have dedicated inports and outports to explicitly describe their interface. These ports can also be documented. Based on our experience, we designed blocks so that, in contrast to processors, they cannot only have multiple inports but also multiple outports. This allows to build configurations as shown in Fig. 5.6 where the result of the assessment as well as the raw metric values are forwarded to the output component.

Runtime The driver component is responsible for interconnecting the processors as defined in the configuration file and running the analysis. During start-up the driver unfolds the blocks, loads the processors' classes via reflection and uses their meta data to ensure type safety. It thereby ensures that the configuration is valid before starting the analysis. Processors are topologically sorted and then ran one after another. During execution, the driver passes the result from one processor to the next. If a processor's result is used more than once, the driver is responsible for cloning it. It additionally

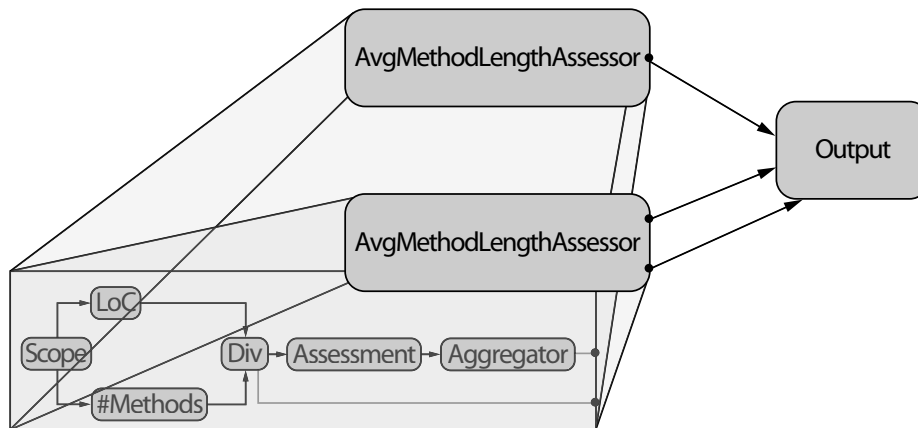


Figure 5.6: Configuration Blocks

performs some monitoring tasks to provide debugging information if one of the processors should fail.

5.2.5 Modularization & Feature Overview

At the time of writing this thesis, ConQAT provides around 300 processors and 80 blocks that provide features as different as database access to store analysis results or the detection of clones in Matlab Simulink models. To be able to manage this diversity, a proper modularization mechanism is of paramount importance. To achieve this, ConQAT provides so-called *bundles* that group processors, blocks and libraries that belong together. These bundles can be viewed similar to what other extensible architecture like the Eclipse platform call *plugins*. Consequently, they not only serve the modularization of ConQAT but also provide the means to extend it. Technically, a bundle is nothing more than a collection of Java classes, ConQAT blocks and other resources like third-party libraries or images. Each bundle is accompanied with a simple XML file called the *bundle descriptor* that stores information about the organization that developed the bundle and its current version. Furthermore, the bundle descriptor describes dependencies to other bundles.

Currently, ConQAT has about 30 bundles. Some of these are of highly experimental nature and some are developed for specific industrial partners and, hence, contain functionality that is relevant only in the context of a specific environment. Figure 5.7 gives an overview of the most important bundles and illustrates bundle dependencies by putting bundles that depend on others on top of these. The size of the depicted bundles was chosen for layout reasons only and does not reflect the amount of functionality provided by the bundles. The following sections describe the functionality of these bundles and thereby give an impression of ConQAT's current functionality⁸.

⁸Almost all of the bundles make heavy use of existing third-party libraries, e.g. for parsing Java source or byte code. Although a lot ConQAT's functionality could not have been implemented without these libraries, we do not discuss the individual libraries to keep the description of the bundles as concise as possible.

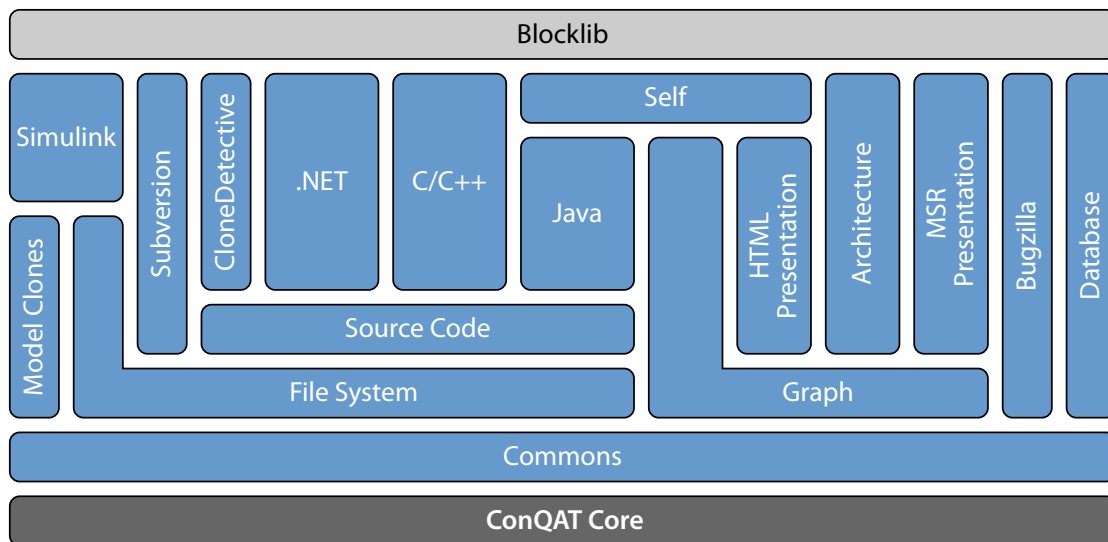


Figure 5.7: ConQAT Core & Bundles

ConQAT Core Shown at the bottom of the figure is the *ConQAT Core* which itself is not a bundle but a normal Java application. The core defines the interfaces for the processors and contains the *driver* that is responsible for reading ConQAT configurations and executing them. However, the core does neither provide processors, blocks or libraries. Hence, it is not specifically geared to any domain and could technically be used to execute ConQAT configurations completely unrelated to quality assessment. If ConQAT was viewed as a language, the core would be the compiler and the virtual machine.

Commons The *Commons* bundle provides processors and blocks that have been identified to be beneficial for a wide range of applications. Most importantly, the *Commons* define the Java interface *IConQATNode* that is used by the majority of processor to exchange information although processors can technically use any Java type. The interface *IConQATNode* is of very generic nature and merely describes a tree of nodes where each node can store arbitrary values identified by a key. Hence, its central methods are *getChildren()*, *getValue()* and *setValue()*. Other bundles can create specializations of this interface to describe more concrete structures, e. g. a tree of Java source files.

Additionally, the *Commons* bundle provides around 60 processors that work on *IConQATNodes* and simpler data types like strings. Examples are processor for filtering nodes based on values stored at them or processors for sorting trees based on specific properties. Moreover, the *Commons* bundle offers processors that implement simple aggregation operators like minimum, maximum or summation and processor for basic arithmetic functions. More advanced aggregations are provided by statistics processors that perform rank-relation calculations like the determination of percentiles for a collection of metric values. Besides this, the bundle provides a rich library for dealing with *IConQATNode*-trees, e. g. for traversing or conversion to other data structures. These libraries can be used from all bundles that depend on the *Commons* bundle. If ConQAT was viewed as a language, the *Commons* bundle would be the standard class library of the language.

File System The *File System* bundle provides processors and blocks that implement functionality that handles file system access. This includes a processor that scans a file system for files whose names match a certain pattern and builds a tree of *IFileSystemElements* from it which is a specialization of *IConQATNode*. Based on these file system nodes other processors implement functions like counting the lines of a file, determining its size or its last modification time. Special filters are provided that take into account the file content by searching for specific patterns in text files. An example for a more advanced processor is the *DuplicateFileAnalyzer* that detects binary identical files by comparing their MD5⁹ hashes. The bundle furthermore provides a library for accessing files that implements a cache. Hence, processors that access the content of a file do not have to re-read a file if it has been processed before. The cache is implemented in a memory-sensitive manner and supports dynamic uncaching if the system is short of memory.

Source Code The central role of the *Source Code* bundle is to provide lexical analyzers for different programming languages. Through these, it can currently tokenize source code files in Java, C/C++, C#, Visual Basic, VB.NET, PL/1 and COBOL. These scanners are implemented in a library that also caches the token streams to avoid redundant scanning of the same files. Using the token stream, the bundle provides, amongst others, processors to search for textual patterns in specific token types like comments and to detect redundant literals (e. g. magic numbers) in source code.

Java The *Java* bundle builds on the *Source Code* bundle and provides a parser to access the syntax tree of Java files and a byte code analyzer to access their byte code. The bundle provides processor to calculate basic metrics like *depth of inheritance tree* and *number of methods*. Search for well-known anti and bug patterns is provided by processors that interface the popular convention checkers PMD¹⁰ and FindBugs¹¹. The result of unit tests can be integrated into dashboards through processors that read JUnit¹² test reports. Test coverage analysis is offered by a similar processor that interfaces the test coverage analyzer Cobertura¹³. Furthermore, the bundle provides sophisticated filters that e. g. allow to filter all Java interfaces or all classes that specialize a particular class. Another set of processors allows to assess JavaDoc comments and to create dependency graphs for Java classes.

C/C++ Due to the complexity of C and particularly of C++, the *C/C++* bundle does not provide a parser for these languages. Instead, the bundle provides processors to integrate assessment results from commercial C/C++ analyzers like Klocwork¹⁴ and the Siemens Code Inspector that was used in one of our projects. Additionally, the bundle contains a processor for extracting a dependency graph of a C/C++ system based on the *include* directives in the source code.

.NET The *.NET* bundle provides processors and blocks for the analysis of systems written in .NET languages like C# and VB.NET. This includes a processor that creates a source file tree from the project

⁹The Message-Digest algorithm 5 is a hash function defined by RFC 1321.

¹⁰<http://pmd.sourceforge.net>

¹¹<http://findbugs.sourceforge.net>

¹²<http://www.junit.org/>

¹³<http://cobertura.sourceforge.net/>

¹⁴<http://www.klocwork.com/>

description files used by the *Visual Studio* IDE which is used in most .NET-based projects. Moreover, the bundle contains processors to extract information from the .NET byte code. As there currently is no .NET byte code analysis library available for Java, this is implemented by calling a .NET analysis program whose results are forwarded to ConQAT processor written in Java. The bundle, furthermore, offers an integration of the .NET convention checker FxCop¹⁵.

CloneDetective The *CloneDetective* bundle contains the token-based clone detection tool that has been described in [168] and [167]. The clone detection tool is tightly integrated with ConQAT and makes use of advanced filtering features to tailor the clone detection in order to reduce the number of false positives. For example, the clone detective can use the filters provided by the *File System* bundle to exclude certain regions of files from the detection or, more sophisticatedly, use different normalization algorithms for different file regions. Moreover, a *blacklisting* mechanism is provided that allows to exclude specific clones from the detection results.

Model Clones The *Model Clones* bundles provides processors to carry out clone detection on graph-based models. Its implementation and application has been described in [76]. While thematically related to the *CloneDetective* bundle the two bundles do not share common functionality as the detection of clones in graphical models is fundamentally different from token-based detection of source code clones. The *Model Clones* bundles provides the basic algorithm for detecting model clones. However, this must be concretized for specific model types, e. g. Matlab Simulink.

Simulink The *Simulink* bundle provides a concretization of the model clone detection algorithm of the *Model Clones* bundle and can, hence, be used to detect clones in Matlab Simulink models. To achieve this, the bundle contains a parser for the Simulink file format. Besides the clone detection, this parser is used to implement convention checks similar to PMD or FindBugs for Simulink and Stateflow models.

Subversion The *Subversion* bundle provides processors to assess the Subversion source code management system¹⁶. These processors can be used to acquire information stored by Subversion, e. g. the authorship of files or to assess the conformance of commit messages to agreed patterns. The bundle also offers processors to analyze properties of the working copies of files stored on the local hard disk.

Bugzilla The *Bugzilla* bundle allows to access the widely-used issue tracking system Bugzilla¹⁷. This can be used to collect project progress information, e. g. the number of open issues or to assess process conformance. For example, it can be checked if issues are closed only by members of the quality assurance team.

¹⁵[http://msdn.microsoft.com/en-us/library/bb429476\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx)

¹⁶<http://subversion.tigris.org/>

¹⁷<http://www.bugzilla.org/>

Database The *Database* bundle provides access to relational databases that support the JDBC interface¹⁸. Its central purpose is to collect and store trend data in a database. To do so it provides processors that extract values stored at *IConQATNodes* and store them in a database together with a time stamp. Other processors are used to extract series' of time-stamped values from the database that can then be charted by one of the output bundles.

Graph The *Graph* bundle provides a specialization of the *IConQATNode* interface that describes hierarchical graphs and processors that create such graphs from data structures generated by other bundles such as the *.NET* or *Java* bundle. For example, it takes only two processor to create a hierarchical graph that describes the class dependencies of a Java system. To analyze these graphs, the bundle provides processors to determine standard graph metrics like *fan-in* and *fan-out* as well as sophisticated centrality metrics like the famous *Page-Rank*. Moreover, the bundle contains processors to determine clusters in graphs and to detect cycles.

Architecture The *Architecture* bundle builds on the *Graph* bundle and allows to compare a system's actual architecture to an intended architecture in order to find architecture violations. To achieve this, the bundle implements a simple XML-based language for the definition of intended architectures. As the implementation of the bundle is purely based on graphs, it is neither depended on a specific programming language nor on a particular type of dependencies. Hence, architecture compliance analysis based on include directives in C can be carried out in the same way as compliance analysis based on call-dependencies in Java.

HTML Presentation The *HTML Presentation* bundle is the central bundle for displaying the analysis results. It provide processors to format *IConQATNodes* as tables and tree-tables as well as numerous processors for generating diagrams like bar charts, pie charts, line charts, radar and scatter plots. Additionally, the bundle provides processors for displaying hierarchical graphs and processors for laying out standard tree maps [265] as well as cushion tree maps [301]. The graph layout example in Fig. 5.8a shows architectural dependencies of a system where intended dependencies are colored green, prohibited dependencies are red and currently tolerated ones are yellow. Experience showed that the tree map visualization shown on the right of Fig. 5.8 is a powerful tool for visualizing analysis results with respect to the location within the system. In Fig. 5.8b, for example, each rectangle represents a Java class where the size of rectangle is determined by the LOC of the class. The color of the rectangle indicates the number of FindBugs warnings that was found in the class. As the position of the rectangles reflects the position of a class within the system decomposition, a quality engineer can use this view to quickly locate areas that exhibit more quality defects than others. This is further supported by tooltips that show information like the class name and details on analysis results.

Quality dashboards typically do not only show a single chart or table but consist of multiple views to display different quality characteristics for different parts of a system or even for different systems. To support this, the *HTML Presentation* bundle has a processor that creates a single HTML page that provides an overview of all analyses and serves as an entry point to the other views. An example of this page is shown in Fig. 5.9. As the center of the page shows, analysis results can be arranged in groups so that analyses that belong to the same system or are of similar nature are shown together. If

¹⁸<http://java.sun.com/javase/technologies/database>

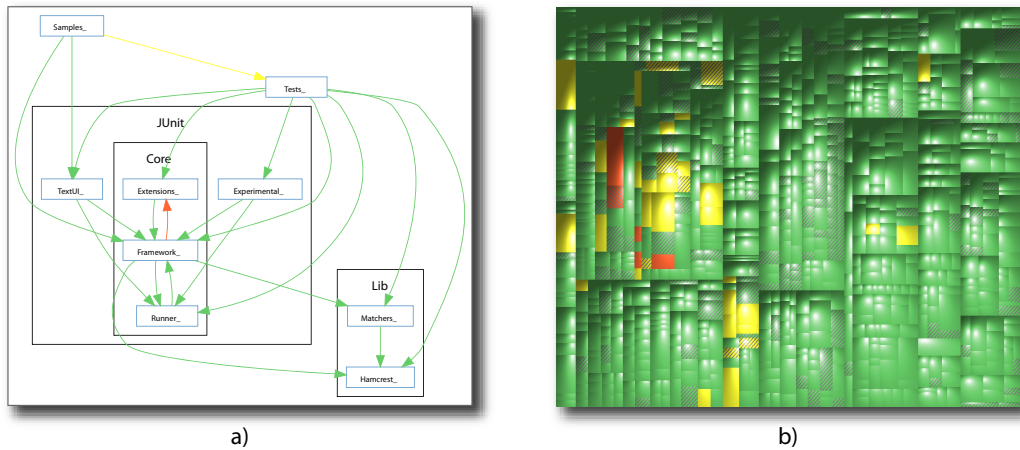


Figure 5.8: Graph Layout & Tree Maps

analyses use a traffic-light style assessment scale, the results are automatically aggregated and shown in the colored bars for each analysis. For each group, these assessment results are again aggregated and displayed as colored bar in the header of the group. The left frame of the main page contains a collapsible tree that is for navigational purposes, i. e. for directly assessing the results of analyses.

It needs to be stressed that all output generated by the *HTML Presentation* bundle is purely client-side HTML. Hence, no server component is required to display the results. This allows to easily copy the results from one machine to another and also facilitates straight-forward backups of results. Of course, the output can be equally simply copied to a web server to make it centrally available for all concerned project participants.

MSR Presentation As ConQAT uses the same flexible mechanism it uses for the analysis also for the output, other output processors can be defined and even run in parallel to the HTML output. One example is the *MSR Presentation* bundle that generates output files in the MSR MEDOC format. These files can be converted to the PDF format using XSLT. Hence, the *MSR Presentation* bundle allows to generate a single condensed report file as it is sometimes required in industrial contexts for archival purposes. Currently, the *MSR Presentation* bundle does not support all the charts and diagram types offered by the *HTML Presentation* bundle as they have either not been implemented yet or do not work well with a non-interactive format like PDF.

Self The *Self* bundle is specifically designed to analyze ConQAT itself. For example, it allows to detect dependencies between bundles that exist transitively but are not made explicit in the bundle descriptors of the concerned bundles. Similarly, dependencies that are defined but not actually required can be detected. Another analysis concerns the existence of duplicate classes: As bundles can include third-party libraries, it can happen that multiple bundles include the same third-party library, possibly in different versions. As this often leads to unexpected behavior, the *Self* bundle provides a processor to detect classes that are loaded through multiple bundles. The *Self* bundle is a

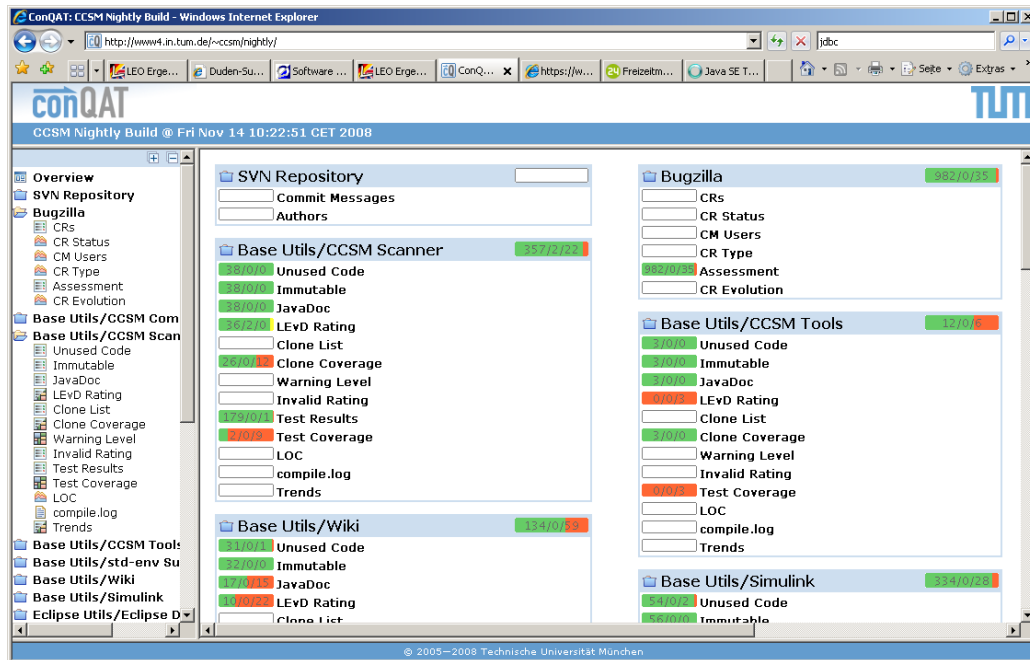


Figure 5.9: Main Page Generated by the HTML Presentation

prime example for a bundle that contains processors that were specifically designed to assess quality characteristics that are relevant in a certain context.

Blocklib The real power of ConQAT is only unleashed by combining processors and blocks from different bundles. An example is the following chain of processors and blocks: The *Java* bundle reads all files of a Java system from the disk and filters all files that are subclass of a particular class as these are known to be generated. The *File System* bundle further filters files that contain a certain textual patterns as these files are generated, too. The remaining files are analyzed for redundancy using the *CloneDetective*. The clone detection results are written to database to store trend data with the *Database* bundle. Finally, the results are visualized as a trend curve as well as a tree map that shows the location of clones with the *HTML Presentation* bundle. If this functionality is to be encapsulated in a block, it is unclear in which bundle the block should reside as it depends on 5 bundles that not naturally depend on each other. To support creating such reusable cross-bundle functionality the *Blocklib* bundle serves as a generic container. The *Blocklib* bundle has dependencies to all other bundles and currently contains about 40 blocks.

5.2.6 Documentation

The large number of available bundles, blocks and processors requires a comprehensive and up-to-date documentation to make ConQAT usable. To achieve this, ConQAT offers the documentation generator ConQATDoc to generate hyperlinked documentation in the HTML format. ConQATDoc is similar to the JavaDoc [110] system that is widely used for Java frameworks. To generate documentation

ConQATDoc automatically traverses a ConQAT installation and extracts all bundles, processors and blocks. For each bundle, it uses the information described in the bundle descriptor plus additional information that may be stored in a file called *bundle.html* to generate an overview of the bundle. This overview contains the bundle's dependencies as well as a list of all blocks and processors provided by the bundle (Fig.5.10a). For processors and blocks, it presents a description of the supported parameters (Fig.5.10b). The information required to generate the documentation is extracted from the annotations in the Java code. To support this, ConQAT requires each processor, parameter and attribute to be equipped with a human readable description. Integrating the implementation and the documentation of processors is advantageous as it helps to avoid redundancy between documentation and implementation and thereby fosters the up-to-dateness of the documentation.

Java Bundle

Processors for analyzing Java

Name	Java Bundle
Id	edu.tum.cs.conqat.java
Version	1.3
Provider	Technische Universitaet Muenchen
Location	F:\conqat-distros\dist\edu.tum.cs.conqat.java
Required Core Version	2.3
Dependencies	edu.tum.cs.conqat.commons (Version 1.3) edu.tum.cs.conqat.sourcecode (Version 1.3) edu.tum.cs.conqat.filesystem (Version 1.3)

This bundle provides a special scope for reading Java files and their byte code as well as a multitude of processors for analyzing and assessing them.

Included Processors

CoberturaCoverageAdapter	This processor reads and parses a test coverage result file generated by Cobertura (http://cobertura.sourceforge.net/) and annotates nodes accordingly.
JavaDocAssessor	This analyzer checks if JavaDoc comments for types, methods and fields are present. If not present a message is added at key 'JavaDocMessages'. Every leaf element is assessed GREEN unless it has any kind of messages. A list of fields to ignore may be specified.

Processor JavaDocAssessor

This analyzer checks if JavaDoc comments for types, methods and fields are present. If not present a message is added at key 'JavaDocMessages'. Every leaf element is assessed GREEN unless it has any kind of messages. A list of fields to ignore may be specified.

Full name: edu.tum.cs.conqat.java.assessment.JavaDocAssessor
Contained in: Java Bundle

Keys

Name	Type	Description
JavaDocMessages	List<String>	Key for Javadoc Messages
JavaDocAssessment	Assessment	Key for Javadoc Assessment

Output Type

edu.tum.cs.conqat.java.scope.IJavaElement

Parameters

Name	Multiple	Description
include-type	[0, INF]	Add modifier that describes which types (classes) should be included in the analysis. If no modifiers are specified all types are analyzed.
input	[1, 1]	The input this processor works on.

Figure 5.10: ConQATDoc Example

5.2.7 Configuration Editor »cq.edit«

Advanced ConQAT configurations often contain multiple dozens of processors and blocks. As creating and maintaining these configurations in the XML format is tedious and error-prone, ConQAT provides the graphical configuration and block editor *cq.edit*. This editor consists of multiple Eclipse plugins and can, hence, be run within an existing Eclipse installation or as a standalone application. The editor provides access to all blocks and processors of a ConQAT installation and allows to create configurations or blocks using drag&drop. Connections between processors are also defined using drag&drop. *cq.edit* applies ConQAT's type checking system to ensure that the built configuration does not violate the type safety. Detected problems are presented to the user with the native mechanisms of the Eclipse platform, i. e. through markers and the problem view. *cq.edit* support users in finding the appropriate blocks or processors through a search mechanism and the integration of their ConQATDoc documentation.

If *cq.edit* is used as plugin for the Eclipse Java development environment, the editor directly links processors to their implementation and, hence, also supports the development and maintenance of ConQAT processors. This is completed by a set of ConQAT-specific wizards that serve the creation of new processors and their parameters as well as a specialized editor for ConQAT bundle descriptors.

However, it needs to be pointed out that `cq.edit` is a relatively recent development that has not yet reached the maturity of ConQAT itself. Consequently, certain functionality is still missing. Despite these shortcomings, all current `cq.edit` users agree that it chiefly improves productivity with respect to the manual editing of ConQAT configurations.

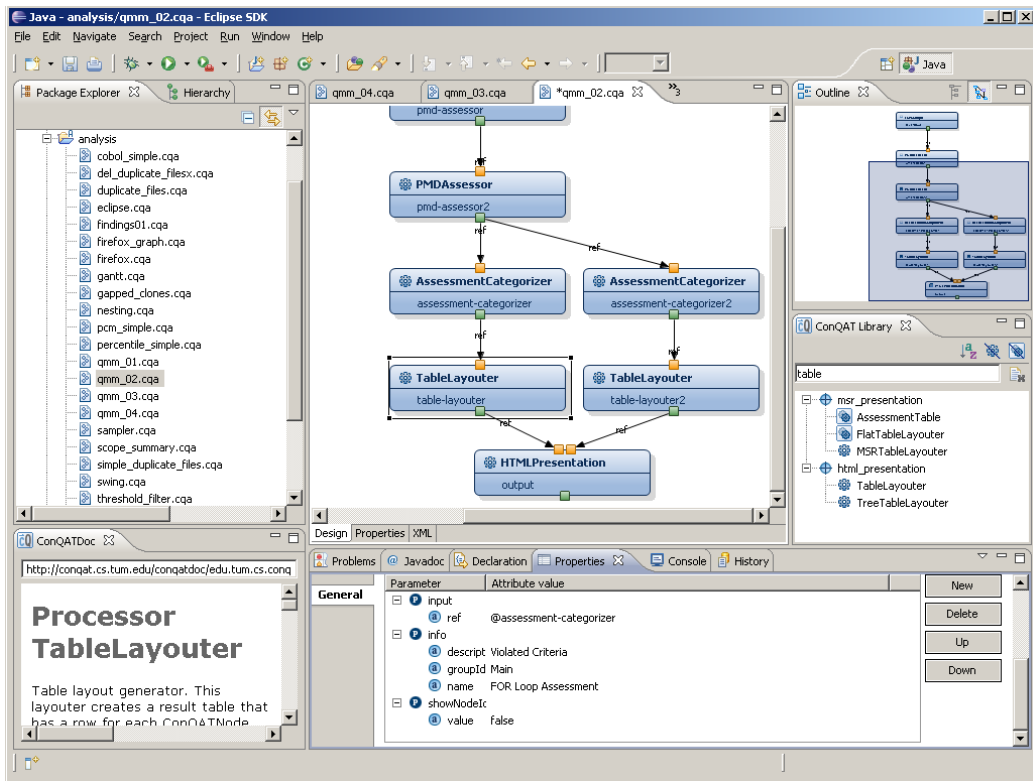


Figure 5.11: Screenshot of the ConQAT Editor `cq.edit`

5.2.8 Integration with the QMM

What has been described of ConQAT so far is independent of the quality modeling approach proposed in this thesis. As we see the lack of integration with an explicit definition of quality as one of the central shortcomings of quality analysis tools, this section explains how ConQAT can be integrated with the quality metamodel QMM to play the role of the *Q-Relator* advocated in Sec. 4.3.2. To achieve this, ConQAT provides the bundle *QMM* that uses the QMM Core library (Sec. 5.1.5) to load an existing quality model and relate it to the analysis results generated by other ConQAT processors.

Examples of quality reports generated by the *QMM bundle* have already been shown in Fig. 4.13. This section details on the integration of ConQAT and the QMM by describing a ConQAT configuration for the example quality model discussed in Sec. 4.2.4. The configuration depicted in Fig. 5.12 contains the *JavaScope* processor that scans a given directory for all Java source files and generates a tree-like data structure that consists of the Java packages and classes. This data structure is forwarded to the *PMDAssessor* processor that employs the open source analysis tool PMD to assess the QMM

fact [Body | WELL-FORMEDNESS], i. e. it checks if the loop body is properly enclosed in braces. To capture the assessment result, the processor annotates each class node in the source file tree; either with the traffic light color green if no violation was found in the class or with color red if a violation is found. In the latter case, the class node is additionally annotated with a message that details on the finding and its location. To assess the facts [Body | INTRICACY], [head | PURPOSIVENESS] and [counter variable | APPROPRIATENESS] the *ECJAssessor* is used which works similar to the *PMDAssessor*, i. e. it allows to run different types of assessments on the Java syntax tree¹⁹. For each of the facts the source file tree is annotated as described above. The facts [Head | PURPOSIVENESS] and [Body | INTRICACY] can be assessed only semiautomatically. Hence the processor also uses the color yellow to annotate classes where the processor could not automatically decide if the fact is adhered to.

This can be exemplified with the analysis that checks if the counter variable of a FOR-loop is modified within the body of a loop. If the counter variable is a local variable, the processor can check the modification quite simply and, hence, assess it automatically. If the counter variable, however, is a field of the enclosing class, the analysis is more difficult as methods that are called from inside the loop may change the counter variable. However, it can still detected direct manipulations of the variable in the loop and, consequently, rate the class red. If the variable is not modified and the FOR-loop does not call any methods, it can be safely rated green. In all other cases, however, the processor cannot automatically decide and rates the class yellow.

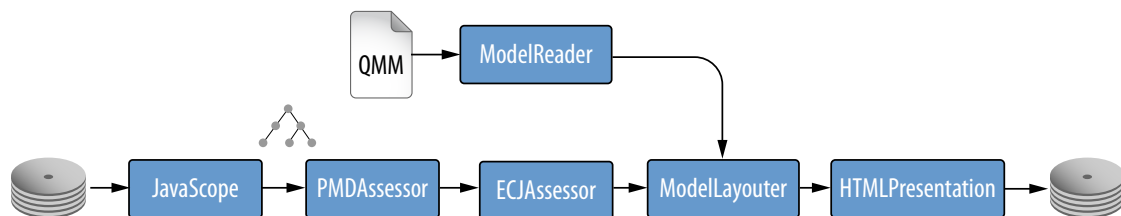


Figure 5.12: ConQAT Configuration for QMM-Integration

After the assessments were run by *PMDAssessor* and *ECJAssessor*, all source files are annotated with four assessment results for the four analyzed facts. To relate the assessment results to an explicitly defined quality model, the configuration contains the *ModelReader* processor that uses the QMM Core library to load a quality model stored in the XML format. The *ModelReader* provides an option to limit the loaded model to one or more working sets to keep the output concise. The loaded model is forwarded to the *ModelLayouter* that relates the assessment results to the quality model. To define the relation between assessment results and model facts the processor uses the keys where the assessment results are stored and the permanent ids of the facts defined by quality model.

The last processor of the configuration, the *HTMLPresentation*, writes the output to disk. It generates an HTML table as shown in Fig. 5.13. For each fact, the table shows the fact description and an assessment overview bar (on the right). The assessment overview bar visualizes the percentage of files that were assessed as red, yellow or green and also presents the actual figures. In the depicted example, 11 files were rated red with respect to fact [Body | INTRICACY], 115 were rated green and 16 were rated

¹⁹The *ECJAssessor* was specifically developed for ConQAT to implement Java source code analyses that are not supported by already existing tools. In contrast to PMD, the *ECJAssessor* employs the industrial-strength compiler provided by the Eclipse IDE (the abbreviation *ECJ* stands for *Eclipse Compiler for Java*).

yellow, meaning that the processor could not make a final decision here. Clicking the assessment overview bar points the browser to a new table that illustrates which files did not conform to which facts and also details on the exact nature of non-conformance. As fact [FOR-Loop | APPROPRIATENESS] must be assessed manually, the assessment overview bar for this fact remains empty and, hence, clearly signals that this fact requires further manual inspection.

Depending on the project context, different ways of presenting the conformity results may be required. Due to ConQAT's open and flexible architecture this can usually be realized on a configuration level, i. e. no programming is required. More sophisticated customizations may require implementation of new processors but can nevertheless be efficiently carried out as ConQAT already provides support for essential task like reading the quality model. One example customization is a configuration that generates a table with all violations for each file. With the help of the *MSR Presentation* bundle, this table can then be written to a PDF file to serve as prefilled review checklist as it was discussed in the last chapter.

For Loop Assessment (Model)			
For Loop Assessment			
Entity	Attribute	Description	Assessment
System			
FOR Loop		A for loop is a programming language statement which allows code to be repeatedly executed. A for loop is classified as an iteration statement. Unlike many other kinds of loops, such as the while loop, the for loop is often distinguished by an explicit counter variable. This allows the body of the for loop (the code that is being repeatedly executed) to know about the sequencing of each iteration. For loops are also typically used when the number of iterations is known before entering the loop.	
	Appropriateness (manual)	Each loop construct has a specific usage. A for loop is used only when the loop variable is increased by a constant amount for each iteration and when the termination of the loop is determined by a constant expression. In other cases, while or do-while should be used. When the terminating condition can be evaluated at the beginning of the loop, while should be used; do-while is used when the terminating condition is best evaluated at the end of the loop.	
	Body	The body of a FOR loop.	
	Well-Formedness (automatic)	The statement(s) which are conditionally selected by an if or else (or else if) statement, and the statements within the body of a while, do... while or for loop, shall always be in a block (enclosed within braces), even if they are a single statement.	126/0/16
	Intricacy (semi-automatic)	Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop. T	115/1/6
	Head	The head of a FOR loop is the FOR statement itself.	
	Purposiveness (semi-automatic)	The only expressions within a for statement should be to initialise, increment (or decrement) and test loop counter(s), and to test other loop control variables.	113/1/6
	Counter Variable	A counter variable controls the iterations of a FOR loop. It is so named because most uses of this construct result in the variable taking on a range of integer values in some orderly sequence (e.g., starting at 0 and end at 10 in increments of 1)	
	Appropriateness (automatic)	Floating point variables shall not be used as loop counters. A "loop counter" is one whose value is tested to determine termination of the loop. Floating point variables should not be used for this purpose.	128/0/14

Figure 5.13: Model Assessment Table

5.2.9 Summary

ConQAT's architecture proves to satisfy our requirements for a flexible yet efficient quality assessment tool: It runs in a non-interactive manner and generates static output in different formats. It is flexible and extensible by offering three different levels of configuration: analyses can be composed using a declarative configuration file, reusable blocks can be built from existing processors and new analyses can be added by implementing new processors. The system's design does not limit analyses to a particular scope, granularity, or type of artifacts and, hence, fulfills the requirement of diversity. ConQAT

allows to explicitly relate the assessment results generated by different tools to each other. In particular, the integration of QMM-based quality models allows to assess the conformity of a system to a specified quality model. In addition to that, ConQAT offers convincing performance characteristics due to the use of caching mechanisms. Even with a complex set of analyses, medium sized projects can be analyzed within matter of minutes. The analysis of large projects can be carried out as part of the nightly build process which is ConQAT's intended mode of operations.

In contrast to the QMM.editor introduced above, ConQAT is clearly beyond the status of a mere research prototype. As the chapter on case studies shows, its customizability and also its reliability led to its adoption in multiple industrial projects where it is used for quality control. Besides this, the open and flexible architecture also makes ConQAT a suitable candidate for research projects that experiment with quality assessment techniques. As such ConQAT has been employed for a number of high quality publications of our [76, 166, 167, 292] and foreign research groups [230].

Currently, one minor shortcoming is the graphical configuration editor `cq.edit`. While it provides most of the required functionality, it does not match ConQAT in terms of maturity and reliability. For a further dispersion and still wider acceptance of ConQAT, a renovation and completion of `cq.edit` is currently in the works.

5.3 Summary

The QMM.editor and ConQAT provide the necessary tool support to operationalize quality models based on the QMM. The following chapter on case studies reports on the successful application of both tools in several industrial and academic projects.

»Experience without theory is blind, but theory without experience is mere intellectual play.«

Immanuel Kant

6 Case Studies

This chapter describes the case studies that were carried out to evaluate the quality modeling and control approach presented in this thesis. Section 6.1 describes a case study undertaken with MAN Nutzfahrzeuge Group where a QMM-based quality model for the maintainability of Matlab Simulink models was designed. Section 6.2 describes the application of a QMM-based quality model to evaluate the maintainability of multiple web user interface frameworks. The case study described in Sec. 6.3 was undertaken with the BMW Group in the context of mainframe software development. It illustrates how the impact of a single fact on the maintenance effort can be evaluated in a quantitative manner. Section 6.4 describes how ConQAT was used to build quality control dashboards at the Munich Re Group and at ABB. Finally, Sec. 6.5 illustrates how manual and automatic quality assessments were integrated into a quality dashboard to control the development of cq.edit in a student lab course.

6.1 Model-Based Development of Embedded Systems (MAN)

This case study describes the application of the maintainability modeling approach, the generation of developer guidelines and the use of quality assessment tools in the context of model-based development in the automotive domain. The case study was carried out with the MAN Nutzfahrzeuge Group and led to the adoption of the quality modeling approach into the MAN standard development process. This study has partly been published in [85].

6.1.1 Environment

The MAN Nutzfahrzeuge Group is a German-based international supplier of commercial vehicles and transport systems, mainly trucks and buses. It has over 34,000 employees world-wide of which 150 work on electronics and software development. Hence, the focus of this study is on embedded systems in the automotive domain.

MAN brought its development process to a high level of maturity by redesigning it according to best practices and safety-critical system standards. Most parts of the process are supported by an integrated data backbone developed on the eASee framework from Vector Consulting GmbH. On top of this backbone, a complete model-based development approach has been established using the tool chain of Matlab/Simulink and Stateflow as modeling and simulation environment and TargetLink of dSpace as C-code generator.

Matlab/Simulink is a model-based development suite aiming at the embedded systems domain. It is commonly used in the automotive area. The original *Simulink* has its focus on continuous control engineering. Its counterpart *Stateflow* is a dialect of statecharts that is used to model the event-driven

parts of a system. The Simulink environment already allows to simulate the model for validation purposes. In conjunction with code generators such as Embedded Coder from MathWorks or TargetLink by dSpace it enables the complete and automatic transformation of models into executable code.

6.1.2 Goals

Goal of this study was to develop an explicit definition of quality requirements for Simulink models with a focus on long-term maintainability. Importantly, these requirements should be communicated to developers in the familiar form of quality guidelines. The assessment of conformity, which is achieved at MAN mainly through manual reviews, should be supported by additional automatic reviews.

6.1.3 Study Description

The study consisted of three work packages that were dedicated to defining the quality requirements with an QMM-based quality model, generating quality guidelines from it and developing automatic analyses to support conformity assessments.

Quality Modeling For the definition of a quality model for Simulink models, we started from a maintainability model we had developed in a project in the field of telecommunication. This model already covered various areas that we considered important for MAN, too. Examples are the parts of the model dedicated to architectural aspects or to the development process. We augmented the existing model with model elements that address Simulink/Stateflow-models that are used as basis for code generation. Although such models are seemingly different from traditional source code, we found that a great number of source-code-related facts could be reused for them as they fundamentally serve the same aim: to specify executable production-code.

Specifically, we extended the facts tree of the maintainability model with 87 facts (64 new entities and 3 new attributes) that describe properties of entities not found in classical code-based development. Examples are states, signals, ports and entities that describe the graphical representation of models, e. g. colors. The development of the entities tree was mainly carried out in a bottom-up manner by including relevant entities of the Simulink metamodel. Furthermore, we modified the activities tree to match the MAN development process and added two activities (*Model Reading* and *Code Generation*) that are specific for the model-based development approach. 84 impacts describe the relation between facts and activities. The newly developed parts of the maintainability model are based on three types of sources:

1. Existing guidelines for Simulink/Stateflow
2. Scientific studies about model-based development
3. Expert know-how of MAN's engineers

Our focus lay on the consolidation of four guidelines available for using Simulink and Stateflow: the MathWorks documentation [202], the MAN-internal guideline, the guideline by dSpace [94] (the developers of the TargetLink code-generator) and the guideline published by the MathWorks Automotive Advisory Board (MAAB) [195]. Due to confidentiality reasons, we are not able to fully describe the MAN-specific model here. However, we present a number of examples that demonstrate our approach.

We started with a simple translation of the existing MAN guideline for Stateflow models into the maintainability model. For example, the MAN guideline requires state machines to have an external output signal that describes the currently executed state of the state machine. This simplifies testing of the model and improves the debugging process. In terms of the model this is expressed as: [Stateflow Chart | ACCESSIBILITY] $\xrightarrow{+}$ [Debugging] and [Stateflow Chart | ACCESSIBILITY] $\xrightarrow{+}$ [Test]. We describe the ability to determine the current state with the attribute ACCESSIBILITY of the entity Stateflow Chart that describes a Stateflow state machine. In the model we carefully distinguished between the Stateflow Chart that describes logic of the state machine and the Stateflow Diagram that describes the graphical representation. As the examples show, we used the impact set $I = \{-, +\}$ to express positive and negative impacts of facts on activities.

Another example shows how new scientific results can be incorporated into the model: Up to now the use of Simulink and Stateflow has not been intensively investigated in terms of maintainability. However, especially the close relationship between Stateflow and the UML statecharts allows to reuse results. A study on hierarchical states in UML statecharts [68] showed that the use of hierarchies improves the efficiency of understanding the model in case the reader has a certain amount of experience. This is expressed in the model as follows: [Stateflow Diagram | STRUCTUREDNESS] $\xrightarrow{+}$ [Model reading].

Guideline Generation We regard quality models as central knowledge bases w.r.t. quality issues in a project, company, or domain. This knowledge can and should be used to guide development activities as well as reviews. However, the model in its totality is too complex to be comprehended entirely and can, hence, not be used as a quick reference. Furthermore, quality models as proposed in this thesis are not well-known among developers and cannot be dealt with by them without a substantial amount of training. Hence, it was a central requirement of MAN that the quality model must be transformed to the classic quality guideline format developers are familiar with. Additionally, the new guideline documents should be organized similar to the existing internal MAN guidelines, i. e. there should be separate document sections for Simulink- and Stateflow-specific aspects.

To cater for different audiences, the documents were split in two parts. The first is a very compact checklist-style section with essential information only. This representation is favored by experts who want to ensure that they comply to the guideline but do not need any further explanation. It is also used as checklist for manual reviews. For novices, the remainder of the document contains a hyper-linked section providing additional detail.

To be integrated into MAN's development infrastructure, the guideline documents had to be generated in the MSR MEDOC format that is generally used for all documents at MAN. These documents

can be converted to PDF documents and handed to the developers. An example of a generated guideline is shown in Fig. 6.1¹. The capability for generating guidelines in the MSR format was integrated in the quality model editor QMM.editor.


MAN Nutzfahrzeuge AG																				
	Autor: F. Deissenboeck Abtl.: CCSM	StateFlow Guideline 1 Checkliste																		
		Datum: 06.08.2007 Version: 1.0 Status: Review																		
1.1.4	Flowchart																			
	<table border="1"> <thead> <tr> <th>Id</th> <th>Regel</th> <th>Details</th> </tr> </thead> <tbody> <tr> <td>405</td> <td>Es dürfen keine Flowcharts verwendet werden.</td> <td>[S.10]</td> </tr> </tbody> </table>	Id	Regel	Details	405	Es dürfen keine Flowcharts verwendet werden.	[S.10]													
Id	Regel	Details																		
405	Es dürfen keine Flowcharts verwendet werden.	[S.10]																		
	Tabelle 5: Flowchart																			
1.1.5	History-Junction																			
	<table border="1"> <thead> <tr> <th>Id</th> <th>Regel</th> <th>Details</th> </tr> </thead> <tbody> <tr> <td>420</td> <td>Im StateFlow-Chart dürfen keine History-Junctions benutzt werden.</td> <td>[S.10]</td> </tr> </tbody> </table>	Id	Regel	Details	420	Im StateFlow-Chart dürfen keine History-Junctions benutzt werden.	[S.10]													
Id	Regel	Details																		
420	Im StateFlow-Chart dürfen keine History-Junctions benutzt werden.	[S.10]																		
	Tabelle 6: History-Junction																			
1.1.6	Zustand (State)																			
	<table border="1"> <thead> <tr> <th>Id</th> <th>Regel</th> <th>Details</th> </tr> </thead> <tbody> <tr> <td colspan="3">Allgemeines zu Zustand (State)</td> </tr> <tr> <td>412</td> <td>Jeder Zustand hat keinen oder mehr als einen Subzustand.</td> <td>[S.15]</td> </tr> <tr> <td>414</td> <td>Ein Zustand darf nicht überflüssig sein.</td> <td>[S.16]</td> </tr> <tr> <td colspan="3">UND-Zustand (is a Zustand (State) part of StateFlow-Chart is a Komponente part of Statik part of System is a Produkt-Artefakte)</td> </tr> <tr> <td>417</td> <td>Ein UND-Zustand darf nicht später aktiviert werden, als die Nachbarzustände, von denen er Ereignisse empfängt. Seine <i>Activation Order</i> muss also niedriger sein.</td> <td>[S.15]</td> </tr> </tbody> </table>	Id	Regel	Details	Allgemeines zu Zustand (State)			412	Jeder Zustand hat keinen oder mehr als einen Subzustand.	[S.15]	414	Ein Zustand darf nicht überflüssig sein.	[S.16]	UND-Zustand (is a Zustand (State) part of StateFlow-Chart is a Komponente part of Statik part of System is a Produkt-Artefakte)			417	Ein UND-Zustand darf nicht später aktiviert werden, als die Nachbarzustände, von denen er Ereignisse empfängt. Seine <i>Activation Order</i> muss also niedriger sein.	[S.15]	
Id	Regel	Details																		
Allgemeines zu Zustand (State)																				
412	Jeder Zustand hat keinen oder mehr als einen Subzustand.	[S.15]																		
414	Ein Zustand darf nicht überflüssig sein.	[S.16]																		
UND-Zustand (is a Zustand (State) part of StateFlow-Chart is a Komponente part of Statik part of System is a Produkt-Artefakte)																				
417	Ein UND-Zustand darf nicht später aktiviert werden, als die Nachbarzustände, von denen er Ereignisse empfängt. Seine <i>Activation Order</i> muss also niedriger sein.	[S.15]																		
	Tabelle 7: Zustand (State)																			

Figure 6.1: Generated Guideline (German)

Automatic Assessments To support the assessment of Simulink models with automatic analysis, we implemented the ConQAT *Simulink* bundle that provides a parser for Simulink models and multiple processors to analyze the models. Currently supported analyses include:

- Basic size and complexity metrics like number of blocks, number of states (in statecharts) or the nesting depth of states in hierarchical statecharts,
- An analysis to assess the conformance of visual model aspects to MAN's standards. Examples are block colors, block shapes and fonts used for labels.
- An analysis to determine if Simulink models are layouted in a way that the dataflow is mainly directed from left to right.
- An analysis that identifies hierarchical states that have only a single substate.

¹The example guideline is in German as this is the standard for MAN's documents.

- An analysis to determine the usage of library blocks across all MAN models.
- An analysis to find blocks that are not properly supported by the C-Code generator.

Next to this rather straightforward analyses, the first algorithm for finding redundancies (*clones*) in models was implemented with ConQAT. A detailed description of the algorithm, its application and the findings made at MAN can be found in [76].

The analysis of models with ConQAT is integrated with MAN's development process and tools. At specific process stages, all models are automatically checked out from MAN's data backbone and analyzed by ConQAT. The analysis results are written back to the backbone, stored and versioned there. To achieve this, we also implemented a new output component that generates reports in the MSR MEDOC format. An extract of a quality report in the MSR MEDOC format showing a model clone is shown in Fig. 6.2.

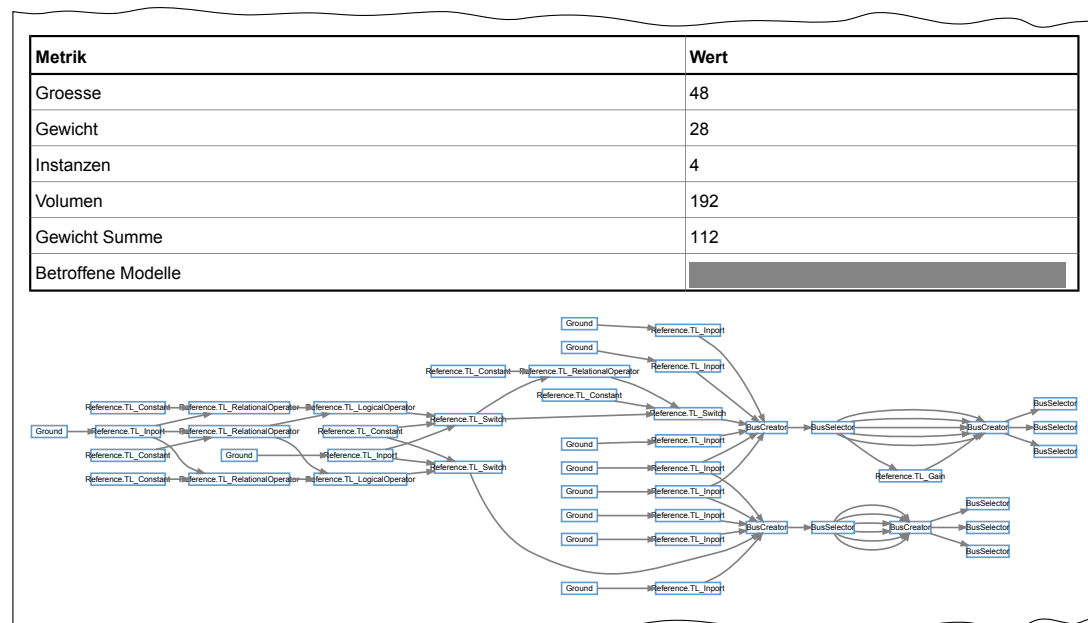


Figure 6.2: Example Quality Report Showing a Model Clone (German)

6.1.4 Results

For clarity's sake the results of the case study are discussed with respect to the work packages introduced above:

Quality Modeling Formalizing the existing guidelines, previous scientific results as well as the expert know-how of the MAN engineers led to the following results:

- *Consolidation of Terminology.* In the case study we found that building a comprehensive quality model has the beneficial *side-effect* of creating a consistent terminology. By consolidating the various sources of guidelines, we discovered a very inconsistent terminology that hampers a quick understanding of the guidelines. Moreover, we found that even at MAN the terminology has not been completely fixed. Fortunately, building a quality model automatically forces the modeler to give all entities explicit and consistent names. The entities of the facts tree of our maintainability model automatically define a consistent terminology and thereby provide a glossary. One of the many examples is the term *subsystem* that is used in the Simulink documentation to describe Simulink's central means of decomposition. The dSpace guideline, however, uses the same term to refer to a *TargetLink subsystem* that is similar to a Simulink subsystem but has a number of additional constraints and properties defined by the C-code generator. MAN engineers on the other hand, usually refer to a *TargetLink subsystem* as *TargetLink function* or simply *function*. While building the maintainability model, this discrepancy was made explicit and could be resolved.
- *Resolution of Inconsistencies.* Furthermore, we were not only able to identify inconsistencies in the terminology but also in content. For the entity *Implicit Event* we found completely contradictory statements in the MathWorks documentation and the dSpace guideline. While the MathWorks guideline claims that »Implicit event broadcasts [...] and implicit conditions [...] make the diagram easy to read and the generated code more efficient.« [202], the dSpace guideline states that »The usage of implicit events is [...] intransparent concerning potential side effects of variable assignments or the entering/exiting of states.« [94]. This obvious inconsistency was only discovered when both guidelines were translated to a QMM-based quality model. In this case, we adopted the dSpace point of view after consulting the MAN engineers.
- *Revelation of Omissions.* An important feature of the quality metamodel is support for inheritance. This became obvious in the case study after modeling the MAN guideline for Simulink variables and Stateflow variables. We modeled them with the common parent entity Variable that has the attribute LOCALITY that expresses that variables must have the smallest possible scope. As this attribute is inherited by both types of variables, we found that this important property was not expressed in the original guideline. Moreover, we saw by modeling that there was an imbalance between the Simulink and Stateflow variables. Most of the guidelines related only to Simulink variables. Hence, we transferred them to Stateflow as well.

The case study demonstrated the applicability of our metamodel and the corresponding method for modeling maintainability in an industrial development environment. After a short time, the 2-dimensional structure was accepted by the MAN engineers. Especially the model's explicit illustration of impacts on activities was seen as beneficial as it provides a sound justification for the quality rules expressed by the model. Moreover, the general method of modeling – that inherently includes structuring – improved the guideline: although the initial MAN guideline included many important aspects, we were able to reveal several omissions and inconsistencies. An important result was the creation of a consistent terminology as a side-effect of the quality modeling. This was perceived very useful by the MAN engineers.

Another insight we gained during the development of the quality model is that proper tool support for creating and maintaining quality models is indispensable. At the beginning of the case study, quality models were stored in a relational database with only minimal editing support. While this served

the purpose of guideline generation, it made creating and maintaining quality models very cumbersome. We found that especially in the early phase of quality model development, a lot of changes and additions are required that truly demand proper tool-support. This insight led to the development of the QMM.editor (see Chap. 5.1). The graphical editor also helped to convey the fundamentals of QMM-based quality models to MAN engineers as it helped to clarify the basic concepts.

Guideline Generation The generation of quality guideline documents turned out to be a challenge due to the following reasons. Importantly, some of these challenges are not directly related to our quality modeling approach but rather caused by the peculiarities of integrating it with an existing technical and social project infrastructure:

- Generating quality guidelines from a formal quality model leads to documents that sometimes lack the elegance of manually written documents. This is best illustrated with an example taken from the quality model described in Sec. 4.2.4: There the fact [Body | INTRICACY] describes that the counter variable of a FOR loop is modified within the loop body. As this is considered to increase program comprehension efforts, it has a negative impact on the activity Program Comprehension. Initially, the quality guideline generated for this fact consisted only of the description of the fact $\mathcal{D}_f([\text{Body} | \text{INTRICACY}])$ and its impact. Hence, the guideline document contained a paragraph:

Negative: Modification of counter variable within loop body.

However, MAN engineers are accustomed to guideline documents that are formulated as rules and expected a phrase like:

Do *not* modify the counter variable within the loop body.

As attempts to automatically generate such phrases from the fact description and the impact were not successful², we extended the QMM to include an instruction function \mathcal{D}_{F_I} that associates each fact with an explicit instruction phrase as in the example above. While this instruction is redundant to the fact description and the impact, we found that this poses no problems in practice as the redundancy is local to the fact and not spread across the quality model. We conclude, that certain compromises need to be made to establish a successful transition from a formal quality model to a classic guideline document.

- As it was required that guideline documents were prepared in the MSR MEDOC format, a guideline exporter for this format had to be developed and integrated into the QMM.editor. While this may be viewed as merely a technical problem, we found that conformance to this requirements was *vital* for successful integration of our quality modeling and control approach into MAN's development processes. Only after we were able to satisfy this requirement, MAN engineers could actually work with the generated guidelines. Hence, we conclude that the ability to seamlessly integrate with an existing project infrastructure is crucial for a quality modeling framework in an industrial context.

²This was further complicated by the German language used in the MAN quality models.

- Rather naively, we unexpectedly faced challenges that are clearly beyond any conceptual or technical problems but rather of sociological nature. So we found that it required significant effort to convince people of the benefits of the new guideline documents as they were accustomed to the previous guidelines for years. However, explaining the *whole* quality modeling and control approach instead of only presenting the new guidelines helped to overcome objections and to convince all participants of the benefits.

After the aforementioned challenges had been overcome, we found that automatically generated guideline documents provide the following benefits:

- By splitting the guideline documents in a short checklist-style part and a part that contains detailed explanations, we were able to optimally support seasoned developers as well as new project participants. As this kind of guideline is highly redundant by design, it would be very tedious to write and maintain them manually without introducing inconsistencies.
- As guideline generation is based on the working sets defined in the quality model, we were able to generate guidelines that are specifically tailored to the needs of developers. For example, developers not working with Stateflow can obtain a guideline that omits all Stateflow-related content. However, it is still ensured that this guideline is consistent with all other guidelines used by other developers. Again, this would be very tedious to achieve with manually written guidelines.
- Changes to the guideline structure that had been requested by developers could be implemented very efficiently by modifying the generator accordingly. This enables MAN to optimize the guideline structure without the need to manually change all guideline documents.

In summary, we found that generating quality guidelines is more challenging than expected due to the reasons described above. However, once these challenges are overcome, automatically generated guidelines are superior to manually written ones in terms of usability, consistency and flexibility. Moreover, the guidelines are also guaranteed to satisfy the requirements of assessability, structure and rationale (see Chap. 3) as they are directly derived from a QMM-based quality model that inherently satisfies them.

Automatic Assessments At the time of writing this thesis, the implementation of automatic assessments in the MAN development process has been carried out. However, too little experience has been made to give an detailed account of the results. Our findings so far are very much aligned with what we found in similar contexts (see Sec. 6.4):

- Automatically assessing simple criteria, e. g. colors and fonts used for blocks in Simulink models, is very beneficial as this is very tedious to achieve in manual reviews and can be done automatically with near-zero rates of false positives.
- Assessment of more complex criteria, e. g. clones in Simulink models, requires a high amount of customization to bring down false positive rates to a level acceptable for continuous quality control.

- Automatic assessments only prove to be of real benefit if they are tightly integrated into the existing process and tool landscape. Hence, significant effort was undertaken to seamlessly integrate ConQAT with MAN's data backbone.

While, at this point in time, we cannot report on quality improvements directly related to the continuous application of automatic assessments, MAN's engineers are convinced of its benefits and plan to further broaden the range of criteria that is analyzed in an automatic fashion.

6.1.5 Discussion

At the beginning of the project, MAN was primarily interested in developing a new quality guideline for the development of Matlab Simulink models. For MAN's engineers, the quality modeling approach behind the generated guideline was initially not important as their interest was focused on the guideline that is handed to the project participants. However, during the course of the project we were able to convince MAN of the benefits provided by the modeling approach. Consequently, MAN decided to not only use the generated guidelines but also integrated the modeling approach itself, along with the QMM.editor, into their development process. We see this as a strong indicator for the applicability of our quality modeling approach in an industrial setting. This is emphasized by the fact that the MAN department we worked with is not an R&D department but a serial development department that applies our approach for the actual development of software to be released in their vehicles.

6.2 Web User Interface Frameworks (Interasco GmbH)

This case study describes the application of the maintainability modeling approach to compare the expected maintenance efforts of web applications developed with different user interface frameworks. The case study was carried out with the Interasco GmbH as a diploma thesis [113]. The results of this thesis have been used by Interasco as input for a major reengineering of one of their core products.

6.2.1 Environment

Interasco GmbH is a software and consulting company located in Munich. One of their core products is INTERASCO|bpmf, a framework for managing enterprise structures and business processes. The system supports the central management of projects, resources, documents, appointments and invoices. Additionally, it provides workflow support and an interface to other enterprise systems. Due to diverse and frequent customer requests of mainly perfective nature, the system undergoes continuous evolution. Importantly, a significant share of the change requests does not concern the application logic but the web user interface of the product.

6.2.2 Goals

The web user interface of INTERASCO|bpmf was initially realized with the *Active Server Pages (ASP)* framework³ that is no longer maintained by Microsoft as it has been superseded by ASP.NET⁴. Hence, Interasco planned to replace the outdated UI technology with a more recent one in a major reengineering of INTERASCO|bpmf. Due to the high number of UI-related change requests the key requirement for the new UI technology was high maintainability, or more precisely, low maintenance effort.

Unfortunately, there is a bewildering variety of more than 100 different UI frameworks available that almost all promise low maintenance efforts. Hence, Interasco required a structured method to evaluate different UI frameworks with respect to the expected maintenance efforts in the specific context of INTERASCO|bpmf.

6.2.3 Study Description

In the study, we developed a quality model that describes properties of UI technologies and used the model to evaluate 11 different technologies. Based on the evaluation, we chose the technology most suitable for a reengineering of the INTERASCO|bpmf system and proposed a modified architecture for INTERASCO|bpmf on the basis of the chosen technology.

³<http://msdn.microsoft.com/en-us/library/aa286483.aspx>

⁴<http://www.asp.net/>

Quality Model Design To support the structured evaluation of UI technology, we developed a QMM-based quality model. The model was not developed from scratch but borrowed the activities and a number of facts from a quality model we had developed earlier in a project in the field of telecommunications. This initial model was then extended to suit the evaluation of UI technologies. Hence, it does not contain facts that describe the maintainability of an actual system but rather facts related to the idiosyncrasies of UI technologies. The development of the model was guided by the following sources:

- Interviews that were conducted with Interasco developers to identify issues that hamper maintenance of the ASP-based implementation of INTERASCO|bpmf.
- Numerous non-academic sources like developer web sites, blogs and magazines⁵.
- Most importantly, the graduand took six weeks time to develop a full understanding of the INTERASCO|bpmf system by reading its source code. Over this time, all phenomena that complicated his understanding of the system were carefully reviewed and, if appropriate, included in the quality model. Our assumption was that issues that complicate his understanding of the system are prone to hamper its maintenance in general.

In total, this process led to the addition of 4 activities, 15 entities, 19 facts and 25 impacts to the initial model. The activities cover UI-specific tasks like UI design, implementation and modification. The new entities and facts are used to describe properties of UI technologies. The model is tailored specifically to the context of INTERASCO|bpmf. For example, it includes model elements dedicated to the capabilities of an UI technology to interface the INTERASCO|bpmf system's core that is written in C++ and C#. While this limits the generality of the model and complicates its reuse in a different context, we believe this specialization is necessary to ensure that the model faithfully describes the suitability of UI technologies for INTERASCO|bpmf. Examples of properties covered by the model are:

- *Language Mix.* Many UI frameworks for web applications require mixing multiple languages, e. g. HTML, Java and JavaScript, within one source file. This has been identified as being problematic because program comprehension is severely complicated. Additionally, editing such source files is inconvenient as advanced editor features like syntax highlighting, code formatting and auto-completion often do not work if multiple languages are mixed. In the model this is expressed as [Language Mix | EXISTENCE] $\xrightarrow{-}$ [Program comprehension] and [Language Mix | EXISTENCE] $\xrightarrow{-}$ [Implementation].
- *Variable Declaration.* The script languages of many UI frameworks allow the usage of variables without explicit declaration. Such implicit variable declarations have been identified to complicate program comprehension and debugging. Hence, the impacts [Var. Decl. | IMPLICITNESS] $\xrightarrow{-}$ [Program comprehension] and [Var. Decl. | IMPLICITNESS] $\xrightarrow{-}$ [Debugging] have been included in the model.
- *Data Types.* Similarly, script languages often support variables with loosely defined types. Such variables also make program comprehension and debugging more difficult. Moreover, they complicate the integration of the UI with the application's core that is not written in a loosely

⁵With the exception of [194], the academic community made few contributions that explicitly deal with factors influencing the maintainability of web applications at the time of the study.

typed language as explicit conversions are required. This is expressed by [Data Type | IMPLICITNESS] $\xrightarrow{-}$ [UI Implementation].

- *Tool Support.* The available tool support for a UI framework is a key factor as compilers, debuggers, editors and other tools significantly reduce the maintenance effort. This is, for example, expressed by [Debugger | EXISTENCE] $\xrightarrow{+}$ [Debugging].

Similar to the previous case study, we used the impact set $I = \{-, +\}$ (see Sec. 4.2.3 for details) as the central piece of information for each fact was if it has a positive or negative influence on an activity. However, the remainder of this section will show, that we used a more elaborate scale for the actual evaluation of the different UI technologies.

Evaluation The resulting quality model was then used to evaluate 11 different UI frameworks. These were categorized as:

- *Classic Web Frameworks* use a server-side script language to generate HTML and JavaScript. Candidates in the study were ASP, *Java Server Pages (JSP)*⁶ and PHP⁷.
- *Modern Web Frameworks* usually follow an object-oriented concept and allow to define user interfaces on an abstraction level above HTML. Candidates in the study were ASP.NET and Java Server Faces (JSF)⁸.
- *Web-UI Generators* take abstraction one step further and allow to implement web user interfaces in the same way frameworks like Swing⁹ or SWT¹⁰ are used to implement rich client user interfaces. Hence, developers do not have to deal with script languages to implement web user interfaces. Candidates in the study were the Google Web Toolkit (GWT)¹¹ and the Eclipse Rich Ajax Platform (RAP)¹².
- *Sandbox Applications* are essentially »normal« applications that are run within the browser using a virtual machine. Candidates in this study were Java Applets¹³, .NET User Controls¹⁴, XML Browser Applications (XBAP)¹⁵ and Flex¹⁶.

For each of the 11 UI technologies the 19 facts were evaluated. Each fact $f \in F$ was assigned a value $v_f \in \{0, \dots, 6\}$ that expresses how accurately the fact describes a specific UI technology. We found that this relatively elaborate ordinal scale is required to express differences between different UI technologies. For example JSP has a value of $v_f = 0$ for the fact [Variable Declaration | IMPLICITNESS] as JSP requires variables to be declared explicitly. ASP, on the other hand, is assigned a value of $v_f = 3$ as it allows the developer to globally configure whether variables must be declared or not.

⁶<http://java.sun.com/products/jsp/>

⁷<http://www.php.net/>

⁸<http://java.sun.com/javaee/javaserverfaces/>

⁹<http://java.sun.com/javase/technologies/desktop/>

¹⁰<http://www.eclipse.org/swt/>

¹¹<http://code.google.com/intl/de/webtoolkit/>

¹²<http://www.eclipse.org/rap/>

¹³<http://java.sun.com/applets/>

¹⁴<http://msdn.microsoft.com/en-us/library/y6wb1a0e.aspx>

¹⁵<http://msdn.microsoft.com/en-us/library/ms754130.aspx>

¹⁶<http://www.adobe.com/products/flex/>

Lastly, for PHP the value is $v_f = 5$ as it does not provide explicit declarations of variables (the value is not $v_f = 6$ as implicit variable declarations can at least be identified in a log file). If possible, these values were determined based on the documentation of the respective UI technology. If the documentation did not provide sufficient insight, prototypes were developed to evaluate the technology and the values were then derived from the experience made with the prototype. Though time-consuming, this was an important step to evaluate how well the technology could be integrated with the INTERASCO|bpmf system. As the model itself, the values were chosen to specifically match the context of INTERASCO|bpmf. Therefore, they are not directly transferable to the evaluation of UI technologies in the context of other systems.

Based on these values and the impact function \mathcal{I} , we defined the rating function \mathcal{R}_F that combines the direction of the impact and the assigned value. It is defined as:

$$\begin{aligned} \text{(Fact Rating)} \quad \mathcal{R}_F : (F \times T) &\rightsquigarrow \{0, \dots, 6\} \\ \mathcal{R}_F(f, t) &= \begin{cases} v_f & \text{if } \mathcal{I}(f, t) = + \\ v_f - 6 & \text{if } \mathcal{I}(f, t) = - \\ \text{undefined} & \text{if } (f, t) \notin \text{dom.}\mathcal{I} \end{cases} \end{aligned}$$

Hence, if the impact $\mathcal{I}(f, t)$ of fact f on an activity t is positive and v_f has a high value, the activity is supported well. If the impact $\mathcal{I}(f, t)$ is negative and the value v_f is high, the activity is supported poorly. To be able to compare different UI technologies, we defined function \mathcal{R}_T that expresses how well an UI technology supports a specific activity $t \in T$:

$$\begin{aligned} \text{(Activity Rating)} \quad \mathcal{R}_T : T &\rightarrow \mathbb{N}_0 \\ \mathcal{R}_T(t) &= \sum_{f \in F_t} \mathcal{R}_F(f, t) \end{aligned}$$

where $F_t = \{f \in F \mid \mathcal{I}(f, t) \in \text{dom.}\mathcal{I}\}$ is the set of facts that have an impact on t . Note, that this sum considers all impacts to be equally *important*. While a more sophisticated scheme using weights could be applied, we found the unweighted sum to be adequate for our use case. Based on the value of \mathcal{R}_T for all activities $t \in T$ we were able to compare the different UI technologies with each other. For this comparison, we chose to visualize the performance of each UI technology with radar charts as shown in Fig. 6.3. This proved to be useful as the visualization highly aggregates the information but still allows to identify strengths and weaknesses of each UI technology. We also experimented with aggregations that condensed all activity ratings to a single number but found this to be counterintuitive as the single number was inadequate to express the idiosyncrasies of the different technologies.

In the visualization, all leaf activities $\{t \in T \mid \mathcal{A}^{-1}(t) = \emptyset\}$ represent the axes of the chart. The position on an axis for activity $t \in T$ is defined by $\frac{\mathcal{R}_T(t)}{6 \cdot |F_t|}$ that describes how close the rating is to the

optimally achievable rating. Hence, the bigger the area of the radar chart, the better an UI technology supports the key activities and the lower are the expected maintenance costs¹⁷.

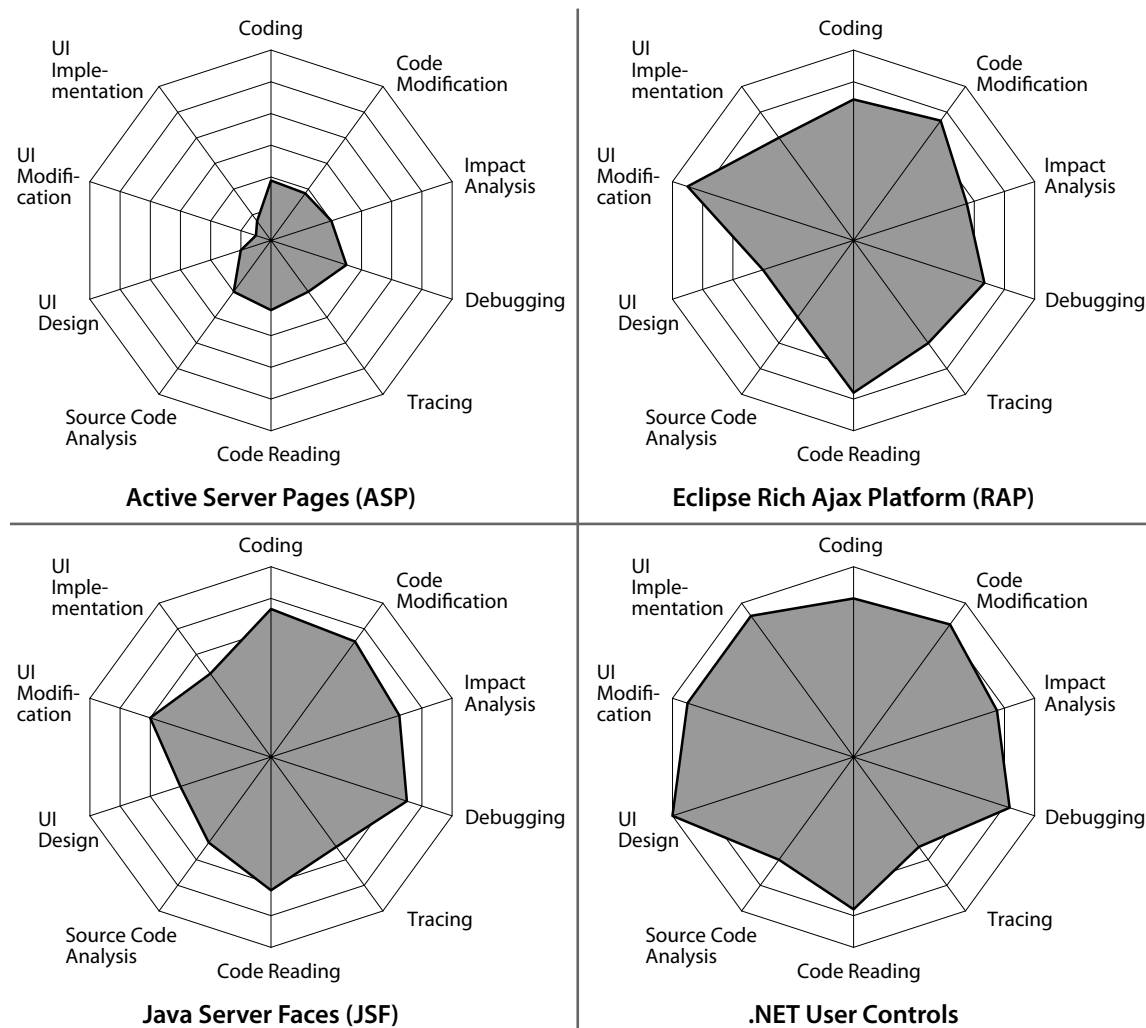


Figure 6.3: UI Technology Radar Plots

6.2.4 Results

We do not report in detail on the results obtained for the different UI technologies as they are not relevant in the context of this thesis. However, we explain how the four example radar chart in Fig. 6.3 can be interpreted: The ASP technology is rated relatively poor for almost all activities as its script language exhibits a number of inherent problems. The JSF technology rates higher for most activities as it raises the abstraction level and supports developers in many UI-specific activities. The Eclipse

¹⁷In the chart in Fig. 6.3, all UI technologies reached a relative value of at least 0.4 for all activities. To emphasize differences between the technologies, the charts, hence, show only the scale from 0.4–1.

RAP technology ranks high for the activity *UI Modification* as the user interface is implemented in plain Java without the need for any additional languages. Modifications are, hence, supported by automated *refactorings* provided by the Eclipse IDE. RAP ranks rather low for the activity *UI Design* as the GUI editor for RAP was relatively immature at the time the study was conducted. The .NET User Controls rank equally high for the *UI Modification* as they, too, do not require additional languages. However, they rank higher for *UI Design* due to their excellent GUI editor and for *UI Implementation* as they can be easily integrated with INTERASCO|bpmf's core that is also (partly) written in a .NET language.

The study's central result was that for the INTERASCO|bpmf-specific context, .NET User controls promised to cause the lowest maintenance effort. Consequently, Interasco has chosen this technology as basis for the reengineering of the INTERASCO|bpmf system. As part of his work, the graduant proposed a modified architecture for INTERASCO|bpmf on the basis of .NET User Controls. At the time of writing this thesis, the implementation of this proposal is well under way and, up to now, the expectations regarding the reduced maintenance effort of the chosen technology have been fulfilled.

With respect to the quality modeling approach, the study led to the insight that creating a quality model that is used to assess a technology is comparatively easier than building a model for the assessment of an actual system. The reasons for this are twofold: First, when the conformance of a system to a quality model is assessed, one has to deal with the problem that each fact needs to be assessed for multiple artifacts. For example, a fact that describes a certain property of methods needs to be assessed for all methods of a system. The results of these individual assessments need to be aggregated in a suitable way (see Sec. 4.2.2 for details). This challenge does not arise for the quality model in this study as it describes properties of *UI technologies* instead of actual *software systems*. Second, the model developed in this study is used only once for this specific study and is not intended to be integrated into a development process. This circumstance allows a certain degree of laxness in the model design. For example, the descriptions of the model elements do not have to be linguistically polished as they are not included in generated quality guidelines or checklists.

6.2.5 Discussion

In this study, the QMM-based quality model proved to be well-suited for structuring the multitude of factors that are expected to influence the effort required to maintain a web user interface. Using a QMM-based quality model we were able to select a suitable UI technology from a bewildering array of different options in a structured manner. Interasco developers and managers emphasized that the clear focus on expected maintenance efforts was preferred to other comparison schemes that merely list features of different technologies without making explicit the consequences for software maintenance. By putting the focus on *efforts* the modeling approach ensures that it is not limited to a specific type of factors, e. g. language features. On the contrary, all factors that affect software maintenance, e. g. available tool-support or circulation of a technology, could be included. Furthermore, the QMM modeling approach allowed to combine a top-down and bottom-up way of building the quality model. The top-down design was aided by the activities tree of the model as developers could break down maintenance of the UI into subactivities and then reason about the facts that influence these activities. The bottom-up design was supported by including facts about the UI technologies,

e. g. the implicit variable declarations mentioned above, and reason about the effects they have on the maintenance activities.

As the reengineering of INTERASCO|bpmf is still under way, we do not have made enough long-term experience to judge if the chosen technology really exhibits the expected low maintenance efforts. However, we consider it a success for the modeling approach that Interasco relied on the results of this study to make the crucial decision on the UI technology used in future versions of one of their core products.

6.3 Mainframe Development Infrastructure (BMW)

This case study focuses on the quantitative evaluation of the economic benefit created by isolated development and test environments for mainframe software development. The study is an example for an in-depth evaluation of the impact that a single fact has on the maintenance efforts. The case study was carried out with the BMW Group. It has partly been published in [80] and [81].

6.3.1 Environment

The object of investigation of our study was the maintenance and test processes used by the BMW Group's mainframe software development division. At BMW, several hundred software engineers develop and maintain critical business information systems with a total of 85 millions lines of PL/I and COBOL code. The division uses two separated IBM zSeries mainframes for development and operation, whereas our study focused exclusively on the development mainframe.

Mainframe Software Development Unlike the more common workstation-based development environments, mainframes in general do not provide developers with isolated environments where they can edit, compile, link and test the code they are working on without interfering with other projects or developers. In fact, if no additional measures are taken, all developers share the same development environment and all test data. Due to the frequent separation of development and operation spaces of a typical mainframe installation, this does not pose any problems for the operation of the software, but creates severe problems for the concurrent development and test of multiple projects. Conflicts between projects can occur during almost all activities (e. g. compile, link, test) and affect almost all development artifacts [228] (e. g. source code, libraries, test data). These conflicts are not only frustrating and time-consuming for the developers, but make sound testing almost impossible as test results cannot be interpreted properly. For example, if a test case fails, it is not decidable whether it failed because of a bug or because another project changed the test data in the shared data base. Unfortunately, isolated test spaces cannot be established for mainframes as easily as in ordinary workstation-based environments where every developer can have his own test space on his own workstation.

The CAP Isolation Mechanism The BMW Group developed a software-based isolation technique on top of the virtualization mechanism provided by the mainframe¹⁸. This technique offers projects isolated test and development environments called CAPs (*capsules*). These CAPs contain a complete copy of the required development environment including compilers, linkers, job control, and test databases. They thereby enable projects to develop and test in an independent, conflict-free manner until they reach a certain degree of maturity and can be integrated into the main development trunk in a special integration test phase. CAPs have the additional advantage of making it easy to *reset* the complete development environment of a project to a specific state. These advantages, however, come at a price, as the initialization, operation and support of a CAP is a non-trivial task that demands significant hardware resources as well as expenses for dedicated personnel.

¹⁸IBM zSeries mainframes provide a coarse-grained virtualization mechanism.

6.3.2 Goals

The qualitative benefit of a CAP can be explained quite easily by pointing out how non-isolated development environments create expensive conflicts and contribute to poor product quality due to unreliable test results. It is, however, very hard to compare these qualitative benefits to the known quantitative costs of the CAP mechanism. Therefore the research question of the study we conducted was: *What is the economic benefit of using a CAP for a software project?*

6.3.3 Study Description

To answer this question, we chose an activity-based approach and used a QMM-based model to describe the situation. In this model, the application of CAPs is expressed with the model's sole fact $f = [\text{Cap} \mid \text{EXISTENCE}]$. Based on this, we transformed the research question formulated above to match the activity-based approach and asked: *What is the impact of f on the activity Maintenance?* Our goal required to answer the question not only in a qualitative manner but demanded a quantitative answer with respect to the economic benefits. To achieve this, we defined the impact function of the quality model to express the savings that can be achieved by the application of the CAP mechanism:

$$\mathcal{I} : (F \times T) \rightsquigarrow I$$

$$\mathcal{I}(f, t) = \frac{e_{\text{Non-CAP}}(t) - e_{\text{CAP}}(t)}{e_{\text{Non-CAP}}(t)}$$

where $e_{\text{CAP}}(t)$ is the average effort spent on activity t when CAPs are used and $e_{\text{Non-CAP}}(t)$ is the effort spent on the same activity when CAPs are not used. Since the model contains only a single fact, the impact function does not depend on facts. As we assume that efforts are never negative, the impact set is defined as $I =]-\infty, 1]$ where a positive value means that the application of CAPs reduces the effort, a negative value means that efforts are increased. The impact function \mathcal{I} is partial in the sense that it is only defined for activities t with $e_{\text{Non-CAP}}(t) > 0$.

As it is hard to directly determine the values of $e_{\text{CAP}}(t)$ and $e_{\text{Non-CAP}}(t)$ for the activity $t = \text{Maintenance}$, our approach aimed at determining the efforts for the more tangible subactivities of the Maintenance activity. The overall effort is then determined by aggregating the efforts using summation. Hence, we included relevant activities of the BMW maintenance process, e. g. Design, Implementation or Unit Test in the model and set out to determine their efforts. For simplicity's sake all activities are modeled as a direct subactivity of activity Maintenance, e. g. $\mathcal{A}(\text{Unit Test}) = \text{Maintenance}$. We defined the effort for the *atomic* activities $t \in T' = \{t \in T \mid \mathcal{A}^{-1}(t) = \emptyset\}$ to be $e'_{\text{CAP}}(t)$ if CAPs are applied and $e'_{\text{Non-CAP}}(t)$ if CAPs are not used. Hence, e_{CAP} is defined as ($e_{\text{Non-CAP}}$ is defined in analogy):¹⁹

$$e_{\text{CAP}}(t) = \begin{cases} \sum_{l \in \mathcal{A}^{-1}(t)} e_{\text{CAP}}(l) & \text{if } \mathcal{A}^{-1}(t) \neq \emptyset \\ e'_{\text{CAP}}(t) & \text{otherwise} \end{cases}$$

¹⁹While function e_{CAP} is noted in recursive fashion it is not really applied recursively since all leaf activities of the model are directly attached to the single root activity.

A Probabilistic Process Analysis Model To determine the functions e'_{CAP} and $e'_{Non-CAP}$, we chose an analytical model that abstracts from the problem under investigation and allows us to focus on the impact of CAPs on development efforts. This model was inspired by an observation of the analogy between software development processes and concurrent systems theory [9]: The development *activities* like implementation or testing are comparable to the *tasks* run by an operating system save the fact that they are carried out by developers and not by CPUs. The *resources* competed for are not memory and file handles but source code, libraries and test data. Similar to the conflict that arises from a concurrent write access to the same memory address in a parallel system, a concurrent change to a program by two different projects produces a conflict in the software development process.

These considerations lead to a probabilistic process model that describes a development process as a system of concurrently executing tasks. The tasks of the system are the *activities* of the software process and the processors are humans (developers) executing these activities. Due to the goal of the overall process and limited resources, there are constraints on the order of the activities entailing the need for coordination. The transitions from one activity to possible subsequent activities are labeled with probabilities. Through this, there may also be *loops* describing costly rework in the development process due to failure or incompleteness at a certain stage of the process. The activities and the frequency of their execution define the cost and the duration of the project.

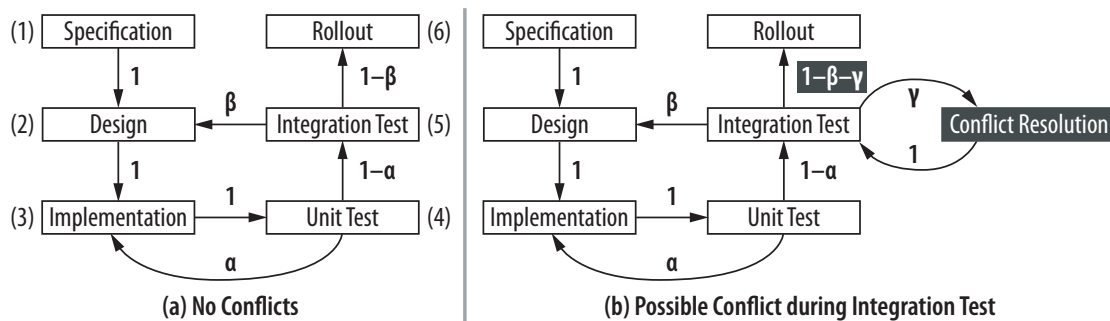


Figure 6.4: Example Processes

Fig. 6.4a shows a model of a simplified software process with the typical activities and transitions between them. Note that this model explicitly describes the loops (cycles) realistically found in software projects. This enables us to e. g. model the alternation between the activities Implementation and Unit Test that takes place in practice: Developers write some code, test it and then go back to implementing more code and/or fix existing code. They do so until they are eventually done with the implementation and all their tests pass. Note that the sum of the probabilities of the outgoing transitions of an activity must always be one. Fig. 6.4b illustrates how resource conflicts during specific activities can be expressed through adding conflict-specific activities and adjusting the transition probabilities accordingly. For example, a conflict with another project during the Integration Test does not only reduce the probability that the project can proceed with the activity Rollout but requires the execution of the additional activity Conflict Resolution.

Operationalization of the Model While this model provides an interesting abstraction of a software development process, it does not yet answer the question stated above. Fortunately stochastics

can help here as the process model can be viewed as a stochastic process or, more precisely, as a *discrete absorbing Markov chain* [127]. Each state of the Markov chain conforms to an activity in the maintenance process. The state transition probabilities conform to the probability to move from one activity, e. g. Implementation, to another, e. g. Unit Test. The last activity, e. g. Rollout conforms to an *absorbing state* that has exactly one outgoing transition to itself with the probability 1.

Absorbing Markov chains are a powerful tool for analyzing processes as they provide well defined methods to determine the expected total number of steps until the chain reaches an absorbing state as well as to calculate the expected number of steps spent in each state. Without going into the mathematical details we illustrate this for the process shown in Fig. 6.4a. For the exemplary probabilities $\alpha = 0.95$ and $\beta = 0.2$ the absorbing Markov chain analysis yields the following expected number of visits to each state (if started in state Specification): Specification is expected to be carried out only once, Design and Integration Test are expected to be performed 1.25 times, and Implementation and Unit Test 25 times. The total number of steps before the chain reaches the absorbing state Rollout is given by the sum, that is 53.5. Figure 6.5 shows how different values for the probabilities α and β influence the expected total number of steps in the example process. While values close to 1 lead to an infinite number of steps in both cases, one can see that increasing β raises the number of steps stronger than increasing α as this transition occurs *later* in the process.

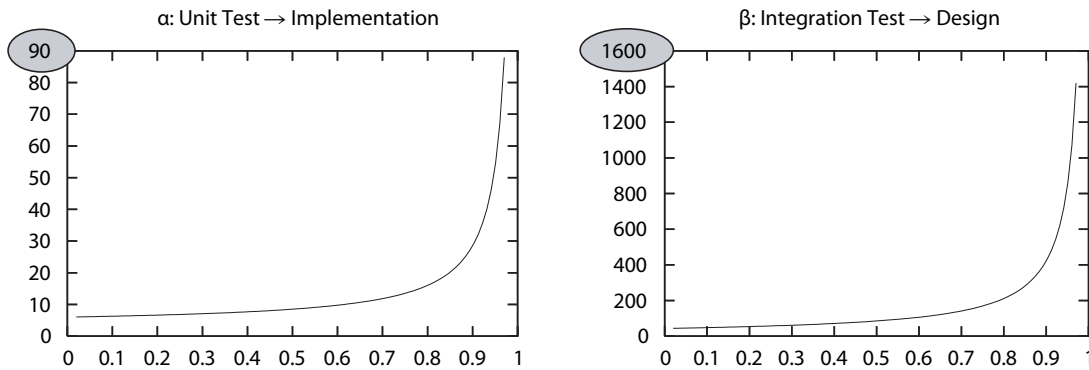


Figure 6.5: Transition Probability vs Total Number of Steps

The expected total number of steps represents a measure for project progress, but it does not yet answer the questions about the total project effort. To achieve this, each atomic process activity $t \in T'$ is now associated with the average effort $e^1(t)$ needed for a single execution of the activity. Importantly, we assume that this effort is not influenced by the application of CAPS. Rather, the frequency of the activity execution is different if CAPS are used or not. Hence, we define

$$e'_{CAP}(t) = e^1(t) \cdot s_{CAP}(t) \quad \text{and} \quad e'_{Non-CAP}(t) = e^1(t) \cdot s_{Non-CAP}(t)$$

where $s_{CAP}(t)$ describes the expected number of executions of activity t if CAPS are applied and $s_{Non-CAP}(t)$ describes the number of executions if no CAPS are used. The next sections explain how these functions are determined by building Markov chain-based process models for the maintenance process that applies CAPS and the same process without CAPS.

Application of the Analysis Model to Isolated Testing To apply our approach to analyze the economic benefit of isolated test and development at BMW, two fundamental pieces of information are needed:

1. transition probabilities
2. average execution $e^1(t)$ effort for each activity $t \in T'$

As it is not realistic to *correctly* determine this information without investing a large amount of effort for empirical studies, we analyzed the two process variations (CAP and Non-CAP) in a *relative* manner. We therefore designed a *reference process*, calibrated it with existing empirical data and parameterized it with the probability for conflicts during development and test. Based on this reference process we designed the process models for CAP and Non-CAP development and compared them using the method presented above. This comparative approach allowed us to abstract from concrete values for the transition probabilities as well as the efforts spent for each activity.

Based on existing process descriptions and interviews with project managers as well as developers, we created the reference process model with 13 activities and 18 transitions (not presented here in its entirety due to confidentiality reasons). This model does not contain special isolation-related activities and therefore consists of the usual specification, design, implementation and test activities. It does, however, carefully distinguish between module tests and two levels of integration tests and contains explicit error analysis activities. Eleven of the 18 transitions of the model have a transition probability unequal one. Using existing process analysis data as well as interviews, we estimated the probabilities and ensured that the remaining impreciseness does not bias our study results (the discussion in Sec. 6.3.4 illustrates that the choice of the transition probabilities is not as crucial as one might expect).

To determine the effort needed for each execution of the activities, we calibrated the reference process with data from well-known empirical studies like [37, 162]. For example, the Markov chain analysis showed that the activity Implementation will be carried out 95.24 times and thereby accounts for 36.78% of the expected total 258.95 process steps. As [37] and other sources point out that implementation usually accounts for $\approx 20\%$ of the total development effort, we concluded that the relative effort $e^1(\text{Implementation})$ of each execution of Implementation activity in our process is 0.21%. These relative measures of effort were later on used to compare the different processes.

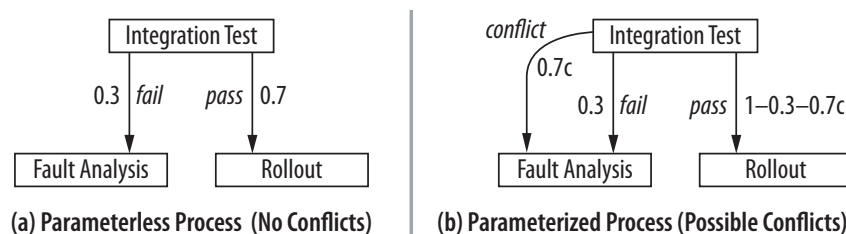


Figure 6.6: Process Parameterization

Obviously, the difference between the CAP and Non-CAP development processes is determined by the number of conflicts with other projects that arise during the different activities. We expressed this by introducing the *conflict probability parameter* c and parameterized the process models accordingly.

Figure 6.6 exemplifies this for the Integration Test and shows how the conflict parameter c influences the transition probabilities. Please note that the CAP process, though isolated, is not fully free of conflicts as conflicts may arise during the Integration Test when the project leaves its CAP.

Based on the previously defined reference process we built specific models for CAP and Non-CAP development. The models differ as the CAP model contains specific CAP-related activities, e. g. CAP Refresh and the Non-CAP model explicitly describes conflict resolution activities (Fig. 6.7). In the figure, the nodes that belong to the reference process are shown as white boxes and the reference process' transitions as solid lines. Gray boxes and dotted lines are extensions that were introduced to model the specifics of the CAP and Non-CAP processes. The CAP and Non-CAP process models were then used to determine the functions $s_{CAP}(t)$ and $s_{Non-CAP}(t)$ that describe the average number of activity executions for all activities $t \in T'$. As pointed out before, this can be done rather easily by applying the standard analysis techniques for absorbing Markov chains.

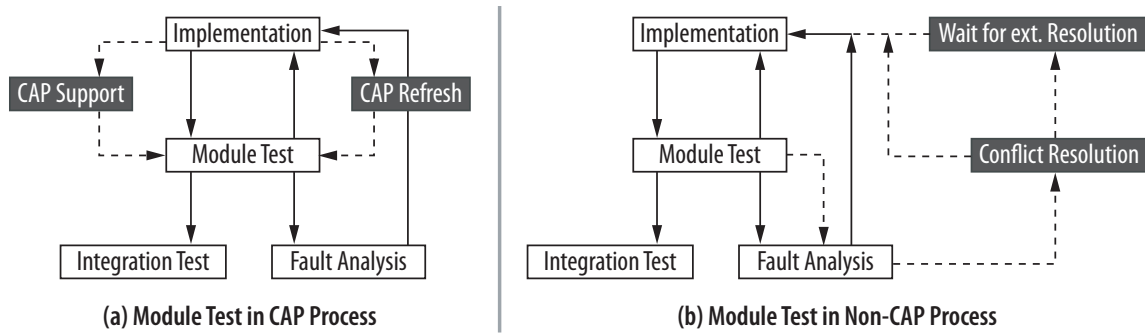


Figure 6.7: Differences between CAP and Non-CAP Process (Module Test)

For both processes the Markov chain analysis was performed. The total effort ($e_{CAP}(\text{Maintenance})$) respectively $e_{Non-CAP}(\text{Maintenance})$) was put into relation with the same calculation for the reference process. Table 6.1 exemplifies this for the example processes shown in Fig. 6.7. It shows the relative effort for each execution of each atomic activity $e^1(t)$ in the second column. The other columns show the expected number of executions for each activity if CAPs are used or not. Additionally, the total efforts $e'_{CAP}(t)$ and $e'_{Non-CAP}(t)$ for the atomic activities are shown. The bottom row shows the aggregation for the Maintenance activity. The total efforts for this activity if CAPs are used is 1.06, i. e. 6% more expensive than the reference process. If CAPs are not used, they are 1.55. Hence, the expected savings caused by the CAP mechanism for this example are:

$$\mathcal{I}(f, \text{Maintenance}) = \frac{e_{Non-CAP}(\text{Maintenance}) - e_{CAP}(\text{Maintenance})}{e_{Non-CAP}(\text{Maintenance})} = \frac{1.55 - 1.06}{1.55} = 0.32$$

6.3.4 Results

The same approach was used to analyze the process models for CAP and Non-CAP development. However, for this analysis also varying conflict probabilities have been taken into account. Figure 6.8

Activity t	$e^1(t)$	$s_{\text{Cap}}(t)$	$e'_{\text{Cap}}(t)$	$s_{\text{Non-Cap}}(t)$	$e'_{\text{Non-Cap}}(t)$
Implementation	0.08	3	0.25	4.3	0.34
Module Test	0.08	3	0.25	4.3	0.34
Integration Test	0.25	1	0.25	1	0.25
Fault Analysis	0.25	1	0.25	1.9	0.48
CAP Support	0.1	0.3	0.03	0	0
CAP Refresh	0.1	0.3	0.03	0	0
Conflict Resolution	0.1	0	0	0.9	0.09
Waiting for ext. Resolution	0.1	0	0	0.5	0.05
Σ		8.6	1.06	12.9	1.55

Table 6.1: Expected Efforts and Steps for the Example Process (Fig. 6.7)

shows the results in two resolutions. On the left, the total effort for all three processes is shown for the conflict parameter interval $[0; 0.6]$. One can easily see that the efforts for the reference and CAP process behave in a similar way whereas the effort for the Non-CAP process increases much stronger. However, the right side with its finer resolution (interval $[0; 0.2]$) shows that for very low conflict probabilities the effort for the CAP process exceeds the effort for the Non-CAP process.

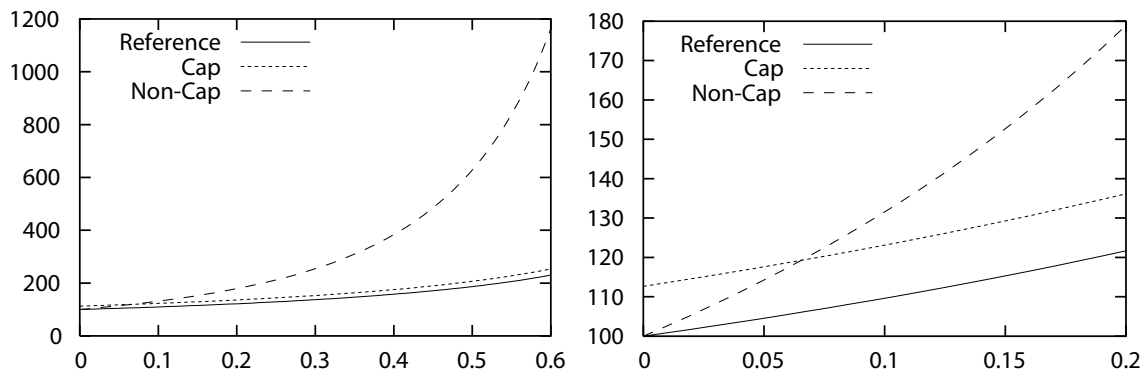


Figure 6.8: Conflict Probability vs Relative Effort

The results can be explained by analyzing the frequencies of each activity in the three process models. In the CAP and reference process an increasing conflict probability raises only the frequency of the integration test that is performed when the project leaves the CAP. In the Non-CAP process, however, the conflict probability also affects the module test. As the test activities constitute *nested loops* in the process this leads to a much stronger increase of the overall effort. It is also obvious that the CAP process has higher costs than the Non-CAP process for very small conflict probabilities as the cost for creating and maintaining the CAPs occurs independently of the conflict probability. This meets the expectation that CAPs are obsolete if there are no conflicts.

Estimation of the Conflict Probability As the results of the process analyses show, the final decision on the economic efficiency of the CAP mechanism depends on the conflict probability parameter c . To determine the conflict parameter, we analyzed the average number of dependencies among

mainframe programs and examined the number of actual changes of these programs by using the configuration management system. The latter is important as program-to-program dependencies do cause conflicts only if both programs are modified at the same time. For the analyzed period of one year we found that 55, 86 *relevant* (i. e. with possible conflict) changes occur for every program every year. Given a work year of 200 days, this resolves to 0.279 relevant changes a day. As the reference process predicts about 100 test activities per year, this finally leads to a conflict probability of $0.279/2 = 0.1395$ or 14%. Note that this does only regard static program dependencies but *not* data dependencies. For data, the conflict probability is expected to be even higher.

The process analysis and the estimation of the conflict probability yield that $\mathcal{I}(\text{Cap, EXISTENCE, Maintenance}) = 0.2$. Hence, we concluded:

Projects with an average number of dependencies save about 20% of total effort through using the CAP isolation mechanism as they avoid additional process cycles and conflict resolution activities.

We therefore recommended to use non-isolated development only for projects with no or very few dependencies. Although we do not have a formal external validation of our results, we can say that our quantitative results fully support our project partners' qualitative experiences. In addition to this, our recommendation was already followed before this study was conducted, as project managers intuitively chose isolated development only for projects with zero or few dependencies.

Threats to Validity The major threats to the validity of these results is the determination of the transition probabilities and the memoryless nature of Markov chains. To evaluate how strongly different transition probabilities influence the results, we performed a sensitivity analysis [257] to determine the transition that has the highest influence on the result. Using the variance-based *Extended FAST Method* [256] we found the transition Module Test \rightarrow Integration Test to be not only the most *important* but with an *total order index* of 0.72 about three times as important as the second ranked transition. We therefore focused our analysis on the most important transition probability and found that changes to this probability do of course change the absolute efforts calculated for each process model. They do, however, *not* change the relation between CAP and Non-CAP development processes.

The memorylessness of Markov chains implies that the transition probability from e. g. Module Test to Implementation Test and others is always the same, no matter how often the activities have been carried out before. As this might contradict one's intuition, we evaluated the influence of memorylessness by introducing a *process memory* in form of a compound interest function for the activity efforts. By defining a negative interest rate (*reduction rate*) we could simulate a situation where each execution of an activity demands less effort than the previous execution. Repeating the analysis for the two process models with this process memory showed again that the memory does influence the absolute results but *not* invalidate the relation between the CAP and Non-CAP development processes.

6.3.5 Discussion

Due to the simplicity of the applied quality model, the study did not create new insights on the creation of quality models itself. However, it helped to significantly advance our approach with respect to a quantitative evaluation of the impact a specific fact has on common maintenance activities. As expected, the study illustrated that a quantitative evaluation of impacts is considerably more challenging than a qualitative solution. In particular, it showed that a quantitative evaluation requires the explicit modeling of activity interdependencies. For example, we found that it is not possible to determine the impact of a fact on the activity Unit Test without taking into account that another activity, e. g. Integration Test, is also affected by the fact and, hence, might cause repeated executions of the Unit Test.

Nevertheless, the study showed that the activity-based approach is well-suited for breaking down complex questions regarding the economic benefits of a specific project characteristic into more tangible questions regarding specific activities. It shows that, given a suitable impact function plus the required aggregation mechanism, QMM-based quality models cannot only be used to define quality characteristics but also to perform a quantitative evaluation of maintenance efforts. The fact that the results of the study fully supported our project partners' qualitative experiences in this complex scenario, makes us confident, that a similar approach can be applied to answer quantitative questions regarding other quality factors described by QMM-based models.

6.4 Quality Dashboards (ABB & Munich Re)

This case study describes the application of ConQAT-based quality dashboards in industrial settings. To continuously control selected quality criteria of software products, ConQAT-based dashboards were used by ABB in Vaasa, Finland and by the Munich Re Group in Munich for four projects. In both cases, the quality dashboards not only helped to keep the current state of quality but also supported step-by-step quality improvements. Parts of this study have been published in [77].

6.4.1 Environment

The Munich Re Group is one of the largest re-insurance companies in the world and employs more than 37,000 people in over 50 locations. For their insurance business, they develop a variety of individual software systems supporting their business processes. Subject of this case study were 3 systems that are based on the .NET platform and are written in C#. They are each developed by different organizations and provide substantially different functionality, ranging from damage prediction, over pharmaceutical risk management to credit and company structure administration. The systems support between 10 and 150 expert users each. Their sizes range from 300 to 500 kLOC²⁰.

ABB is one of the world's leading power and automation engineering companies. It employs about 115,000 employees in more than 100 countries. ABB *Distribution Automation* is a subdivision of ABB *Power Products* and develops products for the protection, automation and monitoring of electrical networks. Subject of this case study was a desktop application written in C# that is used by ABB customers to configure the hardware products. Due to ABB's worldwide operations, the maintenance is carried out at locations in Vaasa, Finland and Bangalore, India. The system has about 500 kLOC. In the study the quality control infrastructure was installed for the whole system. However, only the developers of a selected subsystem were made familiar with the quality control approach.

6.4.2 Goals

For both companies, the goal of using a software quality dashboard was to increase the transparency regarding the quality of the maintained software product. To achieve this, project managers as well as developers sought a solution that enables them to monitor key quality criteria in a continuous manner. In ABB's case this requirement was emphasized by the distributed development environment that makes controlling product quality particularly challenging. As the dashboards were to be introduced for the maintenance of relevant systems, minimal disturbance of the maintenance processes and operations of the systems were a key requirement of both companies. Consequently, the focus was on quality criteria that can be assessed in an automatic manner.

6.4.3 Study Description

Both companies did not plan to significantly change their practice of quality management but rather wanted to extend the existing practice by the application of a quality dashboard. Hence, we did not

²⁰thousand lines of code

apply a QMM-based quality model but selected specific quality criteria and supported their control by a ConQAT-based quality dashboard.

Criteria Selection Both companies planned to start with a small set of criteria to gain experiences without investing too much efforts. Furthermore, the initial set of criteria should be kept small to not overwhelm users of the dashboard. The process of selecting these criteria consisted of three steps:

1. In workshops as well as interviews with different project participants (developers, project managers, architects) we elicited quality criteria that are relevant for the projects. While this elicitation process did not always follow a structured approach, it was designed in a top-down manner, i. e. starting from the core activities of the respective maintenance process, we tried to elicitate quality criteria that pose problems for these activities.
2. From the criteria identified in the elicitation process, we selected the ones that can be analyzed in an automatic manner with reasonable effort. We then configured ConQAT to analyze these criteria and ran it on the system under investigation. Crucially, we also analyzed criteria that were not discussed or dismissed during the elicitation to later present these results to the project participants, too. These criteria were selected on the basis of our experiences with similar projects.
3. In the next step, the results from the first analyses were discussed with the project participants in workshops. In these workshops, we presented the results of the criteria that the project participants considered important along with the results for the criteria we additionally analyzed. From this extended set of criteria, a subset of criteria was chosen that promised to be most beneficial for the respective project. The decision on the criteria took into account the following factors: (1) the impact the criterion has on the maintenance process, (2) the current state of quality of the system with respect to the criterion, (3) the efforts required for creating or configuring the analysis (if not yet provided by ConQAT), and (4) the expected quality of the analysis results with respect to the rate of false positives.

In the case of ABB, we chose to analyze source code redundancy (*code clones*), unused code and internationalization (i18n) issues. Code clones were selected as they are harmful for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs [183, 251] and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [167]. The identification of unused code was chosen as such code unnecessarily bloats the systems and complicates program comprehension. The identification of i18n issues, e. g. hard coded error messages in English, were chosen as they are crucial for ABB that sells its products in several countries. The clone detection was carried out with our own clone detection tool CloneDetective²¹ that is based on ConQAT [166]. The other analyses were performed with FxCop²², a static analysis tool for .NET languages. In the case of unused code, the analysis was limited to types of unused code that can be detected easily and produces low rates of false positives. Examples are unused local variables or methods with visibility *private* that are never called.

²¹<http://www.clonedetective.org/>

²²[http://msdn.microsoft.com/en-us/library/bb429476\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx)

In the case of the Munich Re, we also chose to analyze source code redundancy due to the reasons outlined above. In addition, we focused on an *architecture conformance analysis* that compares the intended architecture with the actual architecture implemented in the system. Today's object-oriented programming languages typically do not provide dedicated structuring mechanisms beyond the class level. Hence, architectural rules, e. g. regarding componentization or layering, are not made explicit in the system. Consequently, developers can (intentionally or unintentionally) violate these rules and thereby create a system that violates the original architectural concept. Such violations can significantly increase maintenance efforts. For example, an architectural rule can state that a certain component may only be accessed via a dedicated interface as this allows to change the internals of the component without affecting other components. If this rule is, however, violated and other components directly access the internals of the component, changes to its internals become considerably more costly.

Setup After the final decision on the criteria to be analyzed, ConQAT was configured to match the requirements for each project. For the unused code and i18n issue analysis, this mainly consisted of selecting the relevant FxCop rules and creating an appropriate configuration for FxCop. For the clone detection, most efforts were devoted to reducing the number of false positives generated by CloneDetective; e. g. by excluding generated code and stereotype code patterns like the *getter* and *setter* methods typically found in object-oriented systems.

The architecture analysis obviously requires a description of the intended architecture. The initial architecture descriptions were developed together with the system architects based on the existing architecture documentation. Usually, this initial description does not fully match the architecture of the system. However, this mismatch is not only caused by architecture violations but also by the outdatedness of the architecture documentation. Hence, each violation reported by the architecture analysis was discussed with system architects to decide if it really constitutes a violation or if the architecture description needs to be updated accordingly. This process was supported by a graph visualization generated by ConQAT that shows allowed architectural dependencies as well as architecture violations (Fig. 6.9). Details on the architectural analysis can be found in [103].

On average, about one person day was required for the initial setup and the tailoring of the clone detection in each project. In the case of ABB, the configuration of FxCop required half a person day. In the Munich Re projects, about two days were spent on the creation of the architecture description for each project.

Infrastructure Integration To continuously control the selected quality criteria, ConQAT and other required tools like FxCop were integrated into the nightly build process of the respective systems. As Fig. 6.10 shows, the source code is checked out from the version management system, compiled and analyzed with ConQAT and other tools every night. The results are published via a web server to make them available for all project participants. For this, different visualizations were used. The results of the architecture analysis were visualized with the graphs shown in Fig. 6.9. Additionally, a simple list of violations depicting the offending classes is included in the report to support developers in removing violations. Clone detection results are also reported as a list that contains all clones including their location in the source code. As this list is usually rather long, the results are also visualized with tree maps (cf. Fig. 5.8) that give an overview of the distribution of clones over

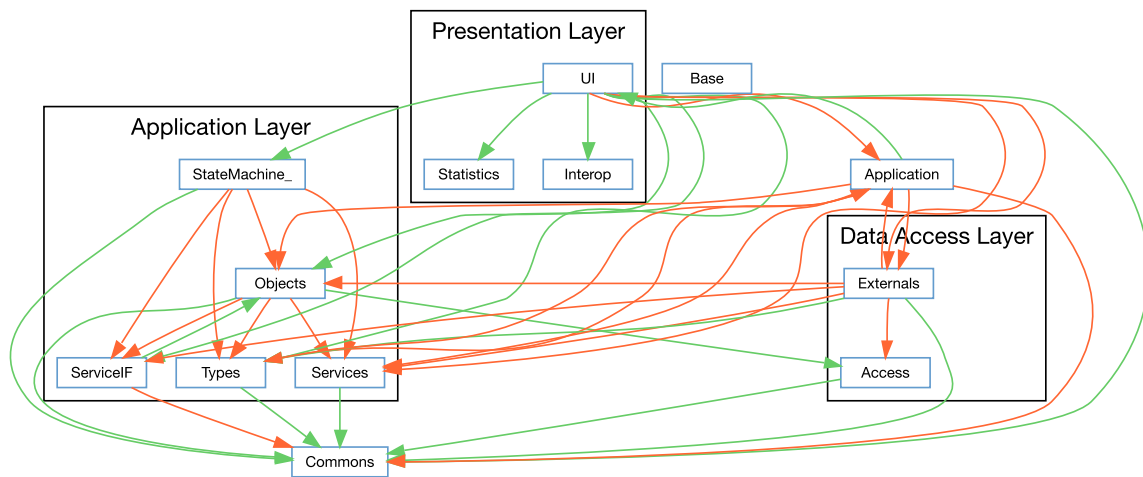


Figure 6.9: Architecture Analysis Graph

the system. Furthermore, the web server provides a clone detection result file that can be loaded with the standalone clone analysis application that comes with CloneDetective. This supports developers in deeply investigating clones to understand causes and consequences. Analysis results regarding unused code and i18n issues were also visualized with tree maps and as lists. In addition, relevant metrics like *clone coverage*²³ and *number of architecture violations* were stored in a database to trace their development over time. These developments were visualized with line charts. Integrating CON-QAT into the nightly build process of the projects required one person day on average.

Process Integration While the technical integration of the analyses into the nightly build is necessary, it is not sufficient for continuously controlling the selected quality criteria. An integration into the organization's or project's development process is required to ensure that the results of analyses are considered and necessary action is taken.

Successfully integrating the quality control measures into an existing process is challenging as it depends not only on technical but also on organizational and sociological factors. Examples are the maturity of the organization with respect to quality assurance processes in general and the dedication as well as the standing of the person that advocates the new quality control measures within a team. As these factors were not under our influence, we confined ourselves to giving suggestions regarding the process integration but left the decisions largely to the organizations. If desired, we provided trainings for all concerned project participants to teach them in interpreting the analysis results. As we could not provide trainings to all developers in ABB's case due to its multi-site operations, we wrote a handbook that explains the core functionality of the quality dashboard.

Due to the reasons explained above, we encountered very different ways of integrating the quality control measures into the existing processes for the different projects we analyzed. In ABB's case, it is left mainly to the individual developer to consult the analysis reports and react accordingly.

²³The clone coverage is the ratio between the size of the part of the system that is part of at least one clone and the entire system size. It depicts the probability that an arbitrary system statement is cloned.

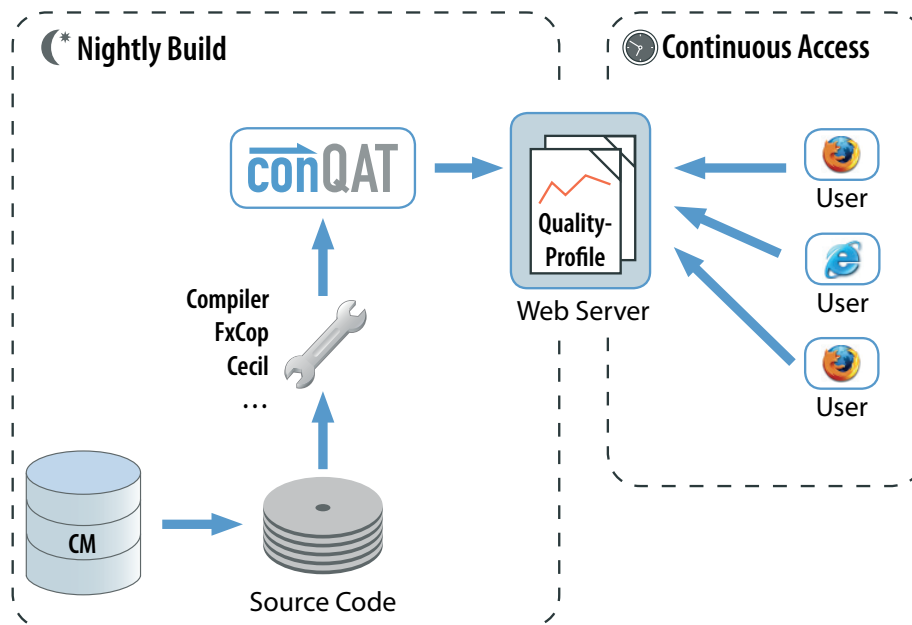


Figure 6.10: Nightly Build Integration

Additionally, developers are strongly encouraged to report relevant findings and developments in their regular developer meetings. These meetings include the project manager who can decide on appropriate measures if the decision cannot be taken by the developer himself.

Also in the case of Munich Re, regular developer meetings are the central means for discussing the quality analysis results and deriving adequate measures. However, for specific quality criteria, the Munich Re aims at designating a single person responsible for monitoring the criterion and reacting accordingly. For example, in one project a developer took care of the architecture violations. After removing all of them in the current systems, he now follows a *zero violation policy* and ensures that new violations are dealt with immediately.

6.4.4 Results

The initial setup of the quality control infrastructure was carried out at ABB in May 2008, at the Munich Re between April and June 2008. While it is still in use at both companies, the following results are based on data collected until August 2008 (ABB) respectively December 2008 (Munich Re).

Munich Re At the Munich Re we found that the application of continuous quality control not only helped to maintain the status quo but, in fact, led to stepwise quality improvements. Figure 6.11 shows the development of the *clone coverage* metric over the analyzed period of time for two of the three projects. In the upper project, the clone coverage was reduced by 10%, in the lower by 25%. In the third project (not shown) no significant reduction was achieved. As there are currently no good quantitative estimates for the effect of clones on maintenance efforts, it is hard to precisely capture

the benefit of the reduction in clones. However, in another study carried out with Munich Re, we found that clones do not only increase maintenance efforts but are also fault-prone [167]. Hence, we consider a reduction of the clone coverage by 10% respectively 25% a significant improvement in product quality.

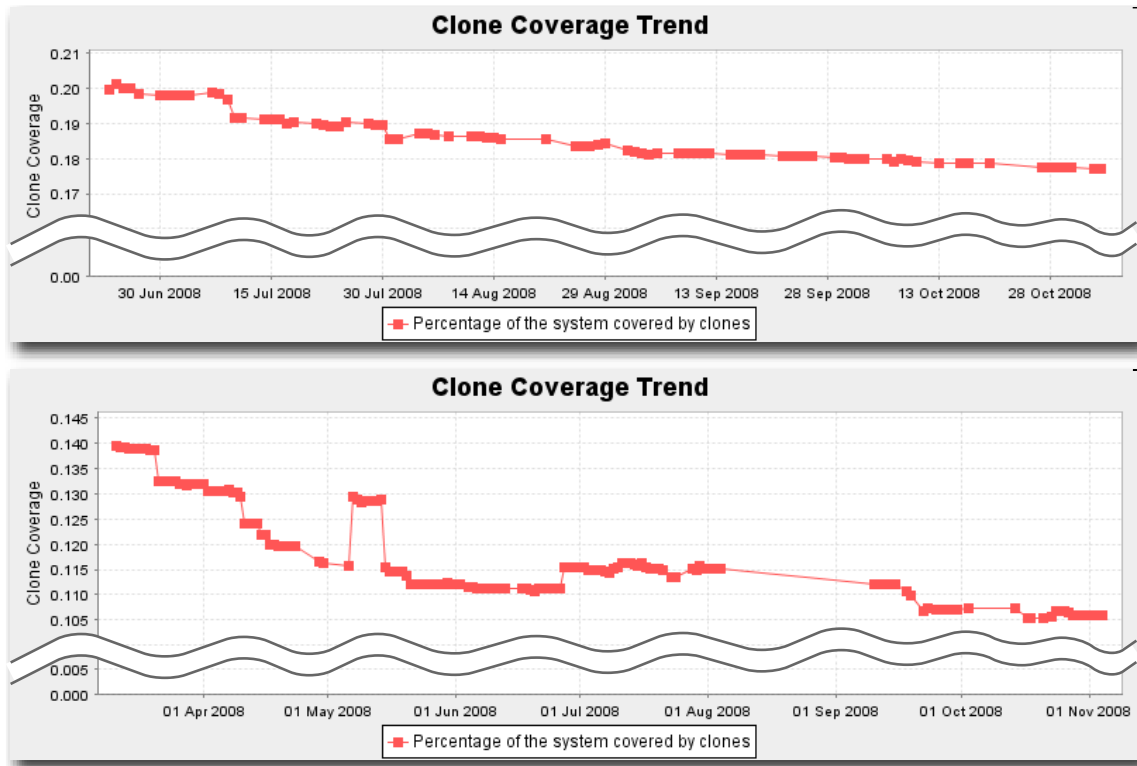


Figure 6.11: Clone Coverage Trends (Munich Re)

With respect to the architecture analysis, we found that the creation of the architecture description already was a valuable step in itself. The newly created architecture description not only represents an accurate and up-to-date documentation for the architecture but is also well-suited to be kept up to date due to the possibility to automatically compare it with the implemented architecture of the system. We also found that the different projects applied different strategies for dealing with the identified architecture violations. While one project removed all violations and further on adopted the *zero violation* policy, other projects only dealt with the most crucial violations immediately. To enable these projects to differentiate between known but tolerated violations and newly introduced ones, we extended the architecture analysis mechanism to explicitly support violation toleration. Such tolerated violations are visualized as yellow edges in the dependency graphs to make them distinguishable from newly introduced violations. In summary, the architecture violation analysis did not help to remove all violations in all projects but, nevertheless, allows to identify new violations as soon as they are introduced.

ABB While the status quo regarding the i18n issues was maintained at ABB over the course of the three month study period, no significant improvement was achieved. This is illustrated by the trend curves shown in Fig. 6.15. The temporary rise of the number of issues was caused by a temporary increase in system size. The trends, however, are very different for the number of warnings regarding unused code (Fig. 6.13²⁴) and for clone-related metrics (Fig. 6.14²⁵): With respect to warnings about unused code, a reduction of 61% was achieved and code cloning was reduced by about 42%. Interviews with developers revealed that this mainly was the result of thorough clean-up activities that took place in the first half of August.

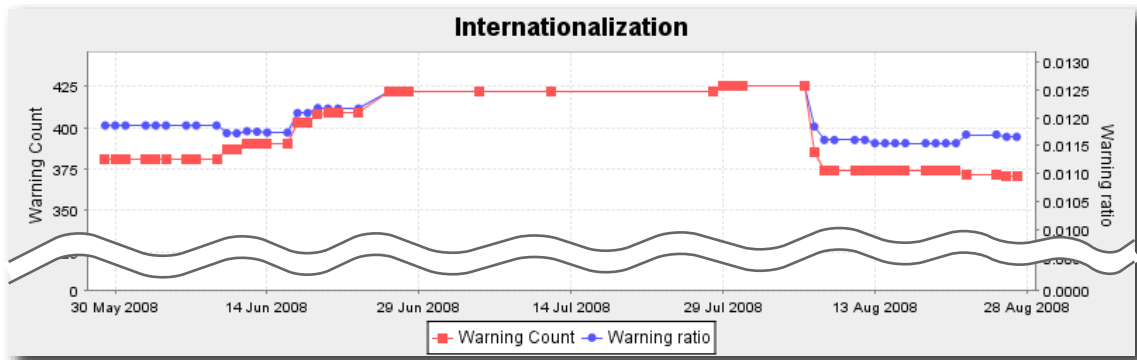


Figure 6.12: I18N Issues Trend for Selected Subsystem (ABB)

However, it needs to be kept in mind that these trends capture only the selected subsystem for which developers were made familiar with the quality control approach. Over the same period of time, the trends for the whole system show no significant improvements although the status quo was maintained. This is illustrated by the clone trends for the system shown in Fig. 6.15.

General In general, we found that for all projects the application of a quality dashboard efficiently and effectively supported continuous quality control. Our partners at ABB and Munich Re considered the installed dashboards as highly beneficial and concluded that they enable them to control the selected quality criteria with relatively few additional efforts. For most of the selected quality criteria, the applied quality control approach did not only help to maintain the status quo but, in fact, led to stepwise quality improvements. An insight we gained with this study was that the trend curves do not only support the monitoring of developments over time but are also perceived very motivating by developers. Particularly, when developers carried out dedicated clean-up activities, the trend curves were considered helpful as they provided timely feedback on quality improvements.

During the study we encountered a number of challenges that need to be addressed to make continuous quality control successful:

- In order to apply continuous quality control, a certain degree of *awareness* for quality issues is required from all project participants. It is not enough that upper management decides on a

²⁴The figure shows the total number of warnings found and the number of warnings per line of code.

²⁵Next to the clone coverage, the figure shows the total number of clones and the number of cloned source code statements (units).

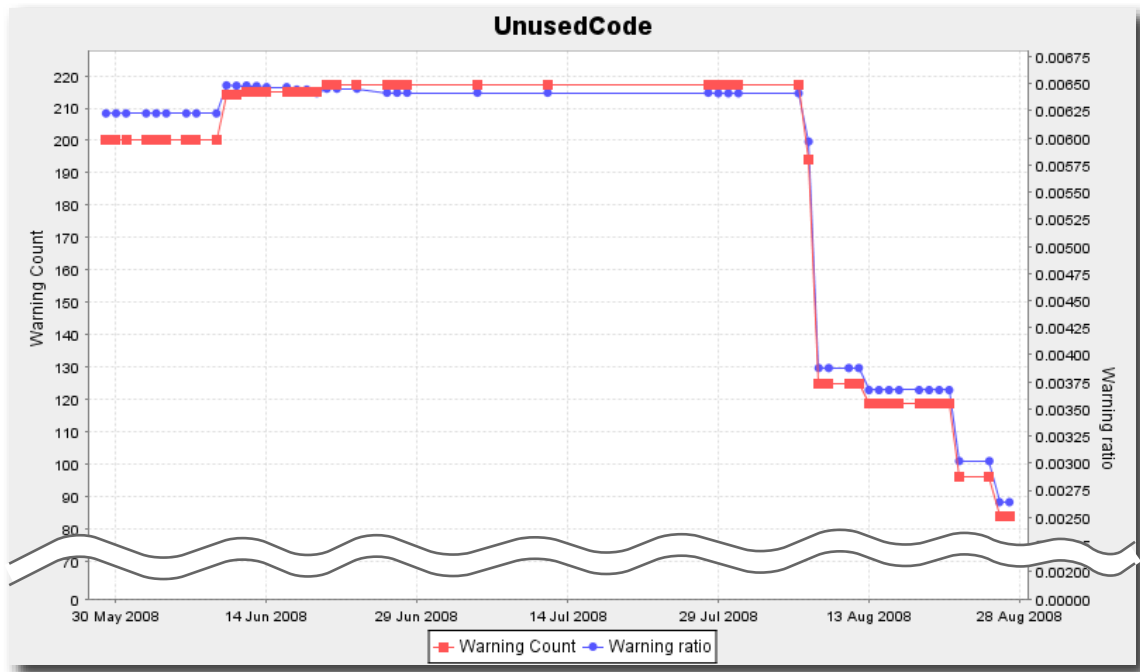


Figure 6.13: Unused Code Trend for Selected Subsystem (ABB)

»quality improvement program« if project managers and developers either do not see a need for it or do not understand the issues that are addressed. However, we also found that sound analysis results are a major factor in convincing project participants of the importance of quality control measures. In fact, striking results, e.g. a list of inconsistent clones that represent faults [167], was found to be a strong motivator for developers and upper management alike.

- Next to a sound technical integration of analysis tools, a firm integration of quality control in the maintenance process is required. As measuring alone does not yield quality improvements, roles and processes need to be defined that ensure that assessment results are monitored and necessary action is taken. Due to the differences in the development organizations and culture, we cannot give precise guidelines on how such processes must be designed at this point. In fact, we found that different approaches work with varying degrees of success in different environments. As pointed out before, the success of a certain strategy largely depends on organizational and individual factors.
- A fair amount of training is required for all project participants to make ideal use of the quality dashboards. This training must not only include material on the basic usage of the dashboards but must also ensure that all project participants have a good understanding of the relevance of the selected quality criteria. We found that quality improvements could only be achieved if the concerned persons did not only have a technical understanding of the selected quality criteria but also considered them important themselves. During our first attempts we also found that there is a high risk of overwhelming users of quality dashboards. Thus, we selected a relatively small set of quality criteria in this study and advice all dashboard endeavors to start small.

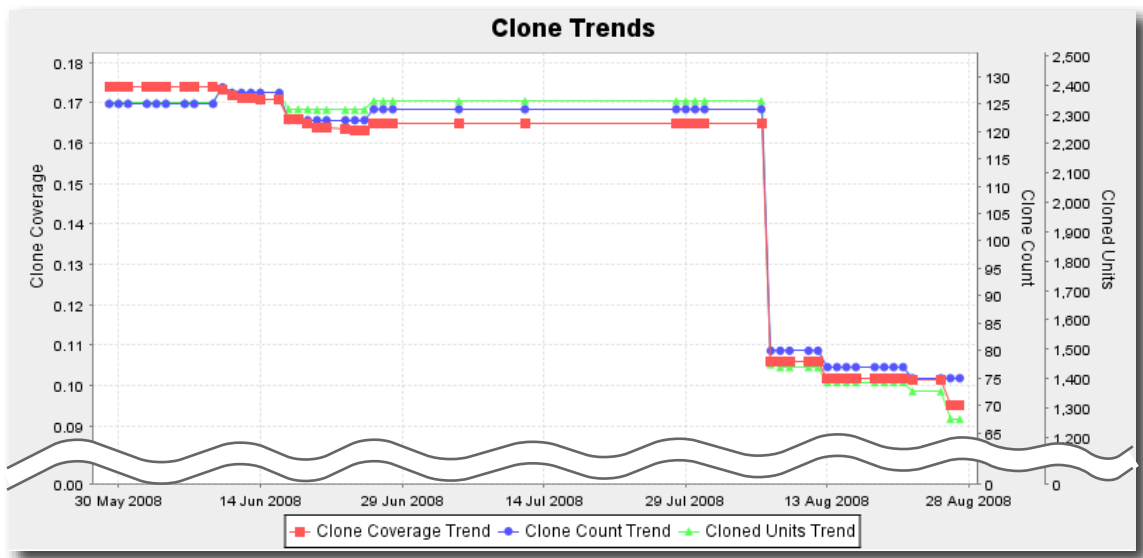


Figure 6.14: Cloning Trend for Selected Subsystem (ABB)

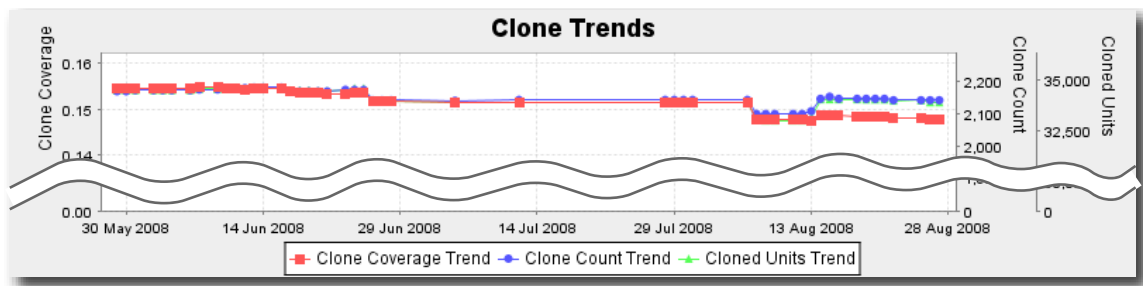


Figure 6.15: Cloning Trend for Whole System (ABB)

While these challenges are clearly of managerial nature, we also found that the application of quality dashboards in an industrial setting poses a number of technical challenges with respect to the analysis tools. One challenge is to present analysis results in a form that suits different project participants equally well. For this, we found that, in fact, different levels of aggregation and different types of visualization support different project participants differently well. Moreover, the suitability of the presentation also depends on the task a project participant carries out and is not strictly bound to his role. For example, graphs that visualize architecture violations were perceived as well suited for creating architecture descriptions. However, continuous monitoring of new violations was best supported by simple violation lists. Clone detection results were visualized with tree maps to indicate their distribution over the systems. While this worked well to provide an overview, developers usually required a more detailed view on individual clones. In this case we found that their requirements were well beyond anything a normal visualization could provide and, hence, equipped them with a specific clone inspection application. Our conclusion is that analysis results are best presented on different levels of aggregation using different visualizations. While this increases train-

ing efforts required to make users familiar with the different kind of reports, it helps to support all project participants in all their manifold tasks.

Another aspect relevant for continuous quality control is the toleration of known quality defects. In most cases, quality control is introduced to systems that are under maintenance for many years. Hence, most of them contain a number of quality defects that are first identified when quality control is applied. As projects rarely have time or resources to remedy these quality defects at once, quality dashboards must provide mechanisms to *tolerate* such problems. This is crucial to be able to distinguish known but accepted problems from newly arisen ones.

Analysis Accuracy Most importantly, we found that the application of analysis tools in a continuous manner raises different requirements regarding the accuracy of analysis results than the sporadic application of the same tools. An experienced user of analysis tools can easily cope with false positives when he carries out an analysis only sporadically. However, when analyses are applied continuously, e. g. during the nightly build, developers and other project participants need to deal with the false positives on a daily basis. We found that this is not only cost-intensive but simply infeasible as developers refuse to respect the analysis results if they are polluted with false positives. In fact, our estimate is that false positive rates above 2% or 3% are not tolerated by developers, i. e. a *precision*²⁶ greater than 97% is required for continuous application. Currently, many analysis techniques (and tools) do not provide such high levels of precision if run »out of the box«. The reasons for this are diverse:

- Sometimes it is not fully defined what accurate analysis results are. An example for this problem is clone detection. As it is not always clear which code fragments are actually clones, certain findings by the tools may be considered false positives by some project participants [297].
- Some analyses are inherently imprecise as they rely on heuristics that do not always satisfy the requirement of high precision. Again, clone detection is a prime example for this problem.
- In some cases, analysis results are imprecise as the analysis tools are not implemented properly. An example is the analysis for unused private fields provided by PMD. In certain situations, it reports fields to be unused although they are actually used. A special case of this problem are certain project idiosyncrasies that analysis tools were not prepared for. As this can be highly project-specific aspects, such problems cannot be called *bugs* of the analysis tools. The effect, however, is the same.
- Another problem is that analysis tools are sometimes applied in an inappropriate context. For example, the results generated for portions of code that are automatically generated and never changed are usually irrelevant as no manual maintenance is performed that could be affected by the findings.

Different methods to address these problems exist. Heuristic approaches usually provide some kind of parameterization that allows to influence the applied heuristics. In most cases, the parameters can be used to increase precision at the expense of a reduced recall. While this is a potentially dangerous

²⁶*Precision* and *recall* are terms originally used in the context of information retrieval to measure the performance of retrieval techniques [198]. In the context of quality analysis tools, *precision* can be interpreted as *the fraction of the generated results that is truly relevant* and *recall* can be interpreted as *the fraction of relevant results identified by the analysis tool*. Ideally, both should be equal one.

approach as important findings may be excluded too, it sometimes is the only feasible way to include an analysis in a continuous quality control approach. Fortunately, recall must not always be sacrificed for precision. In certain cases, the parameters allow to adapt the heuristics to the specifics of a project and thereby help to increase accuracy. However, some heuristic approaches are simply not (yet) suited for a continuous application. Examples are the detection of *inconsistent* clones to identify faults [167] or most approaches that involve natural language analysis, e. g. in the context of program identifiers or comments [243, 244].

An approach to deal with unreliable heuristics or improperly implemented analyses is to use multiple tools that carry out similar analysis and compare their results. Depending on the number of tools and the quality of their analysis results, different strategies can be applied to increase result accuracy. For example, findings can be reported only if more than one tool identified the same problem. Another strategy that can be used to deal with many forms of false positives involves the developers and is usually called *blacklisting*. Blacklisting allows developers to mark individual findings as false positives with the purpose of excluding them from future reports. While this approach is conceptually simple, it is technically challenging as the location of the findings stored in the blacklist must be robust against changes of the system. Furthermore, developers must be equipped with tools to create and maintain blacklists. We applied blacklisting for the clone detection results in all projects of this case study and found it to be a very effective and efficient technique to increase analysis accuracy. We currently extend ConQAT with a generic blacklisting mechanism for all kinds of findings.

The last problem, the application of analysis tools in inappropriate contexts, can usually be addressed with suitable filters. For example, in all case studies generated code was excluded from the clone detection. Again, this is technically more challenging than one would expect. The reason is that generated code is often intermingled with handwritten code within source files. In these cases, it is not possible to exclude whole files; rather an exclusion of specific regions within the files is required.

The provision of different visualizations and different aggregations of the same data, the capability to deal with tolerations and the implementation of approaches to deal with false positives require a high flexibility and customizability of the applied analysis tools and the quality dashboard. This case study generated convincing evidence that the pipes&filters architecture that was chosen for ConQAT is capable to provide the required flexibility and, hence, supports the efficient and effective creation of quality dashboards in realistic industrial settings.

6.4.5 Discussion

Both companies stated that they were not only convinced by the quality improvements that were achieved during this case study but perceive continuous quality control with ConQAT-based dashboards as beneficial for their projects in general. While many questions remain open, e. g. about the ideal process integration, we believe that this case study provides strong evidence for the usefulness of continuous quality control in industrial settings. This is emphasized by the fact that both companies extended their cooperation with our research group to widen the application of quality control. Moreover, a number of other companies now apply ConQAT in a similar fashion.

To fully implement the quality control approach advocated in this thesis, a QMM-based definition of quality criteria and the automatic generation of guidelines still remains to be done. As we will continue our work with both companies, we are confident that this can be achieved in the near future.

6.5 Integrating Manual and Automatic Quality Analysis

This case study describes the integration of manual and automatic quality assessment measures. We apply this combination for the development of ConQAT itself and use it to control product quality in student projects. This study specifically reports on the application of integrated assessment techniques in a student lab course that was dedicated to the development of ConQAT's graphical editor `cq.edit`.

6.5.1 Environment

The tool development group of the *Competence Center for Software Maintenance (CCSM)* at the Technische Universität München actively maintains about 200,000 lines of code that include ConQAT and related tools. The maintenance is performed by 5 permanently employed researchers. In addition, several students work on the code as part of lab courses, bachelor and master theses or as student assistants. Furthermore, a number of industrial partners contribute to some of the open source projects. Over a period of 4 years, 49 different developers contributed to code base. On average, there are about 10 active maintainers. Due to the relatively high fluctuation, the maintainability of the code is crucial. In many cases, student developers spend only a limited time, e. g. 6 month on a project, and, hence, cannot afford to waste time for becoming acquainted with a system that is hard to comprehend or extend. To control the maintainability of the CCSM code base, we apply the techniques and tools proposed in this thesis. In particular, we use ConQAT-based dashboards to execute automated assessments and monitor their results.

However, as repeatedly discussed in this thesis, there is a number of important quality criteria that cannot be assessed in an automatic manner but require manual reviews. Hence, the CCSM adopted manual reviews as its main means for assuring quality of the entire code base. As a fundamental policy, each source file has to be peer reviewed by another developer before it can be released. This rule applies to source code developed by researchers in the same way as it applies to students. In student lab courses, students cannot get credits for the course unless all the source code they developed was manually reviewed and accepted by one of the researchers. The manual reviews do not only help to improve quality but are also well-suited to teach students about software quality and reviewing techniques. So, the consistent application of manual reviews is viewed as beneficial by all project participants. Students, in particular, repeatedly articulated that they enjoy having their source code scrutinized by experts (the researchers).

While the techniques discussed in this case study are applied for all software maintenance carried out by the CCSM, for clarity's sake, this case study focuses on a single student lab course. This lab course was dedicated to the development of ConQAT's graphical editor `cq.edit` (see Sec. 5.2.7). As the developed code would later on be moved to the CCSM code base, stringent quality requirements were applied. During this 5 month course, 14 students developed the first version of the graphical editor. The editor is based on the Eclipse Rich Client Application Platform²⁷ (RCP) and comprised about 25,000 LOC at the end of the course. In preparation of the development of the editor, the students had several programming assignments to become acquainted with the RCP. In these assignments,

²⁷<http://www.eclipse.org/rcp>

about 50,000 LOC were developed that were all manual reviewed by the course advisors. However, only one round of review was performed for the code submitted for these assignments.

An important idiosyncrasy is the asynchronous and distributed type of development usually encountered in our lab courses. As both students and advisors have further tasks besides the lab course, they freely choose their development time as well as the location. Only for dedicated meetings, usually once a week, all project participants get together to discuss progress and next steps.

6.5.2 Goals

The goal of this study was to establish a deep integration of manual and automatic assessment techniques. In particular, the results of manual assessments should be integrated into a quality dashboard along with the results of automated assessments. This should enable the course advisors to keep track of the state of quality of all artifacts developed during the course. Moreover, a tight integration of manual and automatic assessments should help to reduce the efforts spent on manual reviews. The integration should enable reviewers to focus on difficult aspects that demand human involvement and leave simple things to the analysis tools. Moreover, the course advisors wanted to use automatic assessments as kind of quality gate before manual techniques are applied to ensure that artifacts that are manually reviewed exhibit a minimum level of quality.

6.5.3 Study Description

This section describes the approach that was used by the CCSM team to integrate manual and automated assessments in a quality dashboard. As the focus of this study is on the integration aspect, the quality criteria assessed manually and automatically are not discussed in detail. Examples of automatically assessed criteria were code cloning, unused code and the violation of low-level design rules in Java, e.g. the implementation of the *equals()*-method without an implementation of the *hashCode()* method. Examples for manually reviewed criteria were the appropriate choice of program identifiers, suitable comments and the correct application of standard data structures as well as algorithms.

Artifact Quality Life Cycle The central idea of the approach to integrate manual assessments with quality dashboards is the notion of the *artifact quality life cycle* shown in Fig 6.16. By default, a newly created artifact is rated RED. The author of an artifact can change its state to YELLOW to express that he is confident that all quality requirements are met. With this color change, the author signals that the artifact is ready to be reviewed. A reviewer, other than he authors, performs a quality review of the artifact and rates it GREEN if all quality requirements are met or RED if one ore more requirements are violated. This review process is supported by automatic analyses as described below. If the reviewer rated the artifacts RED, the author corrects the quality deficiencies and rates the artifact yellow, when he is finished. A GREEN artifact is automatically rated RED if it is subject to any modification. This way, it is ensured that all modifications are properly reviewed. Ideally, the artifact quality life cycle is tightly coupled with the software maintenance process as illustrated in the next section.

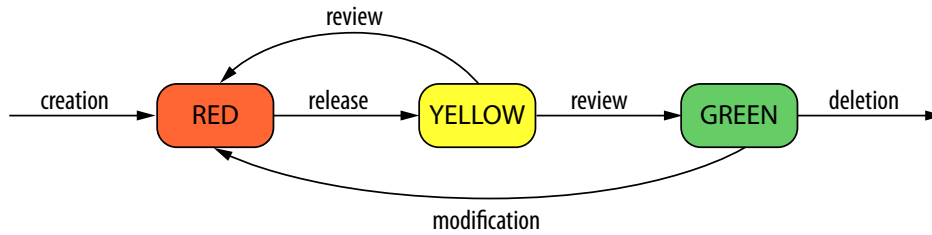
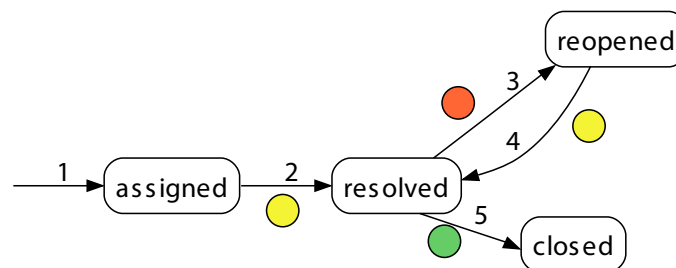


Figure 6.16: Artifact Quality Life Cycle

Process Integration In the lab course, we applied the simplistic software maintenance process depicted in Fig. 6.17. This process is change request driven and designed to be supported by the open source change management tool Bugzilla²⁸. We refer to this process as *Lean Software Evolution and Development (LEvD)* process. The table below explains the process transitions and the relation of the artifact quality life cycle to the maintenance process. In the lab course the assignments as well as the development of the editor were steered by this process. Concretely, the advisors created the change requests in the change management tool Bugzilla and assigned them to the students that implemented them. The students then used Bugzilla as their main mode of communication with the advisors and also with students of other teams. This enabled us to support the course’s asynchronous and distributed type of development.



Transition	Description	Rating
1 → ASS	Every team member can create a CR at any time.	–
2 ASS → RES	The CR is switched to <i>RESOLVED</i> after the work is completed.	All files affected by the CR must have rating <i>YELLOW</i> .
3 RES → REO	If QA decides that the CR is not completed properly, it reopens the CR.	At least one of the affected files must have rating <i>RED</i> .
4 REO → RES	The developers corrects the issues commented by the reviewers and switches the CR to <i>RESOLVED</i>	All files affected by the CR must have rating <i>YELLOW</i> .
5 RES → CLO	If the resolution satisfies all QA criteria, QA puts CR in state <i>CLOSED</i> .	QA must rate all files <i>GREEN</i> .

Figure 6.17: The LEvD Software Maintenance Process

²⁸<http://www.bugzilla.org>

Technical Realization To keep track of the artifacts' quality rating, the rating needs to be stored explicitly. This could either be done in a dedicated database or within the artifact itself. For simplicity's sake we chose the latter option and use a simple mechanism for storing an artifact's rating within comment constructs of the artifacts. An example for a rating stored in a Java source file is shown in Fig. 6.18a; in an HTML file in Fig. 6.18b: The token `@rating` is followed by the assigned rating color and the version the file had when the rating was assigned. The version information is required to identify files that have been modified after they were rated. To support this, the token `@version` is used to store the current version of the file. If this version is greater than the rating version, the file was modified after it was rated. In such situation it is by default assumed to be RED.

While the versioning could theoretically be handled manually, in practice this is not feasible. Hence, the support of a version management system like Subversion²⁹ or CVS³⁰ is required. Such systems usually provide a mechanism to replace a special token like `Rev` with the current version of the file when the file is checked out from the version management system. This ensures that the version information stored with the `@version` token is always up to date and allows to identify files that have been modified by comparing the rating version with the current version.

<pre> package edu.tum.cs.commons; import java.io.File; /** * This class ... * * @version \$Rev: 16749 \$ * @rating GREEN Rev: 16749 */ public class FileSystemUtils {...} </pre> <p style="text-align: center;">a)</p>	<pre> <!DOCTYPE html PUBLIC ...> <!-- @version \$Rev: 1643\$ @rating GREEN Rev: 1640 --> <html xmlns="... " > ... </html> </pre> <p style="text-align: center;">b)</p>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 6.18: Rating Tags

In a similar fashion, review comments are directly stored in the concerned artifact. A review comment is always prefixed with the tag `TODO` followed by the initials of the reviewer. The Eclipse development environment provides built-in support for such *task tags* and thereby supports the developer in locating the reviewer's comments. This is illustrated by the screenshot in Fig. 6.19: Eclipse does not only highlight the `TODO` tag in the code but also provides markers indicating such tags within a file (right) and list of tags that allows to navigate directly to the review comments (bottom).

IDE Integration While the rating information can be maintained manually for the artifacts, a minimal amount of tool support helps to increase efficiency. Hence, we developed a plugin for the Eclipse IDE that visualizes the rating of the artifacts and helps to set their rating. Both functionalities are shown in Fig. 6.20. Each artifact icon in the *Package Explorer* tree is decorated with a small colored circle on the top left. As the figure shows, rating colors are aggregated from bottom to top levels. Consequently, the package *suffixtree* as well as its super packages are rated RED since the package

²⁹<http://subversion.tigris.org/>

³⁰<http://www.nongnu.org/cvs/>

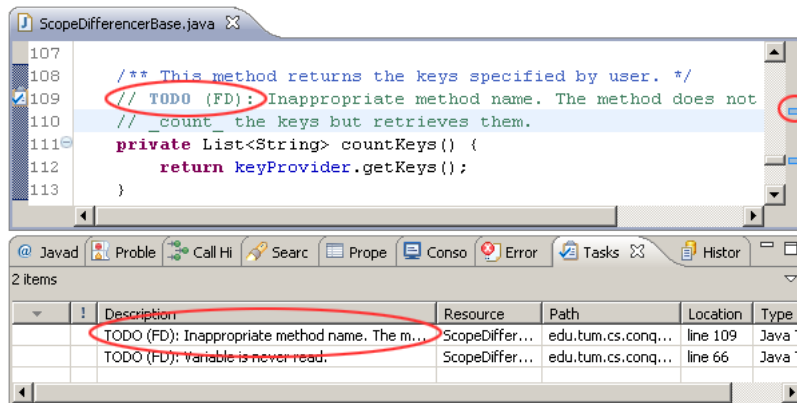


Figure 6.19: Review Comments in Source Code

contains a Java class with rating RED. Right-clicking an artifact allows to select the rating color via a context menu entry. Alternatively, a menu entry can be chosen that keeps the rating color of the artifact but updates the rating version. The same menu entries exist for packages to set the rating for all artifacts that belong to a package. The figure shows that these features do not only work for Java classes but also for artifacts like XML or HTML files.

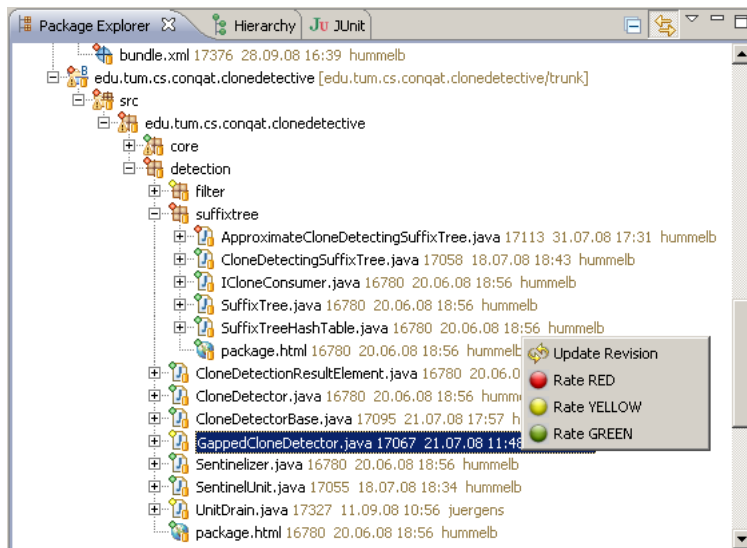


Figure 6.20: Eclipse Rating Plugin

Dashboard Integration The Eclipse plugin allows to set artifact ratings and shows the rating of artifacts that reside in a developer's workspace. However, this still provides no suitable means for the course advisors to control the quality of *all* artifacts. Hence, the rating information was integrated into ConQAT-based quality dashboards. During the course, the dashboard was updated hourly and,

therefore, allowed to continuously monitor the rating of all artifacts. Integrating the rating information into the dashboard was achieved with a ConQAT processor that analyzes the rating of files and annotates the files with the according traffic light color. Like this, the rating of a file and therewith the results of manual reviews can be integrated into a quality dashboard. Once the rating information is integrated, various aggregation and visualization techniques can be applied. We found the tree map representation shown in Fig. 6.21 to be the most important tool for controlling quality. The figure shows a snapshot of all Java classes of `cq.edit`. The size of each rectangle reflects the size of the class in LOC and the color represents the rating. One can easily see that the component on the left is mainly GREEN, i. e. reviewed and accepted. The components on the right exhibit a more patchy pattern with many yellow and red classes. Pointing the mouse brings up a tool-tip with additional information.

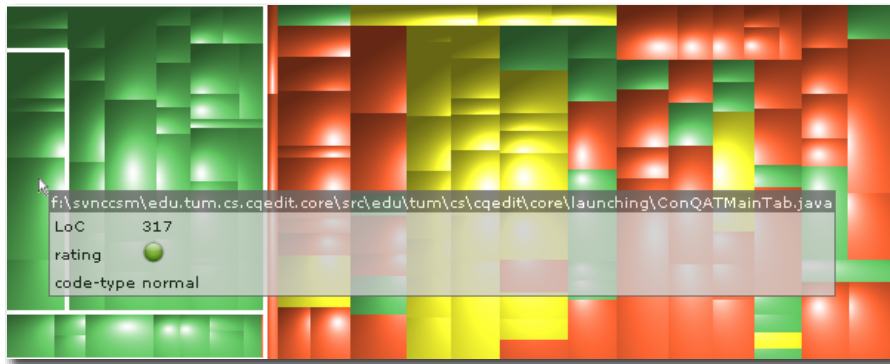


Figure 6.21: Rating Tree Map

A visualization that illustrates trends w.r.t. to the source code rating is the *stacked area chart* shown in Fig. 6.22. The chart displays the total size of the system (in LOC) and depicts the portions of source code that are rated GREEN, YELLOW and RED. This visualization is well-suited to monitor the amount of unreviewed code during phases of system growth.

Obviously, the dashboard can also be used to cross-check the rating information against the results of automatic assessments. For example, the dashboard can identify files that were rated GREEN or YELLOW but violated any of the automatically assessed quality criteria. Moreover, the results of the automated assessments presented by the dashboard can help to improve review efficiency. In the lab course, this was achieved in two ways:

- Before reviewing an artifact, the reviewer consulted the dashboard to check if any of the automated analysis tools reported a finding for the artifact to be reviewed. If a finding was reported, the reviewer simply rated the artifact RED without reviewing it. He then set the associated change request in the state *reopened* and added a suitable comment using the Bugzilla system. The developer of the artifact was alerted to this state change by e-mail and could correct the findings shown in the dashboard. During the lab course, the dashboards were updated hourly as part of an automated build process to ensure up-to-date information. This also enabled the developers of artifacts to check the findings of automated analysis before asking advisors for reviews.
- Obviously, this process works only for automated analyses that generate very low rates of false

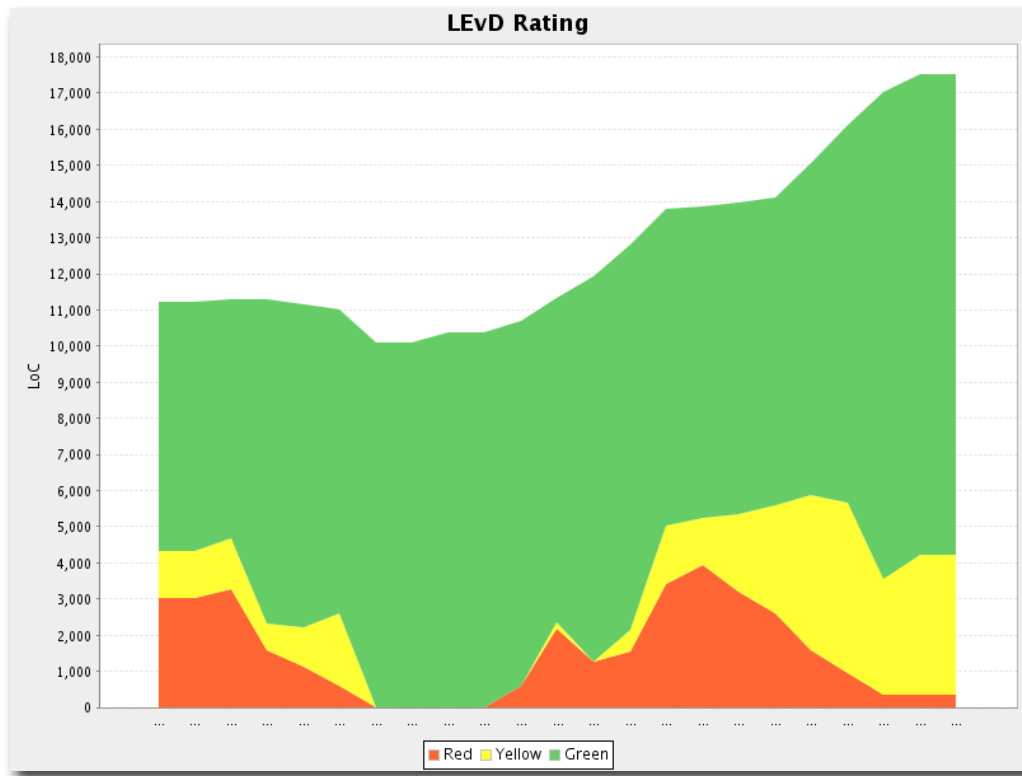


Figure 6.22: Rating Trend Curve

positives. For other analyses, e. g. the detection of redundant literals, the advisors used the analysis results by incorporating them into the reviews. Concretely, the reviewer started the review by checking such analysis results before he started the actual manual review. If the results were true positives, he added the respective review comments to the file using the aforementioned *TODO* tags. After this, he proceeded with his manual review.

6.5.4 Results

We found that the integration of manual and automatic quality assessments in one dashboard greatly helped to control the quality of the system artifacts. While the researchers involved in the lab course had advised several other lab courses before, this was the first time they were truly able to control the quality of the developed system. Using the dashboard they could quickly get an overview of the state of quality of the entire code base and react accordingly. For example, they stopped development of new features at multiple points and made student developers clean up existing code. Progress of such clean-up activities could again be monitored using the dashboard.

Furthermore, advisors concluded that the dashboard helped them to increase review efficiency as they could simply reject artifacts that did not fulfill minimum quality requirements. Moreover, advisors agreed that review effectiveness was increased, too. For example, they could conveniently

consult the dashboard to check if a source file under review shares duplicated code with other artifacts and annotated the file accordingly. Such a check would be impossible with manual reviews only. Even if appropriate tools were available but not integrated into a dashboard it would be very tedious as the analysis tool would need to be run for each reviewed artifact. Similarly, reviewers found that, possibly due to a lack of concentration, they often overlooked issues that the automated tools identified, e. g. unused fields of a class.

Nevertheless, the view of the author of this thesis regarding the importance of manual reviews was again confirmed in this lab course. All project participants agreed that many relevant review findings concerned things that are impossible to analyze automatically. Common examples are identifiers, comments and the inappropriate usage of data structures and algorithms. A great number of findings was related to students inexperience with the humongous Eclipse RCP framework. So, students often implemented functionality that, unbeknownst to them, was already provided by the framework. Importantly, these problems could not have been detected by current clone detection tools as they are limited to finding code that has actually been copied and cannot identify code with equivalent functionality that has been developed independently [165]. Another large share of review findings was dedicated to *over engineering* that was prevalent throughout the lab course. In many places students implemented functionality in a very complex fashion to suit possible future needs that had never been discussed by the course's advisors. Overall, manual reviews were perceived as an indispensable tool for assuring maintainability.

Concordant with our experience in industrial settings (Sec. 6.4), ConQAT proved to be well-suited to build dashboards that offer the required functionality in an efficient and effective manner. It was straightforward to implement the processors that extract the rating information from the source files and, crucially, once this was done, ConQAT's functionality for aggregation, visualization and historization could be applied without further changes. However, we found that most reviewers would have liked a tighter integration with the development environment. Similar to the prefilled checklists proposed in Chap. 4, the reviewers suggested to introduce a dedicated review view in Eclipse that, when reviewing a file, would list all findings stored in the dashboard. This would relieve developers from the tedious and error-prone switching between the IDE and the dashboard.

Regarding the applied maintenance process we found that, even in this asynchronous and distributed setting, it allowed us to control the quality of all development artifacts without reducing maintenance productivity. However, we also found that this process is very demanding if the number of reviewers is too small with respect to the number of developers. As in this course, three advisors had to review the code of 14 students, the advisors often had to struggle to complete their reviews to not delay progress. For a new lab course we, hence, plan to also involve students in the reviewing process by teaching review techniques at the beginning of the course.

Another problem we found concerns the artifact quality life cycle: An artifact that was rated GREEN once, stays GREEN unless it is modified. This notion can be problematic, however, as artifacts can, in fact, exhibit quality decay without being touched. An example for this problem is a class containing a method that implements a certain functionality that is later added to some library or framework. While the class was not modified, a new review would rate it RED as it contains duplicated functionality. We are aware of this inherent limitation of the artifact quality life cycle but do not believe that there is a generic solution to the problem. Instead, we currently rely on the fact that the affected artifact will sooner or later be changed due to some other reason. A review of this change will hopefully bring to light the missed quality deficiency, too.

6.5.5 Discussion

All lab course participants, students and advisors alike, agreed that the integrated application of manual and automated assessment techniques has been highly beneficial for them and the developed system. Advisors were glad that review efforts could be reduced, students enjoyed getting direct feedback on their coding and both parties profited from the high quality of the developed system.

Besides organizing lab courses, the CCSM maintains a code base of about 200,000 LOC that, among others, includes ConQAT and CloneDetective. These tools have a history of more than four years and are now successfully applied in various industrial and academic contexts. Due to the growing user base, these tools have clearly left the area of academic prototypes and demand professional development and quality assurance techniques. For this, we apply exactly the same processes and techniques as used in the lab course, i. e. every source artifact is reviewed until all quality criteria are met and products are only released if all their artifacts are rated GREEN. To achieve this, ConQAT-based dashboards that integrate various automated analyses and manual ratings are used. Based on this experience, we are convinced that such an integration enables us to control maintainability of large and changing systems in a highly productive manner.

6.6 Summary

The above case studies do not evaluate all aspects of the quality control approach proposed in this thesis in equal depth. However, we took great care to ensure that each aspect of the approach was thoroughly evaluated in at least one case study in an realistic setting. The case study undertaken with MAN showed that a QMM-based quality model can be used to describe the various quality criteria of a complex concept like model-based development in the domain of embedded systems. Furthermore, the study showed that quality guidelines can be generated from such a model and used in a serial development department. The suitability of QMM-based models is further emphasized by the case study undertaken with the Interasco GmbH where we used a quality model to evaluate multiple UI frameworks for web applications with respect to their maintainability. The case study concerning process variations of the mainframe development processes applied at BMW showed that a QMM-based model can be used to estimate the effect of specific quality criteria in a quantitative manner. The studies undertaken with Munich RE and ABB illustrate that quality dashboards help to continuously monitor and improve selected quality criteria in industrial settings. The study also showed that the quality control toolkit ConQAT is well-suited to build and customize such dashboards in an efficient and effective manner. This was confirmed by the case study set in an academic context where we showed that dashboards can be used to integrate manual and automated assessment techniques for the long-term quality control of diverse aspects of the multifaceted concept of maintainability.

»Quality is a complex and multifaceted concept.
It is also the source of great confusion...«

David A. Garvin

7 Beyond Maintainability

Considering the generic nature of the activity-based approach to model maintainability, the question arises if the same approach can be used to model other quality attributes. This chapter describes an application of the QMM for modeling *usability*. Furthermore, the chapter illustrates how activity-based models can be applied to describe all relevant quality attributes in an integrated manner.

7.1 Modeling Usability

Usability is a key quality attribute of successful software systems. Unfortunately, there is no common understanding of the factors influencing usability and their interrelations. Hence, there is a lack of a comprehensive basis for designing, analyzing and improving user interfaces. As this situation is very similar to maintainability, we applied the activity-based quality modeling approach to usability in order to evaluate in how far it can be used for quality attributes other than maintainability. To achieve this, we reviewed existing approaches to model usability and evaluated how activity-based quality modeling can overcome their deficiencies. We found, that by separating activities and system properties, sound quality criteria can be identified, thus facilitating statements concerning their interdependencies. A case study demonstrates how QMM-based models aid in revealing contradictions and omissions in existing usability standards. The application of the QMM for usability has partly been published in [304].

7.1.1 State of the Art

This section describes work in the area of quality models for usability. We discuss general quality models, principles and guidelines, and first attempts to consolidate the quality models.

Quality Models for Usability Hierarchical structures as quality models which focus mainly on *quality assurance* have been developed following Boehm's [38] and McCall et al.'s [51] original quality models. However, this kind of decomposition is too abstract and imprecise to be used for analysis and measurement. In addition, since usability is not a part of the main focus, this factor is not discussed in detail. In order to provide means for the operational measurement of usability, several attempts have been made in the domain of *human-computer interaction* (HCI). Prominent examples are the models from Shackel and Richardson [261] or Nielsen [212]. Nielsen, for example, understands usability as a property with several dimensions, each consisting of different components. He uses five factors: *learnability*, *efficiency*, *memorability*, *errors*, and *satisfaction*. *Learnability* expresses how well a novice user can use the system, while the efficient use of the system by an expert is expressed by *efficiency*. If the system is used occasionally, the factor *memorability* is used. This factor differentiates itself from *learnability* by the fact that the user has understood the system previously. Nielsen also

mentions that the different factors can conflict with each other. The ISO has published a number of standards which contain usability models for the operational evaluation of usability. Next to the ISO 9126 (see Sec. 3.2.1) that also includes the attribute *usability*, the ISO 9241 describes human-factor requirements for the use of software systems with an user interface. The ISO 9241-11 [149] provides a framework for the evaluation of a running software system. The framework includes the context of use and describes three basic dimensions of usability: *efficiency*, *effectiveness*, and *satisfaction*.

Principles and Guidelines In addition to the models which define usability operationally, a lot of design principles have been developed. Usability principles are derived from knowledge of the HCI domain and serve as a aid for the designer. For example, the “eight golden rules of dialog design” from Shneiderman [266] propose rules that have a positive effect on usability. One of the rules, namely *strive for consistency*, has been criticized by Grudin [128] for its abstractness. Grudin shows that consistency can be decomposed into three parts that also can be in conflict with each other. Although Grudin does not offer an alternative model, he points out the limitations of the design guidelines. Dix et al. [90] argue as well that if principles are defined in an abstract and general manner, they do not help the designer. In order to provide a structure for a comprehensive catalogue of usability principles. Dix et al. divide the factors which support the usability of a system into three categories: *learnability*, *flexibility*, and *robustness*. Each category is further divided into sub-factors. The ISO 9241-110 [157] takes a similar approach and describes seven high-level principles for the design of dialogs: *suitability for the task*, *self-descriptiveness*, *controllability*, *conformity with user expectations*, *error tolerance*, *suitability for individualization*, and *suitability for learning*. These principles are not independent of each other. For example, *self-descriptiveness* influences *suitability for learning*. Some principles have a part-of relation to other principles. For example, *suitability for individualization* is a part of *controllability*. The standard does not discuss the relations between the principles and gives little information on how the principles are related to the overall framework given in [149].

Consolidated Quality Models for Usability There are approaches which aim at consolidating the different models. Seffah et al. [260] applied the FCM model to the quality attribute *usability*. The developed model contains 10 factors which are subdivided into 26 criteria. For the measurement of the criteria the model provides 127 metrics. The motivation behind this model is the high abstraction and lack of aids for the interpretation of metrics in the existing hierarchically-based models. Put somewhat differently, the description of the relation between metrics and high-level factors is missing. In addition, the relation between factors, e. g. *learnability* vs. *understandability*, are not described in the existing models. Seffah et al. also criticize the difficulty in determining how factors relate to each other if a project uses different models. This complicates the selection of factors for defining high-level management goals. Therefore, in [260] a consolidated model that is called *quality in use integrated measurement model* (QUIM model) is developed. Since the FCM decomposition doesn't provide any means for precise structuring, the factors used in the QUIM model are not independent. For example, *learnability* can be expressed with the factors *efficiency* and *effectiveness* [149].

The same problem arises with the criteria in the level below the factors: They contain attributes as well as principles, e. g. *minimal memory load*, which is a principle, and *consistency* which is an attribute. They contain attributes about the user (*likeability*) as well as attributes about the product

(*attractiveness*). And lastly, they contain attributes that are similar, e. g. *appropriateness* and *consistency*, both of which are defined in the paper as capable of indicating whether visual metaphors are meaningful or not. To describe how the architecture of a software system influences usability, Folmer and Bosch [107] developed a framework to model the quality attributes related to usability. The framework is structured in four layers. The high-level layer contains *usability definitions*, i. e. common factors like *efficiency*. The second layer describes concrete measurable *indicators* which are related to the high-level factors. Examples of indicators are *time to learn*, *speed*, or *errors*. The third layer consists of *usability properties* which are higher level concepts derived from design principles like *provide feedback*. The lowest layer describes the *design knowledge* in the community. Design heuristics, e. g. the *undo pattern*, are mapped to the *usability properties*. Van Welie [286] also approaches the problem by means of a layered model. The main difficulty with layered models is the loss of the exact impact to the element on the high-level layer at the general principle level when a design property is first mapped to a general principle. Based on Norman's action model [214] Andre et al. developed the *User Action Framework* [6]. This framework aims toward a structured knowledge base of usability concepts which provides a means to classify, document, and report usability problems.

Summary In general, we found that the existing models for *usability* suffer from the same problems that have been identified in the case of *maintainability*.

1. Customizability. Many of the models discussed above are concrete quality models that are not designed to be adapted to a specific situation. As it is unrealistic to expect that one quality model would fit all software systems developed today, such models are prone to be ill-fitted for most systems as they either lack important criteria or define irrelevant ones.
2. Assessability. Most quality models contain a number of criteria that are too coarse-grained to be assessed directly. An example is the *attractiveness* criterion defined by the ISO 9126. Although there might be some intuitive understanding of attractiveness, this model clearly lacks a precise definition and hence a means to assess it.
3. Rationale. Additionally, most existing quality models fail to give a detailed account of the impact that specific criteria (or metrics) have on the user interaction. Again, the ISO standard cited above is a good example for this problem, since it does not provide any explanation for the presented metrics. Although consolidated models advance on this by providing a more detailed presentation of the relations between criteria and factors, they still lack the desired degree of detail. An example is the relationship between the criterion *feedback* and the factor *universality* presented in [260]. Although these two items are certainly related, the precise nature of the relation is unclear.
4. Structuredness. Due to a lack of clear separation of different aspects of quality, most existing models exhibit inhomogeneous sets of quality criteria. An example is the set of criteria presented in [260] as it mixes attributes like *consistency* with mechanisms like *feedback* and principles like *minimum memory load*.
5. Operationalization. As with *maintainability*, it remains unclear for most *usability* models how they should be operationalized in the software development process. Since *usability* is also an attribute that needs to be controlled continuously, we consider this a severe limitation.

7.1.2 Activity-Based Modeling of Usability

To address the problems with the quality models described in the previous section, we evaluated in how far the quality metamodel QMM is suited to model *usability*. Transferring the activity-based approach developed for maintainability to usability turns out to be straightforward. In the same manner *maintainability* describes all aspects that influence the activity maintenance, *usability* describes all aspects that have an impact on the activity usage. Also for usability we found the same dangerous mixture of activities and actual system properties as identified in the case of maintainability. A typical example of this problem can be found in [260] where *time behavior* and *navigability* are presented as the same type of criteria. Where *navigability* clearly refers to the navigation activity carried out by the user of the system, *time behavior* is a property of the system and not an activity. One can imagine that this distinction becomes crucial if the usability of a system is to be evaluated regarding different types of users: The way a user navigates is surely influenced by the system, but is also determined by the individuality of the user. In contrast, the response times of systems are absolutely independent of the user. A simplified visualization of the system property and activity decompositions as well as their interrelations is shown in Fig. 7.1 (attributes are not shown for clarity's sake). The activities are based on Norman's action model [214]. The model is described in more detail in Sec. 7.1.3.

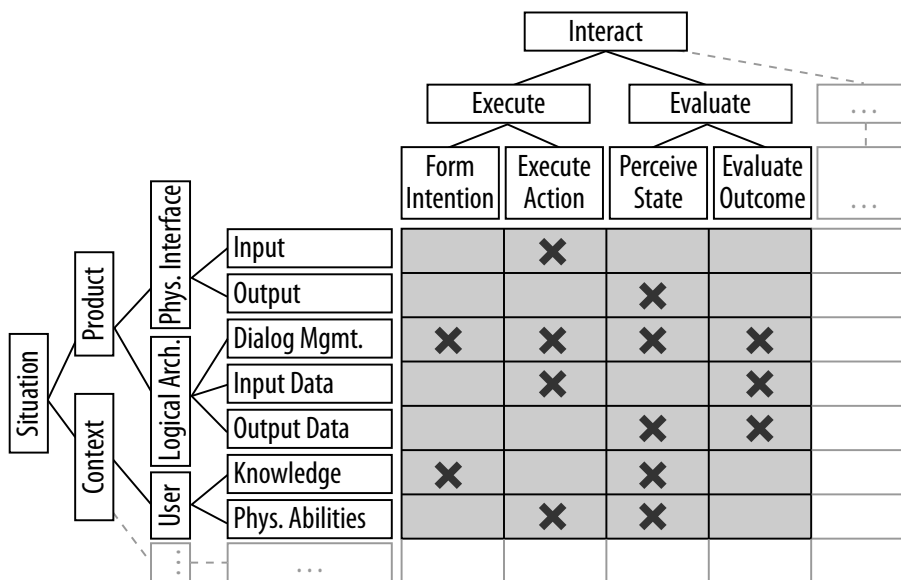


Figure 7.1: Simplified quality model

The final goal of usability engineering is to improve the *usage* of a system, i. e. to create systems that support the activities that the user performs on the system. Therefore, we are convinced that usability quality models should not only feature these activities as first-class citizens, but also precisely describe how properties of the system influence them and therewith ultimately determine its usability.

Our usability model does not only describe the product, i. e. the user interface, itself, but also comprises all relevant information about the situation of use (incl. the user). For this, entities and attributes of the QMM are used in the same way for modeling usability as they are used for maintain-

ability. For example, the model describes that the entity User Interface consists of the subentities Visual Interface and Aural Interface. Attributes are used to equip the entities with desired or undesired low-level quality criteria like CONSISTENCY, AMBIGUOUSNESS, or even the simple attribute EXISTENCE. Thus, the facts, which are tuples of entities and attributes, express system properties. An example is the fact [Font Face | CONSISTENCY] that describes the consistent usage of font faces throughout the user interface.

Activities also follow the decomposition introduced for maintainability models. Typically, the root node of the activity tree is the activity Interact that is subdivided into activities like Execute and Evaluate which in turn are broken down into more specific subactivities. *Activity attributes* allow to express which aspect of an activity is influenced by a fact. For example, we use [Font Face | CONSISTENCY] $\xrightarrow{+}$ [Reading | DURATION] to express that the consistency of the font face has positive impact on the Reading activity as it reduces the *time* it takes to process information on the screen. *Activity attributes* represent an extension to the metamodel used for maintainability. They help to make explicit the different types of impacts that are relevant for usability engineers. While an impact can be expressed by discussing the additional costs associated with it, we found that a finer-granular discussion that explicitly regards aspects like *Error Probability* and *Activity Duration* supports user interface designers in a more direct manner. Without activity attributes, this information would be captured in the prose impact description only. This can lead to inconsistencies and redundancy. It needs to be investigated more thoroughly if activity attributes can improve maintainability models, too. However, based on our experiences so far, we do not expect that the gained explicitness is worth the increased complexity of the metamodel in this case. As in most of the maintainability case studies, we use the impact set $I = \{-, +\}$ to express positive and negative impacts. Hence, impacts are defined as

$$[\text{Fact } f \mid \text{ATTRIBUTE } A_1] \xrightarrow{+/-} [\text{Activity } a \mid \text{ATTRIBUTE } A_2]$$

Another example is [Input Validity Checks | EXISTENCE] $\xrightarrow{-}$ [Data Input | ERROR PROBABILITY] that expresses that the existence of validity checks for the input reduces the likelihood of an error. We found that this extension of the metamodel helped to formalize aspects of the impacts that repeatedly occurred in QMM-based models for usability.

7.1.3 A Quality Model for Usability

Confident that activity-based models are, in general, capable of describing usability aspects, we applied a two-staged approach to evaluate this in detail: First, we built a QMM-based quality model that is based on the references given above as well as on company-specific quality models we encountered in projects with industrial partners. Second, we used this basic model to express the principles and guidelines of the ISO 15005 standard [153]. This standard describes ergonomic principles for the design of *transport information and control systems* (TICS).

The Activity Subtree »Interacting with the Product« The activity tree in the usability model has the root node Use that denotes any kind of usage of the software-based system under consideration. It has two children, namely Execution of Secondary Tasks and Interacting with the Product. The former stands for all additional user tasks that are not directly related to the software product. The

latter is more interesting in our context because it describes the interaction with the software itself. We provide a more detailed explanation of this subtree in the following.

The activity Interacting with the Product is further decomposed, based on the seven stages of action from Norman [214] that we arranged in a tree structure (Fig. 7.2). Scholars often use and adapt this model of action. For example, Sutcliffe [279] linked error types to the different stages of action and Andre et al. [6] developed the *User Action Framework* based on this model. We believe that this decomposition is the key for a better understanding of the relationships in usability engineering. Different system properties can have very different influences on different aspects of the use of the system. Only if these are clearly separated, we will be able to derive well-founded analyses. The three activities, Forming the Goal, Executing, and Evaluating, comprise the first layer of decomposition. The first activity is the mental activity of deciding which goal the user wants to achieve. The second activity refers to the actual action of planning and realizing the task. Finally, the third activity stands for the gathering of information about the world's state and understanding the outcome.

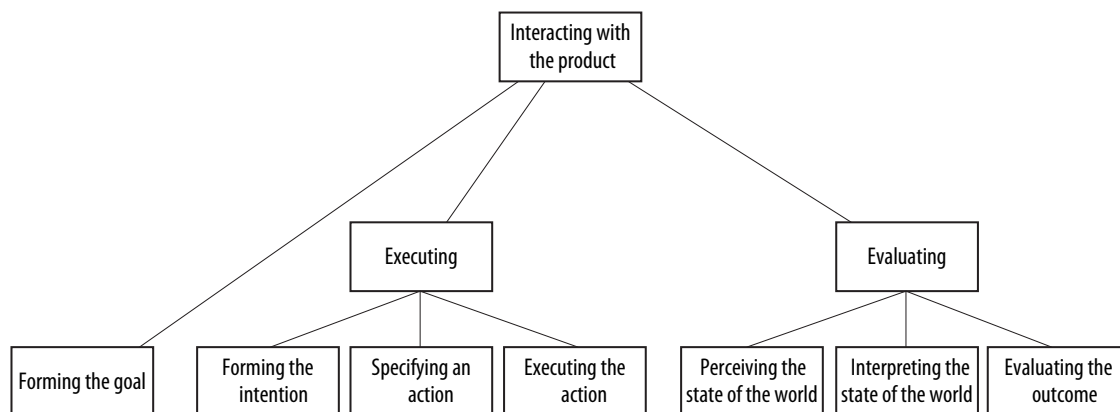


Figure 7.2: The subtree for »Interacting with the Product« (adapted from [214])

The activity Executing again has three children: First, the user forms his intention to do a specific action. Secondly, the action is specified, i. e. it is determined what is to be done. Thirdly, the action is executed. The activity Evaluating is decomposed into three mental activities: The user perceives the state of the world that exists after executing the action. This observation is then interpreted by the user and, based on this, the outcome of the performed action is evaluated.

Attributes To be able to define the relation of the facts and activities to general usability goals, we need to describe additional properties of the activities. This is done by a simple set of attributes that is associated with the activities:

- *Frequency.* The number of occurrences of a task.
- *Duration.* The amount of time a task requires.
- *Physical stress.* The amount of physical requirements necessary to perform a task.
- *Cognitive load.* The amount of mental requirements necessary to perform a task.

- *Error Probability.* The distribution of successful and erroneous performances of a task.

The Entity Subtree »Logical User Interface« The entity tree in the usability model contains several areas that need to be considered in usability engineering, such as the physical user interface or the usage context. By means of the entity User, important properties of the user can be described. Together with the entity Application it forms the Context of use. The Physical Output Devices and the entity Physical Input Devices are assumed to be part of the Physical User interface. However, we concentrate on a part we consider very important: the entity Logical User Interface. Fig. 7.3 shows the decomposition of the entity tree in the form of an architecture diagram to illustrate the dependencies between entities.

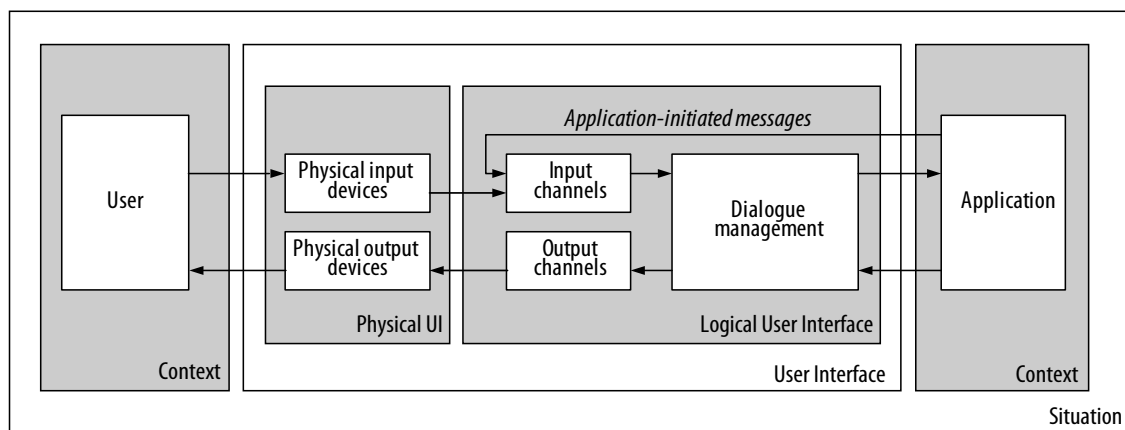


Figure 7.3: The user interface architecture

The logical user interface contains Input Channels, Output Channels, and Dialog Management. In addition to the architecture, we also add data that is sent via the channels explicitly: Input Data and Output Data (not shown). The architecture in Fig. 7.3 also contains a specialization of input data, Application-Initiated Messages. These messages, which are sent by the Application, report interrupts of the environment or the application itself to the Dialog Management outside the normal response to inputs. The entity Application-Initiated Messages is required to model quality aspects in so-called *adaptive* systems that aid the user by automatically initiating certain actions.

Attributes The attributes play an important role in the quality model because they are the properties of the entities that can actually be assessed manually or automatically. It is interesting to note that it is a rather small set of attributes that is capable of describing the important properties of the facts. Moreover, we observe that the attributes used in the usability model differ only slightly from the ones contained in the maintainability model. Hence, there seems to be a common basic set of those attributes that is sufficient – in combination with facts – for quality modeling.

- *Existence.* The most basic attribute that we use is whether a fact exists or not. The pure existence of a fact can have a positive or negative impact on some activities.

- *Relevance.* When a fact is relevant, it means that it is appropriate and important in the context in which it is described.
- *Unambiguousness.* An unambiguous fact is precise and clear. This is often important for information or user interface elements that need to be clearly interpreted.
- *Simplicity.* For various facts it is important that in some contexts they are simple. This often means something similar to small and straightforward.
- *Conformity.* There are two kinds of conformity: conformity to existing standards and guidelines, and conformity to the expectations of the user. In both cases the fact conforms to something else, i. e. it respects and follows the rules or models that exist.
- *Consistency.* Consistency means that the entire product follows the same rules and logic and, hence, exhibits a homogeneous behavior.
- *Congruence.* Congruence aims at correspondence with external facts, such as analogies, or a common understanding of things. Congruence is sometimes referred to as *external consistency*.
- *Controllability.* A controllable fact is a fact which relates to behavior that can be strongly influenced by the actions of the user. The user can control its behavior.
- *Customizability.* A customizable fact is similar to a controllable fact in the sense that the user can change it. However, a customizable fact can be preset and fixed to the needs and preferences of the user.
- *Guardedness.* In contrast to customizability and controllability, a guarded fact cannot be adjusted by the user. This is a desirable property for some critical parts of the system.
- *Adaptability.* An adaptive fact is able to adjust to the user's needs or to its context depending on the context information. The main difference to customizability is that an adaptive fact functions without the explicit input of the user.

Examples The entire model is composed of the activities with attributes, the entities with the corresponding attributes (facts) and the impacts between facts and attributed activities. The model with all these details is too large to be described in detail, but we present some interesting examples: triplets of a fact, an attributed activity and a corresponding impact. These examples aim to demonstrate the structuring that can be achieved by using the quality metamodel.

Consistent Dialog Management. A central component in the logical user interface concept proposed in Sec. 7.1.3 is the Dialog Management. It controls the dynamic exchange of information between the product and the user. In the activities tree, the important activity is carried out by the user by interpreting the information given by the user interface. One attribute of the dialog management that has an impact on the interpretation is its CONSISTENCY. This means that its usage concepts are similar in the entire dialog management component. The corresponding impact description describes that a consistent dialog management reduces the error probability:

$$[\text{Dialog Management} \mid \text{CONSISTENCY}] \xrightarrow{+} [\text{Interpretation} \mid \text{ERROR PROBABILITY}]$$

Obviously, this is still too abstract to be easily assessed and needs to be refined for the specific context. For example, menus in a graphical user interface should always open the same way.

Guarded Physical Interface. The usability model does not only contain the logical user interface concept, but also the physical user interface. The Physical Interface refers to all the hardware parts that the user interacts with in order to communicate with the software-based system. One important attribute of such a physical interface is GUARDEDNESS. This means that the parts of the interface must be guarded against unintentional activation. Hence, the guardedness of a physical interface has a positive impact on activity Executing as it reduces the error probability:

$$[\text{Physical Interface} \mid \text{GUARDEDNESS}] \xrightarrow{+} [\text{Executing} \mid \text{ERROR PROBABILITY}]$$

A physical interface that is not often guarded is the touchpad of a notebook computer. Due to its proximity to the location of the hands while typing, the cursor might move unintentionally. Therefore, a usability model of a notebook computer should contain the triplet that describes the impact of whether the touchpad is guarded against unintentional operation or not.

7.1.4 Modeling the ISO 15005

To further evaluate our usability modeling approach we refined the high-level model described in the last section into a specific model based on the ISO 15005 [153]. The goals of this evaluation were twofold: First, we wanted to demonstrate that the activity-based approach can be used to model the criteria expressed in the standard. Second, we wanted to discover inconsistencies, ill-structuredness, and implicitness of important information in the standard.

The standard describes ergonomic principles for the design of *transport information and control systems* (TICS). Examples for TICS are driver information systems (e. g. navigation systems) and driver assistance systems (e. g. cruise control). In particular, principles related to dialogs are provided, since the design of TICS must take into consideration that a TICS is used in addition to the driving activity itself. The standard describes three main *principles* which are further subdivided into eight *sub-principles*. Each sub-principle is motivated and consists of a number of *requirements* and/or *recommendations*. For each requirement or recommendation a number of examples is given. For example, the main principle *suitability for use while driving* is decomposed among others into the sub-principle *simplicity*, i. e. the need to limit the amount of information to the task-dependent minimum. This sub-principle consists, among others, of the recommendation to optimize the driver's mental and physical effort. All in all the standard consists of 13 requirements, 16 recommendations, and 80 examples. For translating the ISO 15005 to a QMM-based model we applied an approach similar to the one we used for translating quality guidelines for Simulink models (Sec. 6.1). Hence, we inspected each principle, sub-principle and requirement and expressed it with suitable elements, i. e. facts, attributes, impacts, etc. of the QMM. The final model consisted of 41 entities, 12 activities, 15 attributes, 48 facts, and 51 impacts.

Examples To illustrate how the elements of the standard are represented in our model, we present the following examples. An element in the logical user interface concept presented in Sec. 7.1.3 is the Output Data, i. e. the information sent to the driver. A central aspect is the representation of the data. One attribute of the representation that has an impact on the interpretation of the state of the system is its UNAMBIGUOUSNESS, i. e. that the representation is precise and clear. This is especially important so that the driver can identify the exact priority of the data. For example, warning messages are represented in a way that they are clearly distinguishable from status messages.

$$[\text{Output Data} \mid \text{UNAMBIGUOUSNESS}] \xrightarrow{+} [\text{Interpretation} \mid \text{ERROR PROBABILITY}]$$

Another attribute of the representation that has an impact on the interpretation is the CONSISTENCY. If the representations of the output data follow the same rules and logic, it is easier for the driver to create a mental model of the system. The ease of creating a mental model has a strong impact on the ease of interpreting the state of the system:

$$[\text{Output Data} \mid \text{CONSISTENCY}] \xrightarrow{+} [\text{Interpretation} \mid \text{DURATION}]$$

One attribute of the representation that has an impact on the perception is SIMPLICITY. It is important for the representation to be simple, since this makes it easier for the driver to perceive the information:

$$[\text{Output Data} \mid \text{SIMPLICITY}] \xrightarrow{+} [\text{Perception} \mid \text{COGNITIVE LOAD}]$$

A TICS consists of several features which must not be used while driving the vehicle. This is determined by the manufacturer as well as by regulations. One important attribute of such features is its GUARDEDNESS. This means that the feature is inoperable while the vehicle is moving. This protects the driver from becoming distracted while using the feature. The guardedness of certain features has a positive impact on the driving activity as the error probability is reduced:

$$[\text{Television} \mid \text{GUARDEDNESS}] \xrightarrow{+} [\text{Driving} \mid \text{ERROR PROBABILITY}]$$

Observations & Improvements As a result of the QMM-based analysis, we found the following inconsistencies and omissions in the ISO 15005 standard:

- *Inhomogeneous Decomposition.* Like many other quality models and guidelines the ISO 15005 exhibits an inhomogeneous decomposition. For example, two of the three main principles refer to activities (driving and usage of the TICS) whereas the third main principle namely *suitability for the driver* describes the user but not activity he carries out. The mix of different aspects within the decomposition is continued on the level of sub-principles. Some sub-principles describe activities, some describe properties of the system and others characteristics of the user. As argued before, such an seemingly arbitrary decomposition complicates reasoning about the completeness of the model. One possibility to resolve this mix-up, could be to promote sub-principles that only describe activities to main principles while degrading those sub-principles that describe software entities to the requirements level.

- *Requirements with Implicit Impacts.* 9 out of 13 requirements do not explicitly describe impacts on activities. Requirements serve to define the properties which the system entities should fulfill. If a requirement does not explicitly describe its impacts on activities, the impact could be misunderstood by the software engineer. Hence, we suggest that requirements should be described by facts and their impacts on activities.
- *Incomplete Examples.* 14 out of 80 examples only describe facts and their attributes, leaving the impacts and activities implicit. To provide complete examples, we suggest that the examples should be described with explicit impacts and activities.
- *Inconsistent Terminology.* The German translation of the ISO 15005 exhibits an inconsistent terminology. For example, the sub-principle *Controllability* is referred to in some places as *Steuerbarkeit* and *Kontrollierbarkeit* in other places.

7.1.5 Discussion

Usability is a quality attribute that is at least as complex and multifaceted as maintainability. Similarly to maintainability, there is no widely accepted method for explicitly defining what constitutes usability. We found that most previous approaches for modeling usability suffer from similar shortcomings as comparable methods for maintainability do. The problem is that usability modeling approaches are either (1) too inflexible, (2) do not define assessable criteria, (3) do not provide rationale for the criteria or (4) exhibit inhomogeneous criteria break-downs due to the lack of a defined decomposition criterion. Also, it is left unclear for many usability models how they can be operationalized, e. g. for application in constructive and analytic quality assurance.

We found that the quality modeling approach originally developed in the context of maintainability can be applied to usability with minor adjustments only. As »to use« is one of the most fundamental *activities*, it is no surprise that the activity-based nature of the quality metamodel QMM can be applied smoothly to usability, too. The clearly defined decomposition rules of the entity tree help to structure the system and its environment so that clearly understandable impacts on the activities can be described. The case study related to the ISO 15005 standard described above demonstrated that such clearly structured models help to identify omissions and inconsistencies that are prone to go unnoticed in traditional usability guidelines. For example, the standard contains three sub-principles which describe activities, but no impacts on them, as well as 9 requirements that have no described impacts. This hampers the justification of the guideline: A rule that is not explicitly justified will not be followed.

While we did not evaluate this in detail, it is obvious that the same mechanisms that are used to generate guidelines and review checklists for maintainability could be applied to the usability models as well. This allows to operationalize usability models to make them an integral part of the usability engineering process. It is, however, unclear in how far automated analyses can be applied to support analytic means of quality assurance. In our example models, we found only a limited amount of facts that lend themselves to automatic analysis. An example is the minimum size of fonts shown on displays.

One problem is that, in some contexts, the term *usability* includes aspects as fuzzy and subjective as *aesthetics*. Activity-based quality models still help to reason about the user activities that are affected

by such aspects. However, we found that decomposition we apply in the entity tree clearly reaches its limits for such aspects, as there is no obvious way of breaking down e. g. *aesthetics* to the different entities of a system.

7.2 An Integrated Approach for Quality Modeling

The previous section has shown that QMM-based model can be used to model usability in a similar fashion to maintainability. However, both quality attributes were modeled in isolation only. This section describes how activity-based quality models can be used to build integrated quality models that cover different quality attributes while minimizing overlaps and inconsistencies. This work was, in parts, already published in [84, 291, 295].

Overlaps, Inconsistencies & Trade-offs Though there usually is an emphasis on certain quality criteria, e. g. safety in the automotive domain, software development organizations do not focus on a single quality attribute but need to cover a broad quality spectrum. Today they do so by applying different quality models in isolation and thereby create a situation that makes it hard to recognize overlaps and inconsistencies in the various models. For example, many coding guidelines that target *security* contain rules that also appear in guidelines for *safety* or *maintainability*. An example is the rule that states that floating point variables should not be used as counter variables in FOR loops. This is stated by the MISRA guidelines [209], which mainly focus on safety, as well as by the *CERT Sun Microsystems Secure Coding Standard for Java*¹, which focus on security.

Moreover, this situation does not allow to discuss quality trade-offs in a systematic way. A classic example is the long-disputed trade-off between *portability* and *performance*. While [281] reports that in Microsoft Windows portability was deliberately sacrificed for performance, [122] found that for OR-mapping frameworks this trade-off is virtually nonexistent. Another example is the often-encountered trade-off between security and usability as some security features make systems less comfortable to use. We claim that the lack of a systematic concept to integrate the different quality models renders a comprehensive analysis of software difficult and causes overlaps as well as inconsistencies in definitions of quality.

Focus on Stakeholders We found that QMM-based quality models provide a good basis for an integrated modeling of various quality attributes as activities can be applied nearly universally as a structuring mechanism. To understand this, it's necessary to recall that making activities first-class citizens in quality models is not an end in itself but helps to decompose the non-conformance quality cost that are usually difficult to untangle. Applying an activity-based approach in an integrated context that includes quality attributes as different as maintainability, usability and performance obviously is more challenging than limiting the approach to software maintenance. We found that the central problem is to identify all relevant, i. e. cost-inducing, activities. For this, it proved to be useful to take the perspectives of the different *stakeholders* of a software system into account. A software system has multiple stakeholders including the *users* of the system, *developers* that maintain the system and *operators* that take care of the system's operation. While these stakeholders are rather obvious there are less evident ones like *trainers* that train the system's users or even *attackers* that try to penetrate the system's security features. Each of the stakeholders carries out multiple activities with or on the system and each of these activities is associated with conformance and non-conformance quality costs. For example, using a system becomes costly for the users if the systems has poor performance

¹<https://www.securecoding.cert.org/confluence/display/java/The+CERT+Sun+Microsystems+Secure+Coding+Standard+for+Java>

or often crashes due to reliability problems. If the usability is poor, also the costs for user training are prone to be high. The reliability problems also affect the operation costs as the operator has to restart the system frequently. The maintenance cost are obviously affected by the maintainability of the system. Finally, costs also arise if the attacker is not prevented from carrying out his malevolent activities. The quality model presented in this thesis does not attempt to model the blurry concept of *maintainability* but focuses on the impact that non-conformance to various quality factors has on the *maintenance costs*. In precisely the same manner, the integrated model proposed here focuses on the impact that non-conformance has on the total life cycle costs of a software product.

Integrated Modeling Fig. 7.4 exemplifies how the diverse concerns and relationships can be modeled in an integrated manner using a QMM-based quality model. Depicted are 7 facts that concern the software part of the product, its documentation and its user interface. Four stakeholders carry out 6 different *atomic* activities on or with the system. The fact [User Manual | COMPLETENESS] affects almost all activities as developers, operators and users alike benefit from it. Unfortunately, the details provided in the manual might also support an attacker in concocting an attack plan, e. g. based on the well-known SQL injection. The completeness of the glossary helps the user to understand messages generated by the system but also aids the developer in locating domain concepts in the code. While consistency of identifiers mainly concerns the developers, it can also affect operators as they must maintain configuration files that often use the terminology of the identifiers in the code. Source code redundancy (clones), however, is a fact that affects only the developers of the system, particularly when they modify it. The same applies to the fact [Variable | LOCALITY] that expresses that program variables should be defined with a scope as minimal as possible. The appropriateness of the font faces, on the other hand, concerns only the user of the system. Interestingly, the fact [Input Validation | EXISTENCE] does not only aid the user but also prevents the attacker from executing SQL injection attacks.

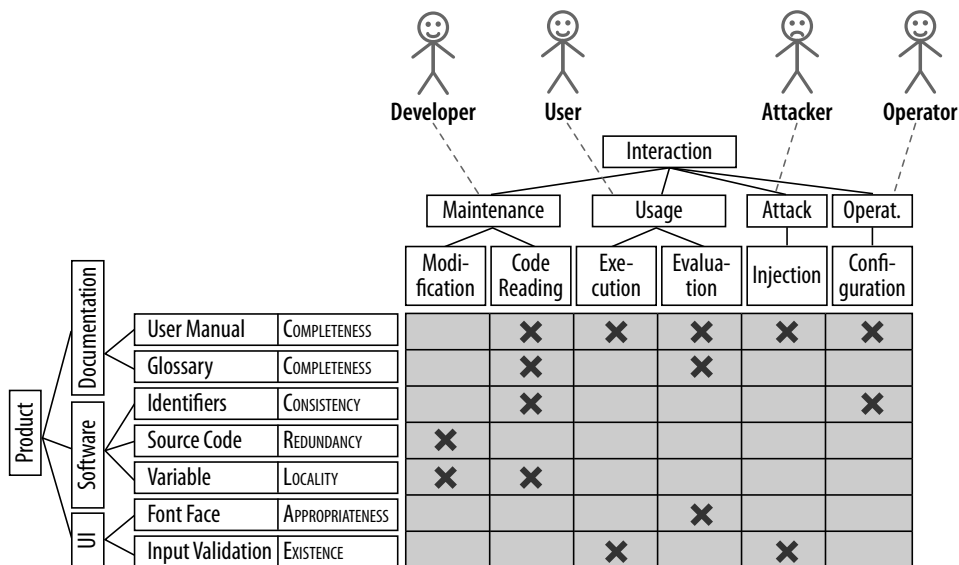


Figure 7.4: An Integrated QMM-based Quality Model

The integrated model allows to reason about the various factors that affect quality and, importantly, their interdependencies. Consequently, the overlaps that frequently occur if isolated models are used for different aspects are resolved. This does not only reduce quality model redundancy but prevents inconsistencies between models. Moreover, the integrated model provides a basis for the structured discussion of trade-offs between different quality aspects. Importantly, such an integrated model can be used in the same manner the maintainability models where used in the case studies presented in Chap. 6. For example, quality guidelines and review checklists can be generated from it. Using working sets or similar modularization mechanisms, such documents can be tailored for the specific user and, hence, kept concise while it is still ensured that all documents are consistent with each other.

It is worth to note that the integrated model deliberately does not include the well-known quality attributes («-ilities») like *maintainability* or *security* that are usually used in the discussion of different quality aspects. As argued in Chap. 3, we came to the conclusion that the classic «-ilities» do not provide a sound basis for structuring the various facets of quality since they do not allow to reason about the costs associated with non-conformance. From our point of view, it is irrelevant if a certain fact, e. g. the use of floating point variables as loop counters, affects maintainability or security. The relevant question is: »What costs are associated with the a use of an inappropriate data type?« While we cannot answer such questions in a quantitative manner right now, we are convinced that activities, that are known to be a solid basis for cost structure analysis in many other areas, will provide the key for a better understanding of quality and the costs associated with it.

7.3 Summary

The application of QMM-based models in the context of *usability* showed that the activity-based concept is not limited to modeling factors important for software maintenance but can be applied for other quality attributes, too. This is confirmed by Wagner and Islam's successful application of QMM-based quality models for software *security* [296]. Furthermore, we illustrated how activity-based models can be used to build integrated quality models that describe aspects as different as usability, maintainability, security or performance in *one* model. In comparison to the use of multiple isolated models, this helps to reduce overlaps and inconsistencies between models and fosters a substantiated discussion of quality trade-offs.

»Science never solves a problem without creating ten more.«

George Bernard Shaw

8 Summary and Outlook

In this final chapter we summarize the contributions of this thesis and put them into context. We also describe our current work and directions for possible further research.

8.1 Summary

Between 60% and 80% of the total life-cycle cost of long-lived systems are spent during the maintenance phase rather than the initial development phase. As half of these efforts are devoted to changing existing functionality and implementing new requirements, the ability to perform software maintenance in a cost-effective and timely manner is a key factor for the commercial success of software developing organizations. With a focus on the software systems themselves, this ability is commonly referred to as *maintainability*. Notwithstanding its undoubted importance, most organizations today do not apply dedicated processes, tools and techniques to assure maintainability. Thus, we today lack an established discipline of *maintainability engineering*. We consider this a precarious situation; particularly because maintainability is known to degrade while software systems evolve.

We claim that the reluctance to actively control maintainability is, foremost, caused by the lack of a precise definition of maintainability: Organizations do not control maintainability since they do not know what it really is. An in-depth investigation of the state-of-the-art in defining maintainability as well as our practical experience, revealed that existing approaches are inadequate as they either (1) describe vague criteria that cannot be actually assessed, (2) do not provide rationale for the presented criteria or (3) exhibit insufficient structures that prevent reasoning about the completeness and consistency of quality definitions. Moreover, most existing approaches cannot be operationalized in the software maintenance process and, hence, provide no solid basis for constructive and analytic quality assurance activities.

This thesis proposes a new quality modeling approach that helps to create a precise and assessable definition of maintainability. The approach is based on the rigidly defined quality metamodel QMM that carefully distinguishes aspects that are typically intermingled: *entities* the system consists of, *attributes* that describe wanted or unwanted characteristics of the entities and maintenance *activities* that are positively or negatively affected by these characteristics. This separation creates distinct hierarchies with clearly defined decomposition criteria that facilitate a structured decomposition of maintainability. By making the impact on maintenance activities explicit, the model provides the rationale for the defined quality requirements so that model users can comprehend why they are included in the model. As the maintenance activities provide a breakdown of the total maintenance effort, this can also contribute to a discussion of quality economics. Furthermore, the quality metamodel requires each fact to be equipped with a precise assessment description and a classification of the assessment type (manual, semi-automatic, automatic) to avoid the definition of non-assessable facts.

Most previous approaches do not provide means for communicating the quality requirements defined by a quality model to developers and other project participants to support constructive quality assurance. We advance on this by automatically generating quality guidelines from quality models that can be tailored to suit the needs of the target audience. To support analytic quality assurance, review checklists as they are typically used in inspections can be automatically generated from activity-based quality models. Moreover, the results of automatic quality assessment tools like static code analyzers can be explicitly put into relation with the quality model. This allows a seamless integration of manual and automatic quality assessment techniques and thereby paves the way for a practice of *continuous quality control* for maintainability.

While many existing approaches view quality models as pen&paper artifacts, the modes of operationalization proposed in this thesis require rich tool support for the creation, manipulation and transformation of quality models that need to be stored in a machine-processable format. We support the creation and maintenance of quality models with a dedicated quality model editor application. This editor can be used to generate quality guidelines and review checklists. With respect to analytic quality assurance, there already is a plethora of automatic quality analysis tools of academic and commercial origin. However, the majority of existing tools operates virtually independently from the definition of quality. We advance on this by providing the *Continuous Quality Assessment Framework* ConQAT that allows to integrate the results of various quality analysis tools as well as manual reviews into a quality control *dashboard*. The dashboards can put quality assessment results into relation with the explicitly defined quality model and thereby enables users to judge if the quality requirements expressed by the model are met. ConQAT advanced to a state that is clearly beyond a mere research prototype and is now used by multiple major companies and number of research groups.

Several case studies were undertaken to evaluate if our modeling approach and the accompanying tool-chain can be applied in practice. In a study with the MAN Nutzfahrzeuge Group we used a activity-based quality model to describe quality criteria of Matlab/Simulink models that are frequently used for the development of embedded systems. Quality guidelines for developers were generated from the quality model and automatic assessments of model quality were integrated with MAN's process infrastructure. The case study showed that our quality modeling approach is well-suited to create a precise definition of quality criteria for maintainable Matlab/Simulink models and led to the adoption of our approach as well as our tools in the MAN development process. In a study undertaken with the Interasco GmbH, a QMM-based quality model was used to evaluate the expected maintenance efforts of different web user interface frameworks. Our modeling approach allowed to structure the relevant quality criteria and thereby guided Interasco in selecting a suitable framework from a plethora of available options. The results of the study have been used by Interasco as input for a major reengineering of one of their core products. Together with the BMW Group we evaluated the economic impact of variations in the project infrastructure on the different maintenance activities in the context of mainframe software development. This case study showed that QMM-based models are principally suited to reason about the impact of quality criteria in a *quantitative* manner. ConQAT's suitability to build quality dashboards in an industrial context was evaluated with our partners Munich Re Group and ABB Distribution Automation. At both partners ConQAT was used to control selected quality criteria in multiple projects. In both cases, ConQAT-based quality dashboards not only helped to keep the current state of quality but also supported step-by-step quality improvements. In a case study that was undertaken in a programming lab course, we evaluated how manual and automatic quality assessment techniques can be tightly integrated using quality dashboards. The integration helped to improve the quality of the developed software while reducing the efforts for

manual reviews. Moreover, the continuous feedback provided by the dashboard helped students to improve their programming skills.

Finally, we evaluated if the quality modeling approach that was developed in the context of maintainability can be transferred to other quality attributes. In a case study dedicated to *usability* we found that only minor changes to the quality metamodel were required to apply the same approach to an attribute that, at first sight, appears to be highly different from maintainability. Furthermore, we showed how the activity-based quality metamodel can be used to build integrated quality models that describe quality attributes as different as maintainability, security and portability in *one* model. This helps to avoid the overlaps and inconsistencies that frequently occur if multiple isolated models are used. In addition, the integrated quality model supports a substantiated discussion of trade-offs between quality attributes.

8.2 Outlook

While the quality control approach presented in this thesis addresses major shortcomings of existing approaches, it would be bold to claim that all problems are solved. This section discusses possible improvements of the presented approach and illustrates directions for further research on modeling maintainability in particular and quality in general.

Assessment & Aggregation Currently, the quality metamodel QMM provides support for quality assessments only through the prose assessment descriptions that are associated with the quality criteria. While this helps to avoid the formulation of criteria that are not assessable, one would often want a more formalized assessment description that includes explicit metrics. Moreover, the QMM provides no support for aggregating assessment results to higher levels. As there are multiple possible aggregation dimensions and aggregation methods' suitability strongly depends on the goals of the aggregation, aggregation is currently left to the quality assessment tools. While flexible tools like ConQAT can be customized to perform all required types of aggregations, omitting precise aggregation instructions from the quality models itself leads to an undesirable degree of freedom in quality model interpretation. Hence, it would be beneficial to extend the quality metamodel to include formalized assessment and aggregation instructions. Referring to the example in Sec. 4.2.4, for example, the model would contain an instruction that states that an instance of the entity For Loop conforms to the quality model if its facts and the facts of its parts, e. g. [For Loop | APPROPRIATENESS] and [Body | WELL-FORMEDNESS], are fulfilled. A Method's instruction could state that a method is conform if all the For Loops as well as the other program constructs it contains are conform. Such instructions could be formalized with a technique similar to the detection strategies used in [201].

Software Development Life Cycle The focus of the approach presented in this thesis is clearly on the late phases of the software development life cycle (SDLC). However, quality requirements, including the ones that concern software maintenance, should ideally be discussed in all stages of the SDLC; beginning with the tender preparation. Hence, we evaluated initial ideas to apply activity-based quality models throughout the SDLC and, particularly, in its early stages. In [295] we propose the 5-staged process shown in Fig. 8.1 to elicitate and refine quality requirements:

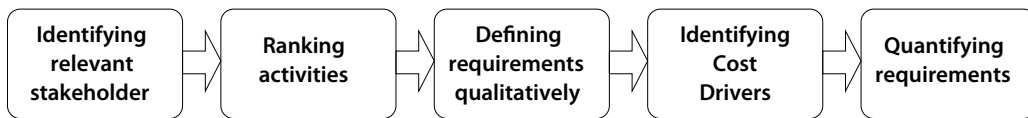


Figure 8.1: Process Steps for Quality Requirements Elicitation and Refinement

1. *Identifying Relevant Stakeholders & Activities.* The first step is, similar as in other requirements elicitation approaches, to identify the stakeholders of the software system. For quality requirements, this usually includes users, developers, maintainers and operators. However, it is often sensible to include less evident stakeholders like user trainers or even malevolent *attackers*. When the stakeholders have been identified, the quality model structure can be used to derive the activities they perform on and with the system. For example, the activities for the maintainer include *concept location*, *impact analysis*, *coding*, or *modification*. Especially the activities of the user can be further detailed by developing usage scenarios.
2. *Ranking Activities.* In the next step, we rank the activities of the relevant stakeholders according to their importance. This results in an list of all activities of the relevant stakeholders. On top of this list are the most important activities, the least important at the bottom. For this, *importance* is best defined via the expected costs associated with an activity. In practice, the importance can be judged by experts or based on experiences from similar projects.
3. *Defining Requirements Qualitatively.* After activities have been ranked, a qualitative definition of requirements is required. For example, if a software system is to be installed on many user workstations, the Installation activity is desired to be *highly automated*. If the software is expected to undergo frequent changes, its Modification should be supported well. If the data security is an issue, the Attack should be as difficult as possible. Hence, quality requirements are expressed in purely activity-based manner.
4. *Identifying Cost Drivers.* Starting with the most important activities, cost drivers for the defined qualitative requirements are identified. They are modeled in form of the quality model's entity tree and put into direct relation to the activities through explicitly modeling their impacts. For example, the activity Testing is positively influenced by the conformance of software components to a common interface. The well-defined structure of the entities tree supports building the model and allows to reason about its completeness. For example, one can relatively easy distinguish cost drivers that are inherent to the system and cost drivers that are located in the systems operational environment. The Testing activity again provides a good example as testing is not only influenced by the system under test but also by the availability of adequate test tools.
5. *Quantifying Requirements.* Finally, the goal is to define quantitative and, hence, assessable requirements. Requirements can be quantified on the activity-level or on the entity-level. An example on the activity level is, that an average Modification activity should take 4 person-hours to complete. Requirements on the entity-level might be quantitatively assessable depending on the attribute concerned. For example, *superfluous code variables* can be counted and an upper limit can be defined. While we assume that it is theoretically possible to quantitatively define all requirements, we are aware that this is not always feasible in practice as either there is no known decomposition of the requirement or its impact is not understood well enough yet.

While this proposal for an activity-based approach to quality requirements worked in our case studies [295], there clearly still is a long way to go to apply quality models throughout the SDLC in a consistent manner. In particular, it is currently unclear how one should deal with the problem that most important system artifacts do not exist in the requirements phase and, hence, cannot be included in the entities tree. To address this problem, we envision a dedicated quality requirements process that is tightly coupled with the main development process. This process serves the step-wise refinement of quality requirements in the same manner a mature development process builds the software system as a series of step-wise refinements from the requirements to the actual software system. If both processes are properly synchronized, the introduction of new entities by the main development process automatically triggers a refinement of the quality requirements. This stepwise refinement again is supported by the quality model structure. The questions to be answered at each refinement step are: »What are the important attributes of the new entity and what is their impact on relevant activities?«

Economic Aspects The activity-based notion that underlies the quality modeling approach presented in this thesis is strongly shaped by economic considerations. However, the quality models built with this approach cannot directly answer questions regarding economic aspects yet. As we view a true economically justification as the ultimate goal for the practice of quality engineering, this is clearly a shortcoming. We currently see two important directions of further work to remedy this:

- *Cost Models.* To be able to reason about economic aspects, quality models must be extended with or put into relation to explicit cost models that allow to describe the effect of quality characteristic in a quantitative manner. By using adequate cost models the conformance and non-conformance costs induced by quality criteria could not only be used as guiding notion in quality modeling but could be actually calculated. A first step into this direction is described in Sec. 6.3 where we used an approach based on absorbing Markov chains to evaluate the different costs of two process alternatives in the context of mainframe software development. However, with respect to quality modeling, this application was limited to a *single* quality criterion. Hence, we currently do not know in how far the applied technique can be generalized to the multiple hundred quality criteria described by realistic quality models. Another potential candidate to address this problem, are *Bayesian Networks* that Wagner used in conjunction with activity-based quality models for predictive purposes [290].
- *Empirical Data.* Independent of the type of model that is applied to reason about economic aspects, empirical data is needed to use such models in a sensible way. Currently, there is a tremendous lack of empirical data regarding many highly relevant questions in software quality. For example, almost no organization today is able to quantify the impact of outdated documentation or code cloning in economic terms. Much less, there is reliable data for specific domains let alone the whole software engineering industry. To remedy this more and larger empirical studies need to be carried out. For this, however, organizations must collect more data in a more structured manner which, of course, first requires them to value the long-term economic benefit generated by such studies.

We are confident, that activity-based models will help support a stronger economic perspective in quality engineering as activities are known to be a solid basis for cost structure analysis in diverse areas.

Model Reuse and Standardization Building an integrated quality model that covers all aspects relevant for an organization is certainly a herculean task. A combination of all quality models described in the chapter on case studies (Chap. 6) already contains more than 800 model elements although the focus was on software maintenance only. A fully integrated model that includes usability, security and other aspects is expected to have multiple thousands of elements. Hence, it is not to be expected that a single organization is capable of building and maintaining an integrated quality model on its own. Instead, we advocate *reusable* quality models that can be shared across projects, organizations or, as a final goal, domains. Ideally, such quality models would be developed by the ISO or similar standardization bodies. In conjunction with an adequate conformance assurance process this would provide a means for product quality *standardization* that goes far beyond anything that is currently offered. Existing standardized quality models like the ISO 9126 neither provide a means to assess if a product conforms to the quality model nor do they support the developing organizations in building products that conform to the model.

Due to the size of such models, a *modularization* mechanism that allows to compose quality models from invidious building blocks is required. Ideally, a *base quality model* would be defined that can then be extended with *specific* models for particular quality aspects, technologies or domains. With such a mechanism organizations or projects could *tailor* quality models to their specific needs while still being consistent with a standardized model. For example, an organization that develops products in the domain of embedded systems using a combined code- and model-based approach with Matlab/Simulink and C, would use the base model plus extensions dedicated to safety critical systems and the applied technologies. An organization that develops business applications based on the Java Enterprise platform (Java EE) would use the base model plus extensions for Java EE and, most likely, extensions dedicated to security of web applications. The modularization of quality models is illustrated in Fig. 8.2.

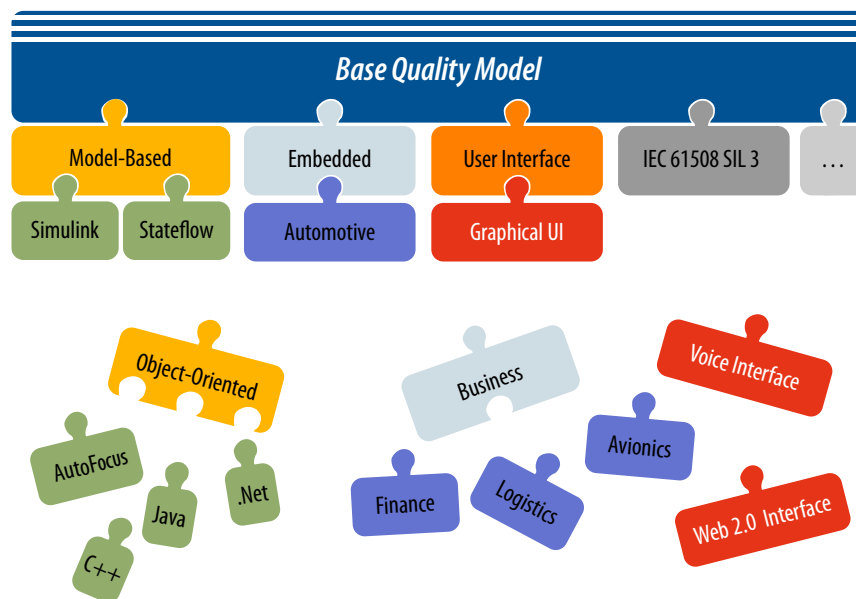


Figure 8.2: Quality Model Modularization

This modularization needs to be explicitly supported by the underlying quality metamodel. In particular, the quality metamodel must provide a means that allows quality models to be extended by other models. Initial experiments showed that the quality metamodel presented in this thesis is, in principal, well-suited for modularization as the entities and activities trees can be augmented by model elements stored in other models. The connection between the models is provided through the unique identifiability of all model elements. However, it needs to be resolved if such an *openness* for extensions is really desired or if dedicated *points of extension* need to be introduced. Moreover, certain extensions would require an explicit notion of *refinement*. For example, one would want to include the generic fact [Identifier | CONFORMANCE] that describes the format of identifiers in the base model and concretize it for specific programming or modeling languages.

Project »QuaMoCo« Regarding the realization of the challenges outlined above the author of this thesis is in a fortunate situation: Together with other leading research institutions and multiple key players of the German software industry, we participate in the BMBF¹-sponsored research project *QuaMoCo*². QuaMoCo's goal is the development of standardized quality model that can be applied in practice to objectively define, assess and continuously control software product quality. QuaMoCo, hence, addresses, among others, all the challenges discussed in this section. We are confident, that the excellent consortium of software quality theoreticians and practitioners that participate in QuaMoCo will be able to successfully tackle these challenges over the course of the three-year project!

¹German Federal Ministry of Education and Research

²More information on the project can be found at <http://www.quamoco.de>.

Bibliography

- [1] H. Abdul-Rahman, P. Thompson, and I. Whyte. Capturing the cost of non-conformance on construction sites: An application of the quality cost matrix. *Int. J. Qual. Reliab. Manage.*, 13(1):48–60, 1996.
- [2] A. Abran, R. Al Qutaish, M.-M. Desharnais, and N. Habra. An information model for software quality measurement with ISO standards. In *Proc. of the International Conference on Software Development*, pages 337–352, 2005.
- [3] A. Abran, J. W. Moore, P. Bourque, and R. Dupuis, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE CS, 2004.
- [4] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: An effective verification process. *IEEE Softw.*, 6(3):31–36, 1989.
- [5] H. Al-Kilidar, K. Cox, and B. Kitchenham. The use and usefulness of the ISO/IEC 9126 quality standard. In *Proc. of the International Symposium on Empirical Software Engineering (ISESE)*, 2005.
- [6] T. S. Andre, H. R. Hartson, S. M. Belz, and F. A. McCreary. The user action framework: A reliable foundation for usability engineering support tools. *Int. J. Hum.-Comput. Stud.*, 54(1):107–136, 2001.
- [7] R. S. Arnold and D. A. Parker. The dimensions of healthy maintenance. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 10–27. IEEE CS Press, 1982.
- [8] D. Ash, J. Alderete, P. W. Oman, and B. Lowther. Using software maintainability models to track code health. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 154–160. IEEE CS Press, 1994.
- [9] J. Bacon. *Concurrent Systems: Operating Systems, Database and Distributed Systems: An Integrated Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.
- [10] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.
- [11] R. D. Banker, S. M. Datar, and C. F. Kemerer. Factors affecting software maintenance productivity. In *Proc. of the International Conference on Information Systems*, 1987.
- [12] R. D. Banker, S. M. Datar, and C. F. Kemerer. A model to evaluate variables impacting the productivity of software maintenance projects. *Manage. Sci.*, 37(1):1–18, 1991.
- [13] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, 1993.
- [14] R. D. Banker and S. A. Slaughter. A field study of scale economies in software maintenance. *Manage. Sci.*, 43(12):1709–1725, 1997.

- [15] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.
- [16] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett. Understanding and predicting the process of software maintenance release. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 464–474. IEEE CS Press, 1996.
- [17] V. Basili, P. Donzelli, and S. Asgari. A unified model of dependability: Capturing dependability in context. *IEEE Softw.*, 21(6):19–25, 2004.
- [18] V. Basili and H. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Trans. Software Eng.*, 14(6):758–773, 1998.
- [19] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.
- [20] V. R. Basili, G. Caldiera, and D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 1994.
- [21] V. R. Basili and D. H. Hutchens. An empirical study of a syntactic complexity family. *IEEE Trans. Software Eng.*, 9(6):664–672, 1983.
- [22] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Software Eng.*, 13(12):1278–1296, 1987.
- [23] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 368. IEEE CS Press, 1998.
- [24] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.
- [25] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Syst. J.*, 15(3):225–252, 1976.
- [26] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Software Eng.*, 33(9):577–591, 2007.
- [27] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *Proc. of the Conference on The Future of Software Engineering (ICSE)*, pages 73–87. ACM, 2000.
- [28] M. Bennis and J.-P. Richter. Architecture of a generic software control centre. In *Proc. of Workshop on Measurement-based Cockpits for Distributed Software and Systems Engineering Projects (SOFTPIT)*, 2007.
- [29] G. M. Berns. Assessing software maintainability. *Commun. ACM*, 27(1):14–23, 1984.
- [30] N. Bevan. Measuring usability as quality of use. *Software Qual. J.*, 5(2):115–130, 1995.
- [31] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.
- [32] T. J. Biggerstaff, B. G. Mitbender, and D. Webster. The concept assignment problem in program understanding. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 482–498. IEEE CS Press, 1993.

-
- [33] W. R. Bischofberger, J. Kühl, and S. Löffler. Sotograph - a pragmatic approach to source code architecture conformance checking. In *Proc. of the European Workshop on Software Architecture (EWSA)*, pages 1–9. Springer, 2004.
- [34] J. Boehm. A new standard for quality requirements. *IEEE Softw.*, 25(2):57–63, 2008.
- [35] J. Boehm, S. Depanfilis, B. Kitchenham, and A. Pasquini. A method for software quality planning, control, and evaluation. *IEEE Softw.*, 16(2):69–77, 1999.
- [36] B. Boehm, L. Huang, A. Jain, and R. Madachy. The ROI of software dependability: The iDAVE model. *IEEE Softw.*, 21(3):54–61, 2004.
- [37] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
- [38] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.
- [39] E. G. Boring. Intelligence as the tests test it. *New Republic*, pages 35–37, June 1923.
- [40] L. Briand, K. E. Emam, and S. Morasca. On the application of measurement theory in software engineering. *Empirical Software Engineering*, 1(1):61–88, Jan. 1996.
- [41] L. C. Briand, J. Wüst, S. V. Ikonovskii, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 345–354. IEEE CS Press, 1999.
- [42] M. Broy, F. Deissenboeck, and M. Pizka. A holistic approach to software quality at work. In *Proc. of the World Congress for Software Quality (WCSQ)*, 2005.
- [43] M. Broy, F. Deissenboeck, and M. Pizka. Demystifying maintainability. In *Proc. of the Workshop on Software Quality (WOSQ)*. ACM Press, 2006.
- [44] T. Bruckhaus, N. H. Madhavji, I. Janssen, and J. Henshaw. The impact of tools on software productivity. *IEEE Softw.*, 13(5):29–38, 1996.
- [45] B. Brykczynski. A survey of software inspection checklists. *SIGSOFT Softw. Eng. Notes*, 24(1):82, 1999.
- [46] D. W. Bucher. Maintenance of the computer sciences teleprocessing system. In *Proc. of the International Conference on Reliable Software*, pages 260–266. ACM, 1975.
- [47] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.
- [48] R. P. Buse and W. R. Weimer. A metric for software readability. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 121–130. ACM, 2008.
- [49] T. Calvo, G. Mayor, and R. Mesiar, editors. *Aggregation Operators: New Trends and Applications*. Physica-Verlag, 2007.
- [50] E. G. Carmines and R. A. Zeller. *Reliability and Validity Assessment*. SAGE, 1979.

- [51] J. P. Cavano and J. A. McCall. A framework for the measurement of software quality. In *Proc. of the Software Quality Assurance Workshop on Functional and Performance Issues*, pages 133–139, 1978.
- [52] T. Chan. Beyond productivity in software maintenance: factors affecting leadtime in servicing users' requests. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 2000.
- [53] N. Chapin. Software maintenance life cycle. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 1988.
- [54] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *J. Softw. Maint. Evol. Res. Pr.*, 13(1):3–30, 2001.
- [55] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *Proc. of the International Workshop on Program Comprehension (IWPC)*, page 241. IEEE CS Press, 2000.
- [56] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
- [57] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.
- [58] A. Cimitile, A. D. Lucia, G. A. D. Lucca, and A. R. Fasolino. Identifying objects in legacy systems. In *Proc. of the International Workshop on Program Comprehension (IWPC)*, page 138. IEEE CS Press, 1997.
- [59] E. K. B. Clark, J. A. Forbes, E. R. Baker, and D. W. Hutcheson. Mission-critical and mission-support software: A preliminary maintenance characterization. Technical report, US DOD, 1999.
- [60] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, page 69. IEEE CS Press, 1998.
- [61] D. Coleman, D. Ash, B. Lowther, and P. W. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.
- [62] D. Coleman, B. Lowther, and P. Oman. The application of software maintainability models in industrial software systems. *J. Syst. Softw.*, 29(1):3–16, 1995.
- [63] M. L. Cook. Software metrics: An introduction and annotated bibliography. *SIGSOFT Softw. Eng. Notes*, 7(2):41–60, 1982.
- [64] R. Cooper and R. S. Kaplan. Measure costs right: Make the right decisions. *Harvard Bus. Rev.*, 67(Sept.-Oct.):96–103, 1988.
- [65] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Syst. J.*, 28(2):294–306, 1989.
- [66] M.-A. Cote, W. Suryn, , and E. Georgiadou. In search for a widely applicable and accepted software quality model for software quality engineering. *Software Qual. J.*, 15(4):401–416, 2007.
- [67] P. B. Crosby. *Quality Without Tears: The Art of Hassle-Free Management*. McGraw-Hill, 1995.

- [68] J. A. Cruz-Lemus, M. Genero, M. E. Manso, and M. Piattini. Evaluating the effect of composite states on the understandability of UML statechart diagrams. In *Proc. of the International Conference on Model Driven Engineering of Languages and Systems*. Springer, 2005.
- [69] B. Curtis. Measurement and experimentation in software engineering. *Proc. of the IEEE*, 68(9):1144–1157, 1980.
- [70] B. Curtis. Substantiating programmer variability. *Proc. IEEE*, 69(7):846, 1981.
- [71] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Trans. Software Eng.*, 5(2):96–104, 1979.
- [72] S. Danicic, C. Fox, M. Harman, and R. Hierons. ConSIT: A conditioned program slicer. In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 216. IEEE CS Press, 2000.
- [73] C. Debou, A. Kuntzmann-Combelles, and A. Rowe. A quantitative approach to software process management. In *Proc. of International Symposium on Software Metrics*, 1994.
- [74] F. Deißböck, B. Hummel, and E. Jürgens. CONQAT - Ein Toolkit zur kontinuierlichen Qualitätsbewertung. In *Tagungsband der Software-Engineering-Konferenz*, page 55, 2008.
- [75] F. Deißböck and T. Seifert. Kontinuierliche Qualitätsüberwachung mit CONQAT. In *Tagungsband der Informatik 2006*, pages 118–125. GI, 2006.
- [76] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 603–612. ACM, 2008.
- [77] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Softw.*, 25(5):60–67, 2008.
- [78] F. Deissenboeck and M. Pizka. Concise and consistent naming. In *Proc. of the International Workshop on Program Comprehension (IWPC)*, pages 97–106. IEEE CS Press, 2005.
- [79] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Qual. J.*, 14(3):261–282, September 2006.
- [80] F. Deissenboeck and M. Pizka. The economic impact of software process variations. In *Proc. of the International Conference on Software Process*. Springer, 2007.
- [81] F. Deissenboeck and M. Pizka. Probabilistic analysis of process economics. *Softw. Process Improve. Pract.*, 13(1):5–17, 2008.
- [82] F. Deissenboeck, M. Pizka, and T. Seifert. Tool support for continuous quality assessment. In *Proc. of the International Workshop on Software Technology and Engineering Practice*, volume 0, pages 127–136. IEEE CS Press, 2005.
- [83] F. Deissenboeck and D. Ratiu. A unified meta-model for concept-based reverse engineering. In *Proc. of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering (ATEM)*. Johannes Gutenberg-Universität Mainz, 2006.

- [84] F. Deissenboeck and S. Wagner. Kosten-basierte Klassifikation von Qualitätsanforderungen. In *Workshopband der Software-Engineering-Konferenz*. GI, 2007.
- [85] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An activity-based quality model for maintainability. In *Proc. of the International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2007.
- [86] T. DeMarco and T. Lister. *Peopleware: Productive projects and teams*. Dorset House Publishing Co., Inc., New York, NY, USA, 1999.
- [87] W. E. Deming. *Out of the Crisis*. MIT Press, 2000.
- [88] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.
- [89] M. T. Dishaw and D. M. Strong. Supporting software maintenance with software engineering tools: A computed task-technology fit analysis. *J. Syst. Softw.*, 44(2):107–120, 1998.
- [90] A. Dix, J. Finley, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice-Hall, 1998.
- [91] J. Doerr, D. Kerkow, T. Koenig, T. Olsson, and T. Suzuki. Non-functional requirements in industry – Three case studies adopting an experience-based NFR method. In *Proc. of the International Conference on Requirements Engineering*, pages 373–382, 2005.
- [92] R. G. Dromey. A model for software product quality. *IEEE Trans. Software Eng.*, 21(2):146–162, 1995.
- [93] R. G. Dromey. Cornering the chimera. *IEEE Softw.*, 13(1):33–43, 1996.
- [94] dSpace. Modeling guidelines for MATLAB/Simulink/Stateflow and TargetLink. Technical report, dSpace, 2006.
- [95] dSpace. Modeling guidelines for MATLAB/Simulink/Stateflow and TargetLink 2.0. Technical report, dSpace, 2007.
- [96] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: An extensible language-independent environment for reengineering object-oriented systems. In *Proc. of the International Symposium on Constructing Software Engineering Tools*, 2000.
- [97] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Software Eng.*, 27(1):1–12, 2001.
- [98] A. Epping and C. M. Lott. Does software design complexity affect maintenance effort? In *Proc. of the Software Engineering Workshop*, pages 297–313. NASA Goddard Space Flight Center, Greenbelt MD 20771, 1994.
- [99] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
- [100] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, 1976.
- [101] M. E. Fagan. Advances in software inspections. *IEEE Trans. Software Eng.*, 12(7):744–751, 1986.
- [102] A. Feigenbaum. Total quality control. *Harvard Bus. Rev.*, 34(6):93–101, 1956.

-
- [103] M. Feilkas, D. Ratiu, and E. Juergens. The loss of architectural knowledge during system evolution: An industrial case study. In *Proc. of the International Conference on Program Comprehension (ICPC)*. IEEE CS, 2009.
- [104] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Software Eng.*, 20(3):199–206, 1994.
- [105] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.
- [106] R. Fjeldstad and W. Hamlen. *Tutorial on Software Maintenance*, chapter Application Program Maintenance Study: Report to Our Respondents, pages 13–30. IEEE CS Press, 1983.
- [107] E. Folmer and J. Bosch. Architecting for usability: A survey. *J. Syst. Softw.*, 70:61–78, 2004.
- [108] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [109] B. Freimut, L. Briand, and F. Vollei. Determining inspection cost-effectiveness by combining project data and expert opinion. *IEEE Trans. Software Eng.*, 31(12):1074–1092, 2005. Senior Member-Lionel C. Briand.
- [110] L. Friendly. The design of distributed hyperlinked programming documentation. In *Proc. of the International Workshop on Hypermedia Design*, pages 151–173. Springer, 1995.
- [111] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [112] M. J. B. García and J. C. G. Alvarez. Maintainability as a key factor in maintenance productivity: a case study. In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 87. IEEE CS Press, 1996.
- [113] A. Gardener. Effiziente Wartung und Weiterentwicklung der Benutzeroberflächen betrieblicher Informationssysteme. Master's thesis, Technische Universität München, 2007.
- [114] D. A. Garvin. What does »Product Quality« really mean? *MIT Sloan Management Review*, 26(1):25–43, 1984.
- [115] V. R. Gibson and J. A. Senn. System structure and software maintenance performance. *Commun. ACM*, 32(3):347–358, 1989.
- [116] T. Gilb. *Software Metrics*. Winthrop Publishers, 1977.
- [117] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.
- [118] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Software Eng.*, 17(12):1284–1288, 1991.
- [119] R. L. Glass. Maintenance: Less is not more. *IEEE Softw.*, 15(4):67–68, 1998.
- [120] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2002.
- [121] A. L. Goel. Software reliability models: Assumptions, limitations, and applicability. *IEEE Trans. Software Eng.*, 11(12):1411–1423, 1985.

- [122] T. Goldschmidt, R. Reussner, and J. Winzen. A case study evaluation of maintainability and performance of persistency techniques. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 401–410. ACM, 2008.
- [123] R. R. Gonzalez. A unified metric of software complexity: measuring productivity, quality, and value. *J. Syst. Softw.*, 29(1):17–37, 1995.
- [124] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 2000.
- [125] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall, 1987.
- [126] J. C. Granja-Alvarez and M. J. Barranco-Garcia. A method for estimating maintenance cost in a software project: A case study. *J. Softw. Maint. Evol. Res. Pr.*, 9(3):161–175, 1997.
- [127] C. M. Grinstead and J. L. Snell. *Introduction to Probability*. AMS, 2003.
- [128] J. Grudin. The case against user interface consistency. *Commun. ACM*, 32(10):1164–1173, 1989.
- [129] F. M. Gryna. *Juran's Quality Control Handbook*, chapter Quality Costs, pages 4.1–4.30. McGraw-Hill, 1988.
- [130] M. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, NY, USA, 1977.
- [131] S. J. Hanson and R. R. Rosinski. Programmer perceptions of productivity and programming tools. *Commun. ACM*, 28(2):180–189, 1985.
- [132] H. J. Harrington. *Business Process Improvement. The Breakthrough Strategy for Total Quality, Productivity, and Competitiveness*. McGraw-Hill, New York (NY), 1991.
- [133] C. S. Hartzman and C. F. Austin. Maintenance productivity: Observations based on an experience in a large system environment. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering*, pages 138–170. IBM Press, 1993.
- [134] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proc. of the International Conference on the Quality of Information and Communications Technology*, pages 30–39, 2007.
- [135] M. Henricson and E. Nyquist. Programming in C++: Rules and recommendations. Technical report, Ellemtel Telecommunication Systems Laboratories, 1992.
- [136] J. Highsmith. *Agile software development ecosystems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [137] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
- [138] L. Huang and B. Boehm. How much software quality investment is enough: A value-based approach. *IEEE Softw.*, 23(5):88–95, 2006.
- [139] A. Hunt and D. Thomas. *The pragmatic programmer: From journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [140] A. Hunt and D. Thomas. Zero-tolerance construction. *IEEE Softw.*, 19(5):100–102, 2002.

-
- [141] G. Hwang and E. Aspinwall. Quality cost models and their application: A review. *Total Quality Management*, 7(3):267–282, 1996.
- [142] L. Hyatt and L. Rosenberg. A software quality model and metrics for identifying project risks and assessing software quality. In *Proc. of the Annual Software Technology Conference*, pages 345–354, 1996.
- [143] IEEE. Standard 610.12 – Glossary of software engineering terminology. Standard, IEEE, 1990.
- [144] IEEE. Industry implementation of international standard ISO/IEC 12207 : 1995 – software life cycle processes. Standard, IEEE, 1996.
- [145] IEEE. 1219 standard 1219 for software maintenance. Standard, IEEE, 1998.
- [146] IEEE. Standard 730 for software quality assurance plans. Standard, IEEE, 2002.
- [147] ISO. Standard 8402 for quality management and quality assurance – Vocabulary. Standard, ISO, 1994.
- [148] ISO. Standard 9000 for quality management systems – Fundamentals and vocabulary. Standard, ISO, 1995.
- [149] ISO. Standard 9241-11 for ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability. Standard, ISO, 1998.
- [150] ISO. Standard 14598 for information technology – Software product evaluation. Standard, ISO, 1999.
- [151] ISO. Standard 9126 for software engineering – Product quality. Standard, ISO, 2001.
- [152] ISO. Standard 9126 for software engineering – Product quality – Part 1: Quality model. Standard, ISO, 2001.
- [153] ISO. Standard 15005 for road vehicles – Ergonomic aspects of transport information and control systems – Dialogue management principles and compliance procedures. Standard, ISO, 2002.
- [154] ISO. Standard 9126 for software engineering – Product quality – Part 2: External metrics. Standard, ISO, 2003.
- [155] ISO. Standard 9126 for software engineering – Product quality – Part 3: Internal metrics. Standard, ISO, 2003.
- [156] ISO. Standard 9126 for software engineering – Product quality – Part 4: Quality in use metrics. Standard, ISO, 2004.
- [157] ISO. Standard 9241-110 for ergonomics of human-system interaction – Part 110: Dialogue principles. Standard, ISO, 2006.
- [158] ISO/IEC. Standard 15504 for information technology – Process assessment. Standard, ISO/IEC, 2004.
- [159] ISO/IEC. Standard 14764 for software maintenance. Standard, ISO/IEC, 2006.

- [160] M. Johnson. The development of measures of the cost quality for an engineering unit. *Int. J. Qual. Reliab. Manage.*, 12(2):86–100, 1995.
- [161] P. M. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa, and T. Yamashita. Improving software development management through software project telemetry. *IEEE Softw.*, 22(4):76–85, 2005.
- [162] C. Jones. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., New York, NY, USA, 1991.
- [163] C. Jones. Activity-based software costing. *Computer*, 29(5):103–104, 1996.
- [164] C. Jones. *Software assessments, benchmarks, and best practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [165] E. Juergens, F. Deissenboeck, and B. Hummel. Clone detection beyond copy&paste (Position Paper). In *Proc. 3rd Intl. Workshop on Software Clones*. IEEE CS, 2009.
- [166] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective – A workbench for clone detection research (Tool Demo). In *ICSE '09: Proc. of the 31th international conference on Software engineering*. IEEE CS, 2009.
- [167] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE CS, 2009.
- [168] E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In *Workshopband Software-Engineering-Konferenz, Lecture Notes in Informatics*. Gesellschaft für Informatik, 2008.
- [169] J. M. Juran. *Juran's Quality Control Handbook*. McGraw-Hill, 1988.
- [170] D. Kafura and G. R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Trans. Software Eng.*, 13(3):335–343, 1987.
- [171] M. Kajko-Mattsson. Preventive maintenance! Do we know what it is? In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 12. IEEE CS, 2000.
- [172] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.
- [173] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know? In *Proc. of the International Software Metrics Symposium*. IEEE CS Press, 2004.
- [174] K. Katheder. Studie zu Software-Wartung. Bachelor's thesis, Technische Universität München, 2003.
- [175] K. Keizer, S. Lindenberg, and L. Steg. The spreading of disorder. *Science*, 322(5905):1681–1685, 2008.
- [176] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Software Eng.*, 25(4):493–509, 1999.
- [177] B. Kitchenham, S. Linkman, A. Pasquini, and V. Nanni. The SQUID approach to defining a quality model. *Software Qual. J.*, 6(3):211–233, Sept. 1997.

- [178] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Softw.*, 13(1):12–21, 1996.
- [179] B. Kitchenham, S. L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Trans. Software Eng.*, 21(12):929–944, 1995.
- [180] B. A. Kitchenham. Software quality assurance. *Microprocess. Microsyst.*, 13(6):373–381, 1989.
- [181] B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *J. Softw. Maint. Evol. Res. Pr.*, 11(6):365–389, 1999.
- [182] S. T. Knox. Modeling the cost of software quality. *Digital Tech. J.*, 5(4):9–17, 1993.
- [183] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proc., 2007.
- [184] H. Krasner. Using the cost of quality approach for software. *Crosstalk*, 11:6–11, 1998.
- [185] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 1997.
- [186] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution – The nineties view. In *Proc. of the International Symposium on Software Metrics*. IEEE CS Press, 1997.
- [187] T. Lethbridge and J. Singer. Understanding software maintenance tools: Some empirical research. In *Proc. of the Workshop on Empirical Studies of Software Maintenance (WESS)*. IEEE CS Press, 1997.
- [188] H. Li, M. Ross, G. King, G. Staples, and M. Jing. Quality approaches in a large software house. *Software Quality Control*, 8(1):21–35, 1999.
- [189] X. Li and C. Prasad. Effectively teaching coding standards in programming. In *Proc. of the Conference on Information Technology Education*, pages 239–244. ACM, 2005.
- [190] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison Wesley, Reading, 1980.
- [191] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.
- [192] M. Lindvall, S. Komi-Sirviö, P. Costa, and C. Seaman. Embedded software maintenance. A DACS state-of-the-art report, Fraunhofer Center for Experimental Software Engineering, 2003.
- [193] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, 1987.
- [194] G. A. D. Lucca, A. R. Fasolino, P. Tramontana, and C. A. Visaggio. Towards the definition of a maintainability model for web applications. In *Proc. of the Conference on Software Maintenance and Reengineering (CSMR)*, page 279. IEEE CS Press, 2004.

- [195] MAAB. Controller style guidelines for production intent using Matlab, Simulink and Stateflow. Technical report, MAAB, 2001.
- [196] MAAB. Controller style guidelines for production intent using Matlab, Simulink and Stateflow 2.0. Technical report, MAAB, 2007.
- [197] W. Mandeville. Software costs of quality. *IEEE J. Sel. Areas Commun.*, 8(2):315–318, 1990.
- [198] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.
- [199] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *International Conference on Software Maintenance (ICSM)*, pages 77–80, 2005.
- [200] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM '04: Proc. of the 20th IEEE International Conference on Software Maintenance*, pages 350–359. IEEE CS, 2004.
- [201] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pages 192–201. IEEE CS Press, 2004.
- [202] The MathWorks. *Simulink Reference*, 2006.
- [203] R. G. Mays. Practical aspects of the defect prevention process. In T. Gilb, editor, *Software Inspection*, pages 336–360. Addison-Wesley, 1993.
- [204] T. J. McCabe. A complexity measure. In *Proc. of the International Conference on Software Engineering (ICSE)*, page 407. IEEE CS Press, 1976.
- [205] J. McCall and G. Walters. *Factors in Software Quality*. The National Technical Information Service (NTIS), Springfield, VA, USA, 1977.
- [206] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.
- [207] M. Metcalf. Fortran 77 coding conventions. *SIGPLAN Fortran Forum*, 2(4):10–15, 1983.
- [208] S. Microsystems. Code conventions for the Java programming language. Technical report, Sun Microsystems, 1999.
- [209] MISRA. Guidelines for the use of the C language in vehicle based software. Technical report, MISRA, 1998.
- [210] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Software Eng.*, 18(6):483–497, 1992.
- [211] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 452–461. ACM, 2006.
- [212] J. Nielsen. *Usability Engineering*. AP Professional, 1993.
- [213] F. Niessink and H. van Vliet. Two case studies in measuring software maintenance effort. In *Proc. of the International Conference on Software Maintenance (ICSM 1998)*, 1998.

- [214] D. A. Norman. Cognitive engineering. In D. A. Norman and S. W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 31–61. Lawrence Erlbaum Associates, 1986.
- [215] J. T. Nosek and P. Palvia. Software maintenance management: changes in the last decade. *Journal of Software Maintenance*, 2(3):157–174, 1990.
- [216] C. Oezbek and L. Prechelt. JTourBus: Simplifying program understanding by documentation that provides tours through the source code. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 2007.
- [217] T. Okubo and H. Tanaka. Secure software development through coding conventions and frameworks. In *Proc. of the Second International Conference on Availability, Reliability and Security*, pages 1042–1051. IEEE CS, 2007.
- [218] P. Oman and J. Hagemester. Metrics for assessing a software system’s maintainability. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 337–344, 1992.
- [219] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *Commun. ACM*, 33(5):506–520, 1990.
- [220] P. W. Oman, C. R. Cook, and M. Nanja. Effects of programming experience in debugging semantic errors. *J. Syst. Softw.*, 9(3):197–207, 1989.
- [221] M. L. Ouchi. Software maintenance documentation. In *Proc. of the International Conference on Systems Documentation*, pages 18–23. ACM Press, 1985.
- [222] G. Parikh and N. Zvegintzov, editors. *Tutorial on Software Maintenance*. IEEE CS Press, 1983.
- [223] R. E. Park, W. B. Goethert, and W. A. Florac. Goal-driven software measurement – a guidebook. Handbook CMU/SEI-96-HB-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA, August 1996.
- [224] D. L. Parnas. Software aging. In *Proc. of the International Conference on Software Engineering (ICSE)*, pages 279–287. IEEE CS Press, 1994.
- [225] M. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [226] T. Pearse and P. Oman. Maintainability measurements on industrial source code maintenance activities. In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 295. IEEE CS, 1995.
- [227] D. E. Peercy. A software maintainability evaluation methodology. *IEEE Trans. Software Eng.*, 7(4):343–351, 1981.
- [228] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Software. Eng. Meth.*, 10(3):308–337, 2001.
- [229] S. L. Pfleeger, R. Jeffery, B. Curtis, and B. Kitchenham. Status report on software measurement. *IEEE Softw.*, 14(2):33–43, 1997.

- [230] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE, 2009.
- [231] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.
- [232] M. Pizka. Software-Wartung. Lecture notes, TU München, 2004.
- [233] M. Pizka. Straightening spaghetti-code with refactoring? In *Proc. of the International Conference on Software Engineering Research and Practice*, pages 846–852. CSREA Press, June 2004.
- [234] R. Plösch, H. Gruber, A. Hentschel, C. Körner, G. Pomberger, S. Schiffer, M. Saft, and S. Storck. The EMISQ method and its tool support-expert-based evaluation of internal software quality. *Innovations in Systems and Software Engineering*, 4(1):3–15, Apr. 2008.
- [235] R. Plösch, H. Gruber, G. Pomberger, M. Saft, and S. Schiffer. Tool support for expert-centred code assessments. In *Proc. of the International Conference on Software Testing, Verification, and Validation (ICST)*, pages 258–267. IEEE CS, 2008.
- [236] I. Podnar and B. Mikac. Software maintenance process analysis using discrete-event simulation. In *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE CS, 2001.
- [237] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [238] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Softw.*, 7(2):46–54, 1990.
- [239] V. Rajlich. Role of concepts in software evolution. In *Proc. of the International Workshop on Principles of Software Evolution*, pages 75–78. ACM Press, 2001.
- [240] V. Rajlich and K. H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000.
- [241] V. Rajlich and P. Gosavi. A case study of unanticipated incremental change. In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 442. IEEE CS Press, 2002.
- [242] S. Ramanujan, R. W. Scamell, and J. R. Shah. An experimental investigation of the impact of individual, program, and organizational characteristics on software maintenance effort. *J. Syst. Softw.*, 54(2):137–157, 2000.
- [243] D. Ratiu and F. Deissenboeck. How programs represent reality (and how they don't). In *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, 2006.
- [244] D. Ratiu and F. Deissenboeck. From reality to programs and (not quite) back again. In *Proc. of the IEEE International Conference on Program Comprehension (ICPC)*. IEEE CS Press, 2007.
- [245] C. A. Reeves and D. A. Bednar. Defining quality: Alternatives and implications. *The Academy of Management Review*, 19(3):419–445, 1994.
- [246] H. D. Rombach. A controlled experiment on the impact of software structure on maintainability. *IEEE Trans. Software Eng.*, 13(3):344–354, 1987.

- [247] H. D. Rombach. Design measurement: Some lessons learned. *IEEE Softw.*, 7(2):17–25, 1990.
- [248] H. D. Rombach, B. T. Ulery, and J. D. Valett. Toward full life cycle control: Adding maintenance measurement to the SEL. *J. Syst. Softw.*, 18(2):125–138, 1992.
- [249] S. Roski. DoD-STD-2167 default Ada design and coding standard. *Ada Lett.*, VI(5):34–44, 1986.
- [250] A. J. Rostkowycz, V. Rajlich, and A. Marcus. A case study on the long-term effects of software redocumentation. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 92–101. IEEE CS, 2004.
- [251] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen’s University at Kingston, 2007.
- [252] W. W. Royce. Managing the development of large software systems. In *IEEE WESCON*, pages 1–9. IEEE CS Press, 1970.
- [253] R. J. Rubey and R. D. Hartwick. Quantitative measurement of program quality. In *Proc. of the National Conference*, pages 671–677. ACM Press, 1968.
- [254] G. Ruhe and M. O. Saliu. The art and science of software release planning. *IEEE Softw.*, 22(6):47–53, 2005.
- [255] H. Sackman, W. J. Erikson, and E. E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM*, 11(1):3–11, 1968.
- [256] A. Saltelli. A quantitative model-independent method for global sensitivity analysis of model output. *Technometrics*, 41(1):39–56, 1999.
- [257] A. Saltelli, editor. *Sensitivity Analysis*. John Wiley & Sons, 2000.
- [258] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Trans. Software Eng.*, 18(5):410–422, 1992.
- [259] P. Schuh. Recovery, redemption, and extreme programming. *IEEE Softw.*, 18(6):34–41, 2001.
- [260] A. Seffah, M. Donyaee, R. B. Kline, and H. K. Padda. Usability measurement and metrics: A consolidated model. *Software Quality Control*, 14(2):159–178, 2006.
- [261] B. Shackel and S. Richardson, editors. *Human Factors for Informatics Usability*. Cambridge University Press, 1991.
- [262] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, 3(2):30–36, 1988.
- [263] W. A. Shewhart. *Statistical Method from the Viewpoint of Quality Control*. Dover Publications, 1986.
- [264] J. S. Shirabad, T. C. Lethbridge, and S. Matwin. Supporting maintenance of legacy software with data mining techniques. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Eesearch*, page 11. IBM Press, 2000.
- [265] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.

- [266] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 3 edition, 1998.
- [267] P. Simmons. Quality outcomes: Determining business value. *IEEE Softw.*, 13(1):25–32, 1996.
- [268] H. Siy and L. Votta. Does the modern code inspection have value? In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 281. IEEE CS Press, 2001.
- [269] S. A. Slaughter, D. E. Harter, and M. S. Krishnan. Evaluating the cost of software quality. *Commun. ACM*, 41(8):67–73, 1998.
- [270] H. Sneed. Estimating the costs of software maintenance tasks. In *Proc. of the International Conference on Software Maintenance (ICSM)*, 1995.
- [271] H. Sneed. A cost model for software maintenance & evolution. In *International Conference on Software Maintenance (ICSM)*. IEEE CS Press, 2004.
- [272] H. M. Sneed. Planning the reengineering of legacy systems. *IEEE Softw.*, 12(1):24–34, 1995.
- [273] H. M. Sneed. Measuring the performance of a software maintenance department. In *Conference on Software Maintenance and Reengineering (CSMR)*, pages 119–127. IEEE CS Press, 1997.
- [274] I. Sommerville. *Software Engineering*. Pearson Addison Wesley, 7th edition, 2004.
- [275] R. Stallmann. GNU coding standards. Technical report, Free Software Foundation (FSF), Sept. 2001.
- [276] G. E. Stark. Measurements for managing software maintenance. In *Proc. of the International Conference on Software Maintenance (ICSM)*, page 152. IEEE CS Press, 1996.
- [277] S. S. Stevens. On the theory of scales of measurement. *Science*, 103(2684):677–680, 1946.
- [278] E. Stone-Romero, D. Stone, and D. Grewal. Development of a multidimensional measure of perceived product quality. *J. Qual. Manage.*, 2(2):87–111, 1997.
- [279] A. Sutcliffe. *User-Centered Requirements Engineering: Theory and Practice*. Springer-Verlag, 2002.
- [280] L. Tahvildari, K. Kontogiannis, and J. Mylopoulos. Requirements-driven software re-engineering framework. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, page 71. IEEE CS, 2001.
- [281] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.
- [282] W. Trochim and J. P. Donnelly. *The Research Methods Knowledge Base*. Atomic Dog, 3rd edition, 2006.
- [283] US DOD (AFOTEC). Software maintainability evaluation guide. AFOTEC Pamphlet 99-102, HQ Air Force Operational Test and Evaluation Center, 1996.
- [284] US DOD (STSC). Software reengineering assessment handbook v3.0. Technical report, US DOD, 1997.

- [285] J. van Gorp and J. Bosch. Design erosion: Problems and causes. *J. Syst. Softw.*, 61(2):105–119, 2002.
- [286] M. van Welie, G. C. van der Veer, and A. Eliëns. Breaking down usability. In *Proc. of the International Conference on Human-Computer Interaction*, pages 613–620. IOS Press, 1999.
- [287] J. Voas. Can clean pipes produce dirty water? *IEEE Softw.*, 14(4):93–95, 1997.
- [288] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.
- [289] S. Wagner. A literature survey of the quality economics of defect-detection techniques. In *Proc. of the International Symposium on Empirical Software Engineering (ISESE)*, pages 194–203. ACM, 2006.
- [290] S. Wagner. A bayesian network approach to assess and predict software quality using activity-based quality models. In *Proc. of the International Conference on Predictor Models in Software Engineering (PROMISE)*. ACM Press, 2009.
- [291] S. Wagner and F. Deissenboeck. An integrated approach to quality modelling. In *Proc. of the Workshop on Software Quality*. IEEE CS Press, 2007.
- [292] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for Java. In *Proc. of the International Conference on Software Testing, Verification and Validation (ICST)*. IEEE CS Press, 2008.
- [293] S. Wagner, F. Deissenboeck, M. Feilkas, and E. Jürgens. Software-Qualitätsmodelle in der Praxis: Erfahrungen mit aktivitätenbasierten Modellen. In *Workshopband Software-Qualitätsmodellierung und -bewertung (SQMB)*. Technische Universität München, 2008.
- [294] S. Wagner, F. Deissenboeck, and S. Winter. Erfassung, Strukturierung und Überprüfung von Qualitätsanforderungen durch aktivitätenbasierte Qualitätsmodelle. In *Workshopband Software-Engineering-Konferenz*, Lecture Notes in Informatics. Gesellschaft für Informatik, 2008.
- [295] S. Wagner, F. Deissenboeck, and S. Winter. Managing quality requirements using activity-based quality models. In *Proc. of the International Workshop on Software Quality (WoSQ)*, pages 29–34. ACM, 2008.
- [296] S. Wagner and S. Islam. Modellierung von Software-Security mit aktivitätenbasierten Qualitätsmodellen. In *Workshopband Software-Qualitätsmodellierung und -bewertung (SQMB)*, 2009.
- [297] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhota. Problems creating task-relevant clone detection reference data. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, page 285. IEEE CS, 2003.
- [298] M. P. Ware, F. G. Wilkie, and M. Shapcott. The application of product measures in directing software maintenance activity. *J. Softw. Maint. Evol. Res. Pr.*, 19(2):133–154, 2007.
- [299] G. M. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold Co., 1971.
- [300] G. Wiederhold. What is your software worth? *Commun. ACM*, 49(9):65–75, 2006.

- [301] J. J. V. Wijk and H. van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *Proc. of the Symposium on Information Visualization*, page 73. IEEE CS, 1999.
- [302] A. Williams, A. van der Wiele, and B. Dale. Quality costing: A management review. *Int. J. Manage. Rev.*, 1(4):441–460, 1999.
- [303] J. Q. Wilson and G. L. Kelling. Broken windows. *The Atlantic Monthly*, 249(3):29–38, 1982.
- [304] S. Winter, S. Wagner, and F. Deissenboeck. A comprehensive model of usability. In *Proc. of Engineering Interactive Systems*. Springer, 2007.
- [305] D. Yeh and J.-H. Jeng. An empirical study of the influence of departmentalization and organizational position on software maintenance. *J. Softw. Maint. Evol. Res. Pr.*, 14(1):65–82, 2002.
- [306] H. Zuse. *A Framework of Software Measurement*, chapter History of Software Measurement. Walter de Gruyter & Co., 1997.