Assessing Software Quality Attributes with Source Code Metrics

Andreas Jetter
of Birmensdorf, Switzerland (01-700-897)

supervised by
Harald Gall
Martin Pinzger, Patrick Knab
Assessing Software Quality Attributes with Source Code Metrics

Andreas Jetter

University of Zurich
Department of Informatics

Software evolution & architecture lab
Diploma Thesis

Author: Andreas Jetter, andreas.jetter@gmx.ch
Project period: 3.4.2006 - 3.10.2006

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich
I would like to thank Prof. Harald Gall for giving me the opportunity to write this thesis and for providing a superb infrastructure.

Foremost, I would like to thank Martin Pinzger and Patrick Knab for their help during the last six months. Also special thanks to Co for her patience during the hairy days and my fellow students Michi and Roman for power infusions during downtimes. Last but not least I would like to thank Marcel for proof-reading my work and my parents for their never ending trust.
This thesis is about quality assessment of software systems by using source code metrics.

We define four dimensions and relate them to a number of popular quality models, i.e., the models of McCall, Boehm, ISO 9126, Dromey and Bansiya. We also relate source code metric based quality models (SMQM) to these dimensions and show that the usefulness of SMQM is limited to an architectural view. But from this point of view, it is an expressive tool to assess software.

We discuss several aspects of source code measuring. The objective and subjective viewpoint are contrasted whereas the former is more an engineering approach and the latter is more an artistic one. The danger of use and abuse of metrics is also highlighted as well as the problem of validating and combining source code metrics.

We developed a SMQM inspired by the quality model for object-oriented design (QMOOD) introduced by Bansiya. The quality assessor tool we implemented is able to measure Java source code measures and summarize them into abstract quality attributes. These high level attributes can be visualized in a plot to trace the evolution of the design quality over time.

In a case study we use the quality assessor tool to analyze the open source project “Azureus”. “Azureus” is a medium size bit torrent client. We consider three years, during which “Azureus” had grown from 22'000 to 222'000 lines of code. We measure 19 releases and analyze them by comparing the evolution of the design metrics with the changelog data from the developer’s website. This way we are able to show that there exists a recognizable correlation between these two.
Zusammenfassung

Diese Diplomarbeit behandelt das Thema der Qualitätsbeurteilung von Softwaresystemen mittels Quellcodemetriken.


Inspiriert von Bansiyas Qualitätsmodell für objekt-orientiertes Design (QMOOD) haben wir ein SMQM implementiert. Wir entwickelten ein Quality Assessor Tool, welches Java Quellcode vermessen und zu abstrakten Qualitätsmerkmalen kombinieren kann. Die berechneten Qualitätsmerkmale können dann in einem Graphen dargestellt und die Entwicklung der Designqualität über die Zeit verfolgt werden.

Contents

1 Introduction
   1.1 Contribution .................................................. 1
   1.2 Thesis Outline ............................................... 2

2 Dimensions of Quality Models
   2.1 Dimension: Perspective ....................................... 3
   2.2 Dimension: Quality and Time ................................ 4
   2.3 Dimension: Level of Quality Abstraction ...................... 5
   2.4 Dimension: Compareability of Quality ......................... 5
   2.5 Dimensions and Quality Assessing with SMQM ................. 6
      2.5.1 Viewpoint of the SMQM .................................... 7
      2.5.2 Time and SMQM ............................................ 7
      2.5.3 Abstraction and SMQM .................................... 7
      2.5.4 Comparability and SMQM .................................. 7
   2.6 Summary ..................................................... 8

3 Quality and Measurement
   3.1 Objectivity vs Subjectivity .................................... 9
   3.2 Validation of Measures ........................................ 10
   3.3 Use and Abuse of Measurement ................................ 10
   3.4 The Problem of Metric Combination ......................... 11
   3.5 Metrics in the Object-Oriented Paradigm .................... 11
   3.6 Summary ..................................................... 11

4 Quality Models
   4.1 McCall .................................................. 13
   4.2 Boehm .................................................. 14
   4.3 ISO 9126 ................................................ 15
   4.4 Dromey ................................................ 16
   4.5 Bansiya ................................................ 17
   4.6 Prominent Models in Context with SMQM .................... 18

5 The Model
   5.1 The Four Levels ........................................... 21
   5.2 Design Metric → Design Property ............................ 22
   5.3 Design Property → Quality Attribute ......................... 24
   5.4 The Adapted Quality Model .................................. 24
      5.4.1 Metrics 1.3.6 ........................................... 25
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.2</td>
<td>Used Measurement</td>
<td>25</td>
</tr>
<tr>
<td>5.5</td>
<td>Basis of Comparability</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Quality Assessor Tool</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>Program Structure</td>
<td>29</td>
</tr>
<tr>
<td>6.2</td>
<td>Data Input</td>
<td>30</td>
</tr>
<tr>
<td>6.3</td>
<td>Model Visualization</td>
<td>31</td>
</tr>
<tr>
<td>6.4</td>
<td>Case Study</td>
<td>31</td>
</tr>
<tr>
<td>6.5</td>
<td>Results</td>
<td>32</td>
</tr>
<tr>
<td>6.6</td>
<td>Discussion</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion &amp; Future Work</td>
<td>37</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusion</td>
<td>37</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>38</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Source Code Metric Based Quality Modeling</td>
<td>38</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Quality Assessor Tool</td>
<td>38</td>
</tr>
<tr>
<td>A</td>
<td>Contents of CD-Rom</td>
<td>39</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Model of Viewpoints ................................................. 4
2.2 Level of Abstraction ................................................. 6
2.3 Scope of Quality and Source Code Metric ............................. 7
4.1 Top level characteristic tree ........................................ 15
4.2 Dromey Generic Model ............................................. 17
4.3 Structural Form to Design Carrying Properties ....................... 18
4.4 Quality models embedded ........................................... 19
5.1 Levels and links in QMOOD .......................................... 21
6.1 Core of the Attribute Calculation .................................... 30
6.2 Preference page of the Quality Assessor .............................. 30
6.3 How to show the model visualization ................................ 31
6.4 Table View of the Quality Assessor Tool ............................ 32
6.5 Graph View of the Quality Assessor Tool ............................ 33
6.6 Quality Assessor Tool Workflow ..................................... 33
6.7 Count of changes of the Azureus project ............................. 34
6.8 Analysis of the result Graph .......................................... 35

List of Tables

4.1 Quality-Carrying Properties of Software by Dromey ............... 17
5.1 Definition of Quality Attributes by Bansiva .......................... 22
5.2 Definition of Design Properties by Bansiva .......................... 23
5.3 Computation Formulas for Quality Attributes ....................... 24
5.4 Correlation between Quality Attributes and Design Properties .......... 25
5.5 Presentation of the used Metric ..................................... 26
Chapter 1

Introduction

During the last twenty years the field of software development boomed. With “boomed” we mean not only its market size but first of all in its products. Software vendors develop software for every industry field. Software is everywhere in our daily life. Over time the software products become more complex and universal. They speed up our life in private and business. But since we use software the number of “Bugs” in our life increases. “Bugs” stands for defects in the software we are using. So at a first glance we can say, that software quality depends on the number of bugs it contains.

An error-free running software is an important quality issue for the user. But the past has shown that correctness of the code is not the only quality attribute. For many firms their software became an important asset in their production and even a competitive advantage in their specific industry. So the importance of a software can raise ad infinitum, which leads to relatively long software life time cycles of around 10 years. During that time the software has to be maintained and enhanced to conserve its worth.

Software maintenance consists of failure correction, performance enhancement, and adoption to a new technical environment, whereas software enhancement consists of the introduction of new features and adoption to changes in the industry. All these activities require changes in the source code, which become larger and more complex and, as a direct consequence, more expensive with the progressing age of the software. From this point of view “good” software should be easy to change and to expand. A flexible software suits the needs of an agile development, as well. As we can see, there are different aspects which come together and form software quality.

1.1 Contribution

Kitchenham cites a more detailed list of quality aspects in [KP]. This list is composed by Garvin [Gra84], who summarized this aspects in views. He defined the user view, which sees quality as fitness for the user’s purpose; the view of the manufacturer, which sees quality as conformance to specification; the product view, which sees quality as tied to inherent characteristics of the product; and the value-based view, which sees quality as dependent on what a customer is willing to pay for it. This thesis is about the assessment of software quality with source code metrics and has therefore a product view of quality.

Computer science came up with several models to describe the product quality of software. The ISO tried to consolidate the different quality views in a general quality model [iso06]. Models like the ISO standard give a good idea of what software quality means in the industry. Unfortunately, these guidelines operate on a high abstraction level and are hard to assess for the different software artifacts. A further problem is the objectivity of quality estimation. How can one ob-
jectively assess an abstract quality concept such as maintainability or changeability. Source code metrics promise an objective and automated way to gather informations about the most important outcome of software development: the code.

We analyze several existing quality models to evaluate how useful source code metrics are to express quality on their specific abstraction levels. In order to compare these different models we define four dimensions of quality, i.e., Perspective, Time, Abstraction and Comparability. We relate these different dimensions also with capabilities of a source code metric based quality model (SMQM) and show that this kind of model can not express all quality attributes desired by high level quality models, but in a development specific context a SMQM can provide useful informations.

Further, we line out how the usage of source code metrics can support the assessment of software quality. Others have come up with source code metrics to measure what occurs in the code [CK94, BBM96]. We assess the code quality on a higher abstraction level based on combinations of source code metrics. Our Quality Assessor Tool summarizes metrics to express quality attributes which provide informations about the source code quality and therefore about the inner state of the product.

1.2 Thesis Outline

In Chapter 2 we sketch dimensions of quality. Then we explain several aspects of quality and measurement in Chapter 3, e.g., the use and abuse of quality and the problem of validity. In Chapter 4 we present five popular quality models and relate them to the dimensions from Chapter 2. Chapter 5 presents the model implemented in the Quality Assessor Tool which is described in Chapter 6. The final part of the thesis deals with our conclusions and future work on the subject.
Chapter 2

Dimensions of Quality Models

Software quality is a very abstract term. In this chapter we will describe four dimensions a quality model consists of. The first dimension represents the point of view the model has. The second describes when the quality is measured. The third dimension describes the level of abstractness the model has and the forth dimension takes the base of comparison into account. Since we focus on quality assessment based on source code metrics, we set a source code metric based quality model (SMQM) in context to these dimensions. A SMQM is defined by a quality model that exclusively uses metrics which can be measured directly from the source code. This metrics are then used to express quality attributes on a higher level of abstraction.

2.1 Dimension: Perspective

Software can be considered from different perspectives because important aspects of one perspective do not have to be as much important for another one. However we segment four main perspectives. The first one is the one of the programmer. They normally work on a low abstraction level and are interested in several classes and modules of the whole product. A broad quality model is not useful for them, but a concrete model can advise them how to implement quality into the code. The second perspective is the one of the architects, who have a more global view of the software product and know about the interactions between modules. Their main focus is on the technical side of the product. The third perspective belongs to the managers. The priorities of the managers lie in the software development process. Cost efficiency, in-time-production and satisfaction of the user’s needs are more important to managers than technical qualities. Finally, the position of the user establishes the fourth perspective. He is not interested in technical inside qualities like complexity or flexibility. The things that counts for him are the proper functionality of the software on the one hand and its price on the other. The level of abstractness is highly interconnected with these points of view. But this issue will be treated later on.

Figure 2.1 relates the different perspectives. The user will recognize the product as a whole (product view). He will assess its quality by working with it. The process view and therefore the one of the manager, meets his aspect of quality by the amount of costs and the efficiency of production. In this work we ignore most of the process part of quality and focus more on the product quality.

The architect is the one that draws the design, i.e., a blueprint of the product. He creates the software framework. We define this view as external view because the general view of the whole product is given here. The external view can not be seen as a homogeneous unit. In a larger software project it is a multi layered organization consisting of different abstraction levels of the
software design. These include the modules of architect, chief architect and product architect. A quality model on one specific technical abstraction level provides general informations that are useful to gain an overview of strengths and weaknesses of the product components inside. However, on this specific level, the functional responsibilities become distributed over the single modules. The external view and the process view are geared in the sense that actions in either one of these views can influence the product quality. As an example, the introduction of reviews at regular intervals is a decision made by the management and has a positive influence on the product quality. The external view instead is responsible for implementing the semantics in the sense of software requirements.

Regarding the internal view we define it as the place where the single line of code, the loops, classes and single packages stand in the foreground. This level is tightly coupled to the external view because normally, the responsibilities are delegated by them. The way how these responsibilities become implemented influences the internal quality. This means that the better the state of the art is recognizable in the code, the better its internal quality, i.e., the code becomes more flexible when the object-oriented paradigm is well achieved. In a certain sense, the internal view is responsible for the syntax of the product or its style respectively. This internal quality can influence the external quality reactively because the more flexible the code of a responsibility is implemented, the easier the architect can achieve changes in the requirements of the product. This is indicated by the bidirectional arrow from the inner circle to the outer in Figure 2.1. We want to stress the statement that the internal quality affects the external quality. This means that an improvement of inner quality (source code quality) leads to an improvement of the overall quality of the product.

![Figure 2.1: A model of the viewpoints and the different quality level.](image)

## 2.2 Dimension: Quality and Time

Assessing the quality of a thing is usually based on a snapshot. Software developing is a dynamic process with several stages and, therefore, different amounts of information are available
at different points of the software’s life cycle. At the design phase there are only UML-diagrams available to assess/predict the quality, whereas a legacy system delivers informations from different sources (e.g., source code metrics, bug reports, change metrics, versioning data). The question arises: “When does it make sense to take a snapshot from the software and do we have enough information to make a representative statement about the assessed software?”

The quality assessor model we implemented is based on data available at the design phase and the beginning of the implementation. Bansiya [BD02] claims that in the object-oriented methodology the design serves as a blueprint for the implementation. A design with good quality is more likely to lead to a high quality implementation.

Today, more and more software is developed incrementally. This leads to extensions of and changes in the design and affects the quality in an appropriate manner. So there is also an evolutionary aspect. This means the quality has to be tracked over time. The tracking of quality over different evolutionary phases allows us to check if the changes made affected the quality in a positive way. If one has to add new features, the external quality improves and that has to be visible in the model. The quality assessor tool we implemented tracks the evolution of the design and how the changes between releases had affected the quality attributes.

2.3 Dimension: Level of Quality Abstraction

As mentioned before, there are many points of view of software quality. They are situated on different levels of abstraction visualized in Figure 2.2. Users, for example, are not interested in metrics or complexity measurements and neither are managers. They want to see operating figures, which summarize the status of development. Managers themselves have to care about the development process as well. The managerial level depends on the information provided by the abstraction level underneath. The better the lower level reflects the quality inside the product the better the managerial level can assess the overall quality. So the more abstract the level, the less important become technical details measured by source code metrics.

The architectural level is more focused on the technical details. It needs a better knowledge of the inside of the product. The idea is that a quality model gives a consolidated overview on the quality state of the product’s insides.

A source code metric is in itself already an abstraction. It reflects a certain aspect of the code and can be directly assessed by inspecting the code.

The possibility to filter unimportant information or to generate new information, by extracting and combining them with information of the lower level, are needed to bridge the gaps between the different abstraction levels. A model, who is able to extract data from source code and combine them in a specific way, allows to bridge the gap between the source code and the architectural level. A source code metric based quality model should support abstract constructs on the architectural level as well as direct data acquisition on the source code level. There are already different models which address this issues such as the one presented by Dromey [Dro95].

2.4 Dimension: Compareability of Quality

Quality is always measured in relation to something. The high jumper who jumps 2m40 high is one of the best in the world but this performance is only the best relative to its competitors. So the quality is relative to other performances of the same kind. To assess the quality of one thing one always needs a basis of comparison.

There are several imaginable bases. One is another thing of the same kind. Another is a comparison relative to itself. And the third is the comparison with a threshold or requirement. A
high jumper who was able to increase his average height from 1m80 to 2m made a progress in his performance. Figuratively, we can say he increased his quality.

Software needs a basis of comparison, too. Mostly it is compared by its user, who decides which one fits him the best. But in some cases there is only one software of its kind available. The navigation software of a space shuttle as an example. In such a situation, it is said often that software is good, when it covers the user’s need or the agreed requirements. The most common approach to assess this degree is testing.

Taking the above mentioned basis of comparison into account, only external aspects of high jumpers or software are compared. Under considering the different perspectives, internal quality aspects have to be taken into account. For a high jumper aspects like a constitutive training and a balanced diet are also important for his product “the high jump”. For software it is similar. A good coding style leads to better understandable code and to a better product, in turn. Therefore it is not enough to consider external attributes like the behavior only. Internal attributes like good coding style needs also be compared to receive a quality impression of the software product. This is the issue where the strength of SMQM comes up. The strength of a SMQM lies in evaluating the inside of the software.

**2.5 Dimensions and Quality Assessing with SMQM**

The explained dimensions cover a broad spectrum of the quality term. In this section we relate the different scopes defined to our desire to use source code metrics to assess quality. Thus we are looking for a model that takes source code metrics as its input. This model should allow to assess the quality of the underling software code. We call this model a source code metric based quality model (SMQM).
2.5 Dimensions and Quality Assessing with SMQM

2.5.1 Viewpoint of the SMQM

Since we focus on the assessment of quality with source code metrics, our viewpoint is predetermined. A source code metric by itself does not contain information about features of the whole software. That is the reason why a total source code metric based quality model never can provide enough information to satisfy the needs of the manager or even the user to assess quality. We marked in Figure 2.3 on the left the viewpoints where a source code metric based model can offer useful statements. The fading out of the red color should indicate, that with an increasing level of abstraction within the architectural perspective the source code metric model loses its accuracy.

![Diagram](image)

**Figure 2.3**: Scope of Quality and Source Code Metric. The left side corresponds to Figure 2.1 and the right side to Figure 2.2. The red marks areas where source code based quality models can be settled within the scopes of quality.

2.5.2 Time and SMQM

Source code metrics can be accumulated after a first implementation at the earliest. Studies claim that after a first implementation it is too late and too expensive to improve the software product quality, because fundamental design decisions have already been made and big changes in the design are difficult or even impossible. However, source code metrics can, after a first implementation, assess the code in a retrospective way.

2.5.3 Abstraction and SMQM

As we mentioned above, the abstraction level of source code metrics is quite low. How far the measurement results can be abstracted and how far this abstraction conserves accuracy is questionable. The higher the abstraction of quality the more important become information, which can not be provided by the source code only, *i.e.*, requirements and performance. We marked in Figure 2.3 on the right, the level of abstraction where source code metrics can fully represent the quality of the product in the desired amount.

2.5.4 Comparability and SMQM

The quality assessment based on source code metrics must answer the question about the basis of comparison. A comparison between different software products of different domains becomes difficult on the level of source code metrics because of the different requirements the products have to satisfy. Dependent on these requirements one software is inherently more complex than another. But if two products are built with equal requirements they become comparable through a
source code metric based model. Another basis of comparison could be a former release. Software will be changed several times during its life cycle. This changes are recognizable in a before–after comparison of the source code metrics and so they must also be recognizable in a source code based quality model.

2.6 Summary

In this chapter we presented several dimensions of quality and related them to a source code metric based quality model (SMQM). We have already argued that such a model can only provide limited information about the overall quality of a software product. The main limitation of SMQMs are that source code metrics do not reflect semantics, i.e. the functionality that the software implements. Nevertheless they allow to model the inner quality of the software.
Chapter 3

Quality and Measurement

Measuring is a way to describe characteristics of things. It should allow to become an impression of its inside. Measurement results should then serve as basis of decision. Especially the management likes measures to calculate risks and keep control over the product. Because of its immateriality, software is difficult to measure. There has several abstract quality terms been defined which should allow to minimize the product’s risks. i.e., Maintainability has become critical since software became an important asset for specific firms. But how can we measure the software to assess the degree in of maintainability?

However if something is not measurable, make it measurable as Galileo Galilei already said. There are many metrics proposed by the community to measure what is going on in the code, the development process and the project team. Some of them are quite easy to evaluate because they are directly measurable by counting lines of code or number of attributes. This metrics are mostly provided by the source code itself and analyze its static structure. Others are more complex and can not be collected that easy. They only can be gathered indirectly via a combination of direct metrics or other indirect metrics.

In order to assess the software quality, an appropriate set of software metrics shall be identified which express several quality attributes. IEEE introduced a standard for a software quality metrics Methodology [otICS98]. Another popular approach of defining metrics is the Goal-Question-Metric methodology of Basili et al. [BR88].

Chidamber and Kemerer developed a metric suite for object-oriented design [CK94]. This metric suite was validated by Basili et al. [BBM96]. It contains the following metrics which are partially introduced in our model: Weighted Methods Per Class; Depth of Inheritance Tree; Number of Children; Coupling between Object Classes; Response For a Class and Lack of Cohesion in Methods. Also other metrics from Martin [Mar03] and Henderson-Sellers [HS96] complete the metric suite of our model.

3.1 Objectivity vs Subjectivity

Until now, the understanding of measuring software quality is not yet sophisticated enough and is still far away from being standardized. There are two parties. One believes in the possibility to express objectively the quality of software by using measures. Others believe that quality is a subjective term and to measure it should accommodate this circumstance. The latter points out that measures can not take into account, the environment software depends on. The former believes more in the impartiality of measuring. Measurement is one way to describe the characteristics of real-world objects. It is a mapping between the real world and the mathematical one [PJCK97].
We are sure that there are aspects of quality that are indeed measurable. This measures can give useful informations about the quality of the code. Because of the complexity of quality, a collection of source code measures will not allow to make concluding statements about the software quality as a whole. The measured results have to be related with the context of the software development, i.e., available resources, process organization.

3.2 Validation of Measures

To measure something is relatively easy. The major difficulty is to interpret the results in a meaningful way and the biggest problem is to prove the validity. A measure is valid if it captures numerically the behavior we perceive in the empirical world. It is difficult to demonstrate that a measurement is valid since the proof has to be empirical by its nature. Measurement can make visible what is going on in the code but unfortunately it does not clearly identify what was the cause of the measured mutations.

Validation is critical to the success of software measurement [BK95] and for SMQM. A SMQM without a validated fundament makes no sens. Kitchenham et al. proposed a framework that can help researchers and practitioners to understand, how to validate a measure, how to assess the validation work of others and when it is appropriate to apply a measure in a given situation. Another methodology was designed by Schneidewind [Sch92].

3.3 Use and Abuse of Measurement

The introduction of measures allows implicitly to introduce thresholds. With these thresholds developers receive a direct feedback about the quality of their code. This way they are able to build quality directly into the code by considering the measurement and the corresponding thresholds.

But there is also a danger of abuse. As an example, a software development company claims that understandability is an important aspect of their internal quality. To express this and assess this issue they use a “lines of code” metric. The programmer uses every possibility given by the programming language to pack his code in order to satisfy this aspect. But does the satisfaction of a measurement support the underlying quality aspect? This example shows the problem of measuring. Good quality does not mean the same as to satisfy thresholds. Metrics can be used to identify abnormalities.

3.4 The Problem of Metric Combination

It is often the case that certain characteristics are not directly measurable. A casual strategy to solve this problem is to combine direct metrics. The consequence of this combination is that the more metrics are combined the more they affected by several different influences. It becomes difficult to evaluate a direct connection between the measured result and its cause.

The broader the considered measurement basis is, i.e., measures of a class, package, couple of packages, the whole system, the less visible becomes a single change of the code in the measurement result. Single actions, which lowers the quality, get more and more socialized the broader the scope of consideration is.

The combination of metrics is not always avoidable. Sometimes they are necessary to reach meaningful values for higher abstraction levels.
3.5 Metrics in the Object-Oriented Paradigm

New aspects like encapsulation, cohesion, coupling and inheritance came up with the introduction of the object-oriented paradigm. This aspects are claimed to improve the software flexibility, extendability and changeability with the goal to satisfy our growing needs for complex software. The practice has shown, that this advantages are far from being inherent of the paradigm. There are several rules or heuristics to follow in order to accentuate the gain of object-orientation. Thus, to check in which degree these rules have been followed, metrics are needed. But these rules are still vague and full of trade-offs [Rie96].

Also the questions of how, when and where you measure and assure quality are from being a settled issue. The switch to the object-oriented paradigm has changed the elements that we use to assess software quality. Traditional software product metrics that evaluate product characteristics such as size, complexity, performance, and quality must be changed to rely on some fundamentally different notions such as encapsulation, inheritance and polymorphism which are inherent in object-orientation [BD02].

3.6 Summary

The supply for metrics is large but the validation strategies are weak. However source code metrics already have a long tradition and some of them are quite well validated [BBM96]. When choosing a SMQM it is implicit that one believes more on the objective part of programing. So a SMQM is a try to assess source code quality in a objective way with the danger of abuse and the fact that with multiple combination of the single metrics the causes of the result not can be backtracked anymore. But on the other hand the chance to make well founded decisions.
Chapter 4

Quality Models

Since software quality is important because it affects manifold aspects of our daily life, several approaches to model the quality of a software product have been published. In this chapter we present several of this approaches in chronological order. We will then relate them with the dimensions explained in Chapter 2 and to the SMQM.

4.1 McCall

McCall’s quality model introduced in 1977 [MRW77] is one of the first model of its kind. Its prior focus lies on developers and the development process. By choosing software quality factors, that reflect the user’s and the developer’s point of view, McCall et al. tries to close the gap between these two stakeholders.

McCall’s model is a typical category based, hierarchical model. On the top level, we have three major perspectives. Product revision perspective, at first, defines the ability of the software product to undergo changes. Second, the product transition perspective stands for the adaptability of the software to new environments and, at last, product operations represents the software operation characteristics. Every of this three categories include several quality factors:

- **Product revision:**
  - Maintainability: the effort required to locate and fix a fault in the program within its operating environment
  - Flexibility: the ease of making changes required by changes in the operating environment
  - Testability: the ease of testing the program, to ensure that it is error-free and meets its specification

- **Product transition:**
  - Portability: the effort required to transfer a program from one environment to another
  - Reusability: the ease of reusing software in a different context
  - Interoperability: the effort required to couple the system to an other system

- **Product operations:**
  - Correctness: the extent to which a program conforms to its specification
– Reliability: the system’s ability not to fail
– Efficiency: further categorized into execution efficiency and storage efficiency and generally meaning the use of resources, e.g. processor time, storage
– Integrity: the protection of the program from unauthorized access
– Usability: the ease of the software

In McCall’s Model there are also 23 quality criteria defined. These criteria are attributes of one or more quality factors. Metrics are used in order to quantify aspects of the criteria. The quality metrics are achieved by answering an amount of “yes” or “no” questions. Depending on the given answer, the quality is assessed. McCall’s model has been criticized because the quality judgment is subjectively measured, based on the judgment of the person answering the questions.

### 4.2 Boehm

An other quality model was introduced by Boehm[BBK+78] in 1978. It is also an important predecessor of today’s quality models. Boehm takes the contemporary shortcomings of models into account, which automatically and quantitatively evaluate the quality of software. Basically, his model tries to define software quality qualitatively by a given set of attributes and metrics. There are certain parallels recognizable between McCall’s model and the model of Boehm. For example, both propose a hierarchic structured model with high-level, intermediate level, and low-level characteristics. All of these characteristics influence the upper quality levels.

On the top level characteristics of Boehm’s quality hierarchy, there are three high-level characteristics addressing three main questions that a buyer of software may have.

- As-is utility: How well (easily, reliably, efficiently) can I use it as-is?
- Maintainability: How easy is it to understand, modify and retest?
- Portability: Can I still use it if I change my environment?

At the intermediate level there are 7 quality factors that represent the qualities expected from a software system:

- Portability: The code can be operated easily and well on other environments.
- Reliability: The code performs its intended functions satisfactorily.
- Efficiency: The code executes its intention without waste of resources.
- Usability: The code is reliable, efficient and human-friendly-engineered.
- Testability: The code eases setting up verification criteria and supports evaluation of its performance.
- Understandability: The code is easy to read in the sense, that inspectors can rapidly recognize its purpose.
- Flexibility: The code is easy to change when a desired change has been determined.

At the bottom level of the model there are primitive characteristics metrics hierarchies. This characteristics form the basis to define quality metrics. To build such a basis was one of the goals Boehm wanted to achieve. The model proposes consequently at least one metric, which should
4.3 ISO 9126

The International Organization for Standardization presented a quality model for software products which reached a broad acceptance in the software engineering community [iso06]. The standard is based on the models from McCall and Boehm. It has also a hierarchical design with the total quality at the top, six quality factors on the second level and several subfactors at the bottom. The quality factors are listed below.

- Functionality: A set of attributes that bear on the existence of a set of functions and their
specified properties. The functions are those that satisfy stated or implied needs.

- **Reliability**: A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- **Usability**: A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.

- **Efficiency**: A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

- **Changeability**: A set of attributes that bear on the effort needed to make specified modifications.

- **Portability**: A set of attributes that bear on the ability of software to be transferred from one environment to another.

Up to now, the standard 9126 has been extended by 3 substandards. ISO 9126-2 defines external metrics, ISO 9126-3 defines internal metrics and ISO 9126-4 defines quality in use metrics, for measurement of the characteristics or subcharacteristics. Internal metrics measure the software itself, external metrics measure the behavior of the computer-based system that includes the software, and quality in use metrics measure the effects of using the software in a specific context of use. None of these standards claim to be complete, nor do they assign ranges of values to rated levels or to grade of compliance.

### 4.4 Dromey

In his work [Dro95], Dromey points out, that software does not directly manifest high level quality attributes. Software only possesses product characteristics that influence the quality attributes. Bad product characteristics reduce its quality attributes. The models mentioned above do not make an explicit connection between quality attributes and product characteristics. Dromey’s model focuses on the primary software product, the code. This product orientation is the most important issue of Dromeys work.

Dromey says, that a direct attribute decomposition in the style of the ISO model is not the best way to go, because this only leads to other vague attributes. He proposes a single level of “quality carrying properties” between high level attributes and the product components. This resulted in his generic model (see Figure 4.2). This framework allows a top-down modeling (To each high level quality attribute, quality carrying properties can be assigned to.) as well as a bottom-up modeling (For each component quality-carrying properties are identifiable, which are important to guarantee the high level attributes.).

In the context of software development, Dromey sets “components” equal to “structural forms” of programming languages (e.g., expressions, variables, loops etc.). The set of “structural forms” is determined by the programming language. An example of how such an assignment form structural to quality carrying properties could look like is shown in Figure 4.3. If one of these properties is hurt, this leads to a quality defect which lowers the integrity of the quality property. A “violation of a property” does not mean the same as “the software has a functional defect”. In some circumstance only nonfunctional properties are impacted by quality defects. For example an inconsistent expression contains side-effects. This must not lead to a functional defect but it makes the maintenance more difficult.

Dromey proposes a set of structural properties. An overview is presented in Table 4.1. As an example we discuss the Quality Carrying Property “Assigned”. A variable is assigned if it
receives a value before its first use. So the Property “Assigned” can be applied to variables. This means, if our code has no unassigned variables we built quality into our software. This relationship builds the basis of Dromey’s model. On this basis the connection to high level attributes can be constructed similarly. The product properties characterize the requirements that must be satisfied to build a high level quality attribute into software. The most difficult part in this situation is to evaluate which product property does have the most significant influence on the quality attribute. For his model, Dromey choose a list of quality attributes, which is similar to the ISO 9126 (Functionality, Reliability, Usability, Efficiency, Maintainability, Portability). As an extension to ISO 9126 he added the attribute reusability, which he regards as an important subject.

<table>
<thead>
<tr>
<th>Correctness Properties</th>
<th>Structural Properties</th>
<th>Modularity Properties</th>
<th>Descriptive Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computable</td>
<td>Structured</td>
<td>Parameterized</td>
<td>Specified</td>
</tr>
<tr>
<td>Complete</td>
<td>Resolved</td>
<td>Loosely coupled</td>
<td>Documented</td>
</tr>
<tr>
<td>Assigned</td>
<td>Homogeneous</td>
<td>Encapsulated</td>
<td>Self-descriptive</td>
</tr>
<tr>
<td>Precise</td>
<td>Effective</td>
<td>Cohesive</td>
<td></td>
</tr>
<tr>
<td>Initialized</td>
<td>Nonredundant</td>
<td>Generic</td>
<td></td>
</tr>
<tr>
<td>Progressive</td>
<td>Direct</td>
<td>Abstract</td>
<td></td>
</tr>
<tr>
<td>Variant</td>
<td>Adjustable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistent</td>
<td>Range-independent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Utilized</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: A List of the Quality-Carrying Properties. Each of these properties can be applied to one or more structural forms, and affects one or more quality attributes

### 4.5 Bansiya

Bansiya et al. [BD02] introduced a quality model for object oriented design (QMOOD). It focuses the quality of the design by using source code metrics available on this stage of the development. The model implements a way to map source code metrics to higher abstraction levels. We explain the details of this quality model in Chapter 5.
Prominent Models in Context with SMQM

The first three models mentioned have a global view of quality (see Figure 4.3). They include many aspects of quality in one framework. Therefore they have to be at a high abstraction level. The most difficult challenge is to sketch such a fuzzy thing as quality in a meaningful and human manageable way. We state that especially the latter criterion is very important. If we lose ourself in decomposing quality concepts into smaller pieces, we will not be able to assess any software accurately. The danger to do so increases in an absolute top-down approach because every abstract construct can easily be divided into other abstract constructs but the jump to concrete objects remains difficult. In his model, Dromey addresses exactly this problem of decomposition. We think that this three part strategy, i.e., quality attributes, product properties and components, is a good way to go, because of its tight connection between the abstract modeling and the concrete thing. His approach considers the bottom-up strategy as well as the top-down strategy. We agree that a combination of both strategies leads to a goal-oriented quality model.

The environment the mentioned models are covering is displayed in Figure 4.4. We used the environment model we defined in Chapter 2. All models have in common that they ignore the quality of the development process. They are all focused on the product quality itself and how it can be described. The general models such as the ones from McCall and Boehm as well as the ISO standard includes multiple points of view. This models depend on quality information which has to be gathered from several points of view. The attribute usability has to be assessed from the users perspective whereas the maintainability only can be assessed by the architects. In this
models it is necessary to measure quality at at least two different places. We draw Dromey’s and Bansiya’s models as circles in the external view because their strategies take only information into account which can be extracted from the software itself. So we could say that the first category of models (McCall, Boehm, ISO 9126) tries to unite the internal and external view and the quality as the user perceives it. The second (Dromey, Bansiya) takes the internal and external view into account. This means only information available by sources code of the product itself are considered to assess the quality. This way a technical assessment of the software as product is possible (syntax) but ignoring the users point of view makes it impossible to assess the semantical quality of the product. With this reduction the use of source code metrics for quality assessment becomes applicable. So assessing quality with source code can only cover the architect/developer part of comprehensive quality models like the ISO 9126.

Figure 4.4: Quality models related to the environment.
The quality models mentioned in the previous chapter consolidate as much perspectives as possible. In our work we implemented a model that makes it possible to gain an overview of the internal quality (architecture and implementation) based upon the results of source code metric analysis. Bansiya et al. [BD02] address exactly that issue. They connect low level source code metrics with higher level quality attributes in a quality model for object oriented design (QMOOD). The QMOOD is therefore a possible implementation of a SMQM.

The model was inspired by the idea of Dromey. So it represents basically a three tier setup with quality attribute, design properties and components. Its view is focused on the design, and the approach of using product metrics is based on the assumption that measuring and controlling internal product properties (internal quality indicators) will result in improved external product behavior (absence of failures, simplicity to change, quality in use) [KP].

The model aims at the early stage in the development process by evaluating the quality of the software design. With the introduction of the object-oriented paradigm the importance of the design increased. It represents objects and how they interact with each other. If this interaction is already wisely created, a premise is given for a software with a good quality standard. Of course, this premise is necessary but not sufficient.

To express the QMOOD [BD02] in terms of our four dimensions of Chapter 2. It has the external view of an architect; takes a snapshot of the quality during the design phase; has a technical abstraction level; and the base of comparability is an earlier release (This issue is discussed later on).

![Figure 5.1: Levels and links in QMOOD](BD02)
5.1 The Four Levels

The Quality Model for Object-Oriented Design (QMOOD) consists of four Levels and three relationships between them (see Figure 5.1). The first level \( L_1 \) represents the quality attributes. They should be broad enough to include all aspects of design quality. These attributes are targeting quality attributes of the design. For a detailed definition see Table 5.1. \( L_2 \) represents the Design Properties which influence the quality attributes. For a list of the used design properties see Table 5.2. At the third level we find the design metrics. They express the degree how well the design fits the properties defined in \( L_2 \). Here the issues mentioned in Chapter 3 become critical. If the metrics are not valid in the sense that the measurement does not reflect the defined design property sufficiently, the expressiveness of the whole model is in danger.

\( L_3 \) is an additional level compared to Dromey’s approach. The concrete components of \( L_4 \) are transfered in a numerical form and thus gain objectivity.

As one can see in Figure 5.2 design components represent the fourth level of the methodology (\( L_4 \)). These components are mostly determined by the programming language (e.g., methods, objects, classes). \( L_4 \) delivers the source (methods, classes, packages etc.) which will be measured on the upper level (\( L_3 \)). This process is represented by the arrow \( L_3 \rightarrow L_4 \).

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuseability</td>
<td>Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functional related capabilities.</td>
</tr>
<tr>
<td>Understandability</td>
<td>The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.</td>
</tr>
<tr>
<td>Functionality</td>
<td>The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.</td>
</tr>
<tr>
<td>Extendability</td>
<td>Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>This refers to a design’s ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.</td>
</tr>
</tbody>
</table>

Table 5.1: Definition of Quality Attributes [BD02]

5.2 Design Metric \( ightarrow \) Design Property

The next step consists of assigning design metrics to design properties (\( L_3 \)). Here it is possible to combine the metrics in a meaningful way. The validity of the design property depends directly on the metrics and the combination of them respectively. A combination of source code metrics has to be done carefully because the more metrics are combined the less the influence of a single sources can be backtracked reliably. So a tradeoff must be made between the expressiveness and the traceability of a design property. Bansya uses only one metric for one design property, so we can say that the mapping between \( L_2 \) and \( L_3 \) is direct and a good traceability is given. In Table 5.2 one can see in the last column which metrics are used to assess the design properties.

\(^1\) The used quality attribute terms in this chapter refers to this definition
<table>
<thead>
<tr>
<th>Design Property</th>
<th>Definition</th>
<th>Measured with...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Size</td>
<td>A measure of the number of classes used in a design.</td>
<td>Design Size in Classes</td>
</tr>
<tr>
<td>Hierarchies</td>
<td>Hierarchies are used to represent different generalization-specialization concepts in a design. It is a count of the number of non-inherited classes that have children in a design.</td>
<td>Number of Hierarchies</td>
</tr>
<tr>
<td>Abstraction</td>
<td>A measure of the generalization-specialization aspect of the design. Classes in a design which have one or more descendants exhibit this property of abstraction.</td>
<td>Average Number of Ancestors</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>Defined as the enclosing of data and behavior within a single construct. In object-oriented designs the property specifically refers to designing classes that prevent access to attribute declarations by defining them to be private, thus protecting the internal representation of the objects.</td>
<td>Data Access Metric</td>
</tr>
<tr>
<td>Coupling</td>
<td>Defines the interdependency of an object on other objects in a design. It is a measure of the number of other objects that would have to be accessed by an object in order for that object to function correctly.</td>
<td>Direct Class Coupling</td>
</tr>
<tr>
<td>Cohesion</td>
<td>Assesses the relatedness of methods and attributes in a class. Strong overlap in the method parameters and attributes types are an indication of strong cohesion.</td>
<td>Cohesion Among Methods in Class</td>
</tr>
<tr>
<td>Composition</td>
<td>Measures the “part-of”, “has”, “consists-of” or “part-whole” relationships, which are aggregation relationships in an object-oriented design.</td>
<td>Measure of Aggregation</td>
</tr>
<tr>
<td>Inheritance</td>
<td>A measure of the “is-a” relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.</td>
<td>Measure of Functional Abstraction</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>The ability to substitute objects whose interfaces match for one another at run-time. It is a measure of services that are dynamically determined at run-time in an object.</td>
<td>Number of Polymorphic Methods</td>
</tr>
<tr>
<td>Messaging</td>
<td>A count of the number of public methods that are available as services to other classes. This is a measure of the services that a class provides.</td>
<td>Class Interface Size</td>
</tr>
<tr>
<td>Complexity</td>
<td>A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.</td>
<td>Number of Methods</td>
</tr>
</tbody>
</table>

*Table 5.2: Definition of Design Properties [BD02]*
5.3 Design Property → Quality Attribute

The step \( L_{12} \) is probably the most interesting one. It consists of the combination of design properties to quality attributes. Bansiyas uses weighted design properties to build one quality attribute. For the weightings and property combination see Table 5.3. The backtracking problem explained above also exists in this context.

<table>
<thead>
<tr>
<th>Quality Attribute</th>
<th>Index Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>(-0.25 \times \text{Coupling} + 0.25 \times \text{Cohesion} + 0.5 \times \text{Messaging} + 0.5 \times \text{DesignSize})</td>
</tr>
<tr>
<td>Flexibility</td>
<td>(0.25 \times \text{Encapsulation} - 0.25 \times \text{Coupling} + 0.5 \times \text{Composition} + 0.5 \times \text{Polymorphism})</td>
</tr>
<tr>
<td>Understandability</td>
<td>(-0.33 \times \text{Abstraction} + 0.33 \times \text{Encapsulation} - 0.33 \times \text{Coupling} + 0.33 \times \text{Cohesion} - 0.33 \times \text{Polymorphism} - 0.33 \times \text{Complexity} - 0.33 \times \text{DesignSize})</td>
</tr>
<tr>
<td>Functionality</td>
<td>(0.12 \times \text{Cohesion} + 0.22 \times \text{Polymorphism} + 0.22 \times \text{Messaging} + 0.22 \times \text{DesignSize} + 0.22 \times \text{Hierarchies})</td>
</tr>
<tr>
<td>Extendibility</td>
<td>(0.5 \times \text{Abstraction} - 0.5 \times \text{Coupling} + 0.5 \times \text{Inheritance} + 0.5 \times \text{Polymorphism})</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>(0.2 \times \text{Abstraction} + 0.2 \times \text{Encapsulation} + 0.2 \times \text{Composition} + 0.2 \times \text{Inheritance} + 0.2 \times \text{Polymorphism})</td>
</tr>
</tbody>
</table>

Table 5.3: Computation Formulas for Quality Attributes [BD02].

As we can see the weights can either be positive or negative. The algebraic sign indicates that the specific design property has a positive, or a negative respectively, influence on the quality attribute. For example, reusability is positively affected by the design size (The more classes we have, the more we could reuse), the cohesion (The more cohesive the design the more modules can be unhinged to be used in other projects) and messaging (The more services the design provides the more likely it can be used in another context). On the other hand, coupling lowers the reusability (The more an object is coupled to an other one, the less it is possible to use it in a different context).

The sum of the weighted design properties is in the range of \([-1...+1]\) so that every quality attribute have the same range. For positive influences an initial weighted value of \(+1\) or \(+0.5\) was set. For negative influences \(-1\) or \(-0.5\) has been chosen. This value has then been changed proportionally in that way that the sum of the resulted weights results to \(\pm 1\).

For a better illustration of the correlation between the quality attributes and the design properties see Table 5.4.

5.4 The Adapted Quality Model

Bansiyas approach has multiple elements. We divided it into the mapping from design properties to quality attributes (\(L_{12}\)) and the assignment of metrics to design properties (\(L_4 \rightarrow L_{34} \rightarrow L_3\)). In the later element we combined several steps into one because they are highly interconnected. In our work, we adapted Bansiyas model for the Java programming language and had thus to focus on Java source code metrics. We decided to use the Java Eclipse Plugin “Metrics 1.3.6” [met06] to measure our code and implemented our model with the metrics provided by this plugin.
5.4 The Adapted Quality Model

### Table 5.4: Correlation between Quality Attributes and Design Properties [BD02].

A ↑ indicates a positive correlation between the design property and the quality attribute, i.e., a better design size value influences the reusability positively. A ↓ point to a negative correlation, i.e., a rising coupling value lowers the flexibility.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✆</td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hierarchies</td>
<td></td>
<td></td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>✆</td>
<td></td>
<td></td>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>↑</td>
<td></td>
<td>↑</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coupling</td>
<td>✆</td>
<td>✆</td>
<td>✆</td>
<td></td>
<td>✆</td>
</tr>
<tr>
<td>Cohesion</td>
<td></td>
<td>✆</td>
<td></td>
<td></td>
<td>↑</td>
</tr>
<tr>
<td>Composition</td>
<td>✆</td>
<td>✆</td>
<td>✆</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inheritance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>↑</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>➡</td>
<td>✆</td>
<td>✆</td>
<td>✆</td>
<td></td>
</tr>
<tr>
<td>Messaging</td>
<td>✆</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td>✆</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4.1 Metrics 1.3.6

“Metrics 1.3.6” [met06] is an open source plugin for the Eclipse IDE. It allows to measure source code and make metrics available which are defined in the books “Object-Oriented Metrics, Measures of Complexity” [HS96] and “Agile Software Development” [Mar03].

5.4.2 Used Measurement

This metrics are not exactly equivalent to the metrics Bansiya proposed but they can express the same information. An overview of the replacement of the metrics is given in Table 5.5. In the same table one can recognize which design property is expressed by the replaced metric. A more detailed explanation of the replacement can be found right below.

- “Design Size in Classes” with “Number of Classes”: They are exactly the same.
- “Number of Hierarchies” with “Depth of Inheritance Tree”: Hierarchies reflect the generalization/specialization concepts. In the model this affects the functionality positively. NOH means the more hierarchies in the design the better the functionality. Depth of Inheritance Tree is the average of all classes in the design and means the average distance from class Object in the inheritance hierarchy. The deeper the hierarchy goes the higher the values and the better the functionality.
- “Average Number of Ancestors” with “Abstractness”: The abstraction design property should quantify in which degree abstract concepts are integrated in the design. The more of these concepts are used, the better the extendability and the effectiveness, but the lower the understandability. The abstractness measure provided by [met06] also represents the degree of using abstraction concepts. It relates the number of abstract classes to the total classes in a package.
- “Data Access Metric” vs “1”: We did not have an adequate alternative for this metric. It would have required to implement a tool to extract the metric. Table [5.5] shows that the
### Design Property | Metrics used by Bansiya | Metrics we used...
---|---|---
Design Size | Design Size in Classes (DSC) | Number of Classes
Hierarchies | Number of Hierarchies (NOH) | Depth of Inheritance Tree
Abstraction | Average Number of Ancestors (ANA) | Abstractness
Encapsulation | Data Access Metric (DAM) | 1
Coupling | Direct Class Coupling (DCC) | Instability
Cohesion | Cohesion Among Methods of Classes (CAM) | Lack of Cohesional Methods
Composition | Measure of Aggregation (MOA) | Number of Attributes
Inheritance | Measure of Functional Abstraction (MFA) | \( 1 - \frac{Number of Overridden Methods}{Number of Methods} \)
Polymorphism | Number of Polymorphic Methods (NOP) | Number of Overridden Methods
Messaging | Class Interface Size (CIS) | Number of Methods
Complexity | Number of Methods (NOM) | Weighted Methods per Class

Table 5.5: Presentation of the used Metric

"Data Access Metric" is used to express encapsulation. Encapsulation itself influences several quality attributes (Flexibility, Understandability, Effectiveness). The value “1” is the neutral value from this attribute in this model.

- **"Direct Class Coupling" with "Instability":** The measurement should mirror the interdependencies between classes of the design. A high coupling value has a negative influence on almost every quality attribute. The most negative effect it has on reusability, flexibility, understandability and extendability. The higher the value the more negative its effect. The instability measure of our metric tool reflects this fact as well. It takes efferent coupling and afferent coupling into account.

- **"Cohesion among Methods" with "Lack of Cohesional Methods":** The most important issue of the cohesion design property is the relationship between the methods and the attributes of a class. LCOM considers exactly that issue. While "Cohesion among Methods" is positive correlated to the cohesion design property, LCOM is negative. So we took the reciprocal value.

- **"Measure of Aggregation" with "Number of Attributes":** Bansiya defines the measure as a count of the number of data declarations whose types are user defined classes. In the object oriented paradigm most attributes are user defined and primitive types can be accounted as a sort of composition, too. A count of attributes per class can be used to reflect the composition design property. A greater degree of composition has a positive impact on flexibility and effectiveness.

- **"Measure of Functional abstraction" with "1 – \( \frac{Number of Overridden Methods}{Number of Methods} \):** The measure of functional abstraction is defined by the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. It should measure the "is-a" relationship between classes. Our equation represents this fact, too. If all

---

2The number of classes inside a package that depend on classes outside the package.
3The number of classes outside a package that depend on classes inside the package.
4It is unsure if LCOM is useful in Java it penalizes the proper use of getters and setters as the only methods that directly access an attribute and the other methods using the getter/setter methods.
methods become overridden, we could say that if the interface of a class consists exclusively of overridden methods the “is-a” relationship is not distinct.

• “Number of Polymorphic Methods” with “Number of Overridden Methods”: Polymorphism should express a measure for the dynamic behavior. Bansiya’s metric counts the number of virtual methods. His point of view is more from the inside of the superclass. The number of overridden methods focuses more on the subclass, but can also indicate polymorphism, because method overriding allows a dynamic reaction of the program during the run-time. The high value of overridden methods indicates a good satisfaction of the polymorphism design property. This design property affects the flexibility, functionality, extendibility and effectiveness positively but lowers the understandability of the design.

• “Class Interface Size” with “Number of Methods”: The replacement is straightforward. Both metrics mean the same with the difference that NOM includes also private methods2. Messaging has a positive correlation with reusability and the functionality.

• “Number of Methods” with “Weighted Methods per Class”: Because our analysis is based on existing software source code, we have access to more implementation details whereby we can use the “weighted methods per class”-metric made available by “Metrics 1.3.6” [met06]. This value takes loops, decision, and logical operations into account to calculate the McCabe Cyclomatic Complexity [McC76]. We decided to use this metric because it allows us to pinpoint the complexity. The influence of complexity is correlated negatively to the attribute of understandability. The higher the complexity respectively the “Weighted Methods per Class”-Value the less understandable is the design.

5.5 Basis of Comparability

As introduced in Section 2.4 the quality model has to be compared to something in order to cover the dimension of comparability. One option is to compare the design with another design, but this under lasts the limitation that both designs must have similar requirements to be comparable. The other option is to consider the evolution of the design. During its life cycle a software product becomes changed many times, i.e., new features, adaptation to new technologies or bug fixes. This changes influence the structure of a design. A before-after comparison can deliver informations about the strength of such influences or if action, which should have improve the design quality, has succeeded.

Independent, which option has been chosen a normalization is necessary. The metrics we use have different value ranges as well as the design properties have. This normalization is required because the single design properties become summarized to an abstract quality attribute.

Bansiya et al. [BD02] propose to order different values in an ordinal scale and use the ranks to evaluate the quality attributes in order to normalize the values of designs from different projects. In our tool we chose the second option and compare the designs of several releases of a software product with each other. To do so we normalized the metric values against a basis release. This will be computed straight forward by dividing each value through its corresponded basis. The normalizing process is explained in the next chapter in more detail.
Chapter 6

Quality Assessor Tool

As mentioned in the chapter before we used the quality attributes defined in Table 5.1, the design properties form Table 5.2, and the metrics from the last column from Table 5.5. With these elements we built an Eclipse plugin that is able to use measures extracted from several releases of a Java product and arrange the high level attributes in a graphical view. With help of this view the quality evolution of the product had to be evaluated. In this chapter we firstly present the process of the data extraction, secondly describe the inner constitution of the tool, thirdly apply our tool to the medium sized, open source software “Azureus” [Azur06a], and finally we analyze the output of the tool.

The tool allows to make quantitative statements about software quality and is integrated in the Eclipse IDE.

6.1 Program Structure

The tool consists of a XML-Importer and the “QualityModel”. The XML-Importer is a SAX-parser and reads the used data into the plugin. Other sources of metrics with corresponding extensions are imaginable, i.e., an SQL database which provides already stored measurement results.

The “QualityModel” is the core of our plugin. It contains “Releases” (Figure 6.1) where each release consists of design properties and quality attributes. Each class of design property implements the method to get its absolute value. Each property draws its absolute value directly from the data source. This way it is easy to add new design properties to the release. The release is then capable to normalize the value of the design property. Each quality attribute implements the method to calculate the value of itself based on the value of the design property provided by its release (see Figure 6.1). The “QualityModel” is able to manage several of this releases and serves as data model for the later visualization.
6.2 Data Input

As mentioned above we used the “Metrics 1.3.6”-Eclipse Plug-in to measure the source code of several releases of a given software. We exported the measurement results into XML-files and used them as a data base for our tool.

We decided to build this XML-data-sources because otherwise “Metrics” had to calculate the results of all releases at one time into performance problems.

Our Eclipse-Plug-in allows to add new XML-datasources through preference page (Window → Preferences... → Quality Assessor). Figure 5.2 shows a screenshot. At the moment, this XML-data-sources need to have the “Metrics 1.3.6” format.
6.3 Model Visualization

For the presentation of the results we use two different views. A visualization in form of a table and one in form of a graph. The two views can be displayed by selecting Window → Show View → Other... The two views find themselves under the category “Quality Assessor”. Klick OK to confirm. For a better illustration see Figure 6.3.

Figure 6.3: To show the visualization select Window → Show View → Other... Select the views under the category “Quality Assessor”. Click OK to confirm.

The metrics used in our tool deliver values of different ranges (see Figure 6.4 on the top). So we had to normalize them. Again we followed the proposal of Bansiya [BD02] and selected a release as a basis of comparison. Than we divided every value through its correspondent basis. The results are displayed on the bottom left. The basis can be selected dynamically by the check box at the left side. This allows a better analysis of the project and a better comparison at a specific point in time. The last step consists of the calculation of the quality attributes. There the equitations already mentioned in Table 5.3 come into account. This step corresponds to the step $L_{12}$ in Figure 5.1. Each equitation is filled by the correspondig value of the table at the bottom left. The results are then presented in the table at the bottom right. This table contains the distilled quality informations of the design evolution of the “Azureus” project.

The “Quality Assessor - Graph” view visualizes the values of the quality attribute table (Bottom right Table in Figure 6.4). The ordinate shows the calculated attribute values for each quality attribute and the axis of abscissae the different software releases. The result looks like shown in Figure 6.5. An interpretation of the graph follows later on.

6.4 Case Study

We applied the Quality Assessor Tool on the medium sized Java software “Azureus”. We analyzed the evolution of its design quality over a period of three years by measuring 19 releases.
Figure 6.4: Table View of the Quality Assessor Tool. It is implemented as an eclipse view component. The table at the top shows the absolute values of the design properties. At the bottom the normalized values (left) and the values of the quality attributes (right) are displayed.

During this time the “Azureus” volume of code has grown by factor ten. This means from 22'000 lines of code to 222'000 lines.

The workflow of the Quality Assessor Tool is shown in Figure 6.6. For this case study we checked out 19 releases from the CVS repository of “Azureus” [Azubi06a] and imported them into the Eclipse IDE. Because “Metrics 1.3.6” is only able to calculate the metrics properly when the project compiles, we made all releases running. After compiling a release we exported the measurement results of the source code into an XML-file. To export files “Metrics 1.3.6” provides several options, a hierarchical one and a flat one. Our SAX-Parser was written for the syntax of the flat format. However we exported the results of every release.

We added the different 19 XML-file by using the preference page shown in Figure 6.2. After clicking “Ok” on the preference page the files became parsed and the “QualityModel” was built. Then we opened the two views (“QualityAssessor — Table” and “QualityAssessor — Graph”) to have a look at the results.

6.5 Results

In this section we analyze the graph shown in Figure 6.5. This graph was the result of the process described in the section before. We can recognize a stepwise increase of the quality attributes reusability, flexibility, functionality, extendability, and effectiveness. On the other hand understandability decreases. The evolution of software is affected by changes. We arrange them in
three categories: new features, changes, and bug fixes. The most dramatic change for the structure of a design is the introduction of a new major feature. Normally, after such an introduction, a phase will follow where the main focus lies on fixing bugs and making changes. A deeper knowledge of the project is necessary in order to prove this. It is necessary to take into account, what has been done between the different releases and how they influenced the metrics. We took the changelog of the project and related it to the result of our model. We wanted to see if there are any parallels, given the assumption that new features effect the design more than other changes. We only took the important changes into account which are mentioned on "Azureus" [Azu06b].

We counted the number of new features, changes, and bug fixes listed on the website. A plot can be seen in Figure 6.7.

In the Figure 6.7 and 6.8 we marked corresponding sequences. Indeed there are parallels between these categories.

---

1 refactorings, adoption to new environments

---

![Figure 6.5: Graph View of the Quality Assessor Tool. The ordinate shows the calculated values and axis of abscissae the different software releases.](image)

**Figure 6.6:** Quality Assessor Tool workflow.
between the kind of changes and their influence on the total quality. The red marked boxes point to active phases whereas the blue ones point more to recreation phases. In this phase the focus lies on correcting bugs or refactoring of the code. However this phases should be used to improve the design quality.

![Count of changes of the Azureus project.](image)

During the product phase of the first red box from the left an extension of functionality took place (from release 2.0.4.2 to 2.0.6.0). During the following releases (2.0.7.0 → 2.0.8.0) and the number of bug fixes rose relative to the number of new features. The following blue phase is a time when not much was going on in the project.

The second red box represents a major release with many changes. This major release is also visible in an increase in the quality attribute graph. The major release phase has two steps whereas we can recognize that the first one (2.1.0.0) has more new features and the second one (2.1.0.2) has more bug fixes. Release 2.1.0.4 concludes the active phase.

In the green box we can see that the total number of changes is high but the quality graph does not react immediately (2.1.0.4 → 2.2.0.0 → 2.2.0.2). During this time the number of changes and bug fixes in the program are higher then in all other releases. During this time “Azureus” had to adjust to the new Java environment 1.5. Of course during this time new features were introduced but the effect they normally would have are compensated by the changes. The increase from release 2.2.0.4 to 2.3.0.0. is another major release but this time the increase in the attribute graph is higher because the effect new features have is not that much neutralized by changes and bug fixes.

The last red box shows an active phase again. In release 2.3.0.6 we have an introduction of a number of new features relative to release 2.4.0.0. But the difference between the quality attribute values is more significant. Again a deeper knowledge of the project is needed. The features introduced by release 2.4.0.0 must have a much bigger influence on the structure of the design. Probably they have been major features. Release 2.4.0.2 at last is then again a release of recreation and does not have a big influence on the design structure.
6.6 Discussion

One can say in general that active phases have much more impact on the quality of a design but we have also seen that a changelog by itself cannot provide enough information to fully explain the evolution of the quality attributes. For example, it cannot be deduced that a bug corrected in one release is an immediate consequence of the changes or newly introduced features of the release before. It is also not clear if a change made in a release is made to enhance the quality of the design or it is made as an adjustment to a change in the environment. A clear assignment of cause and effect is difficult and requires a deeper knowledge of the changes and their effective influence of the code.

As one can see, the reusability increases the most but functionality and flexibility or extendability and effectiveness increase, too. Understandability on the other hand decreases. This is intuitively clear because the larger the program the more difficult it becomes to understand. But what does it mean when we look at reusability. Corresponding to the definition in Table 5.1 reusability means the ability of the software to be reapplied to a new problem without significant effort. If we have a closer look at how this attribute is influenced one can recognize that a growing software like “Azureus” must have an increasing reusability in this sense because the attribute is highly influenced by the measurement of the design size and the messaging. So the question arises what does an increasing curve of reusability mean. Further case studies, wherein the observed software becomes refactored are required to show that quantitative statements can be drawn from the quality model.
Chapter 7

Conclusion & Future Work

7.1 Conclusion

In this thesis we investigated several existing quality models, e.g., McCall, Beohm, ISO 9126 etc. and related them to multiple dimensions. We focused on the possibilities to assess general quality with source code metric based quality models (SMQM). We realized that a pure SMQM does not offer enough informations to make a general quality assessment as desired from high level quality models. This will still be something elusive. The main reason why SMQM are not capable to express general quality is that the source code itself does not contain semantical enough information about the software, i.e., it is not possible to assess the degree software covers the requirements with source code metrics.

In fact a SMQM is limited to an architect/developer point of view and is only able to reflect technical insides of a certain product. The code contains informations about itself. So the perspective of a SMQM is mostly predetermined. Parameter values can be set for other dimensions. The point of time in the development when the code quality become assessed is dependent on the availability of source code. The question of comparability must also be posed and allows several possible options.

Given these dimension we adapted the QMOOD of Bansiya[BD02] to implement a SMQM for the Java programming language. We found out that changes in the source code of a project leads to changes of different magnitude in the quality attributes but a clear assignment of cause and effect is not obvious.

Models based on source code metrics have the potential to evaluate a software design, either its evolution or in direct comparison with other designs in the same product domain. Our Quality Assessor Tool makes us capable to track the quality evolution of a design over several releases and allows to get important information about the inner life of the product. This information can support design decisions on higher abstraction levels, i.e., if it is necessary to take action for improving the design of a specific module.

Another important advantage beside the objective way to look at software, a source code metric based approach allows to gather information automatically. This advantage will become more and more important since the size of software explodes.

However, we found out that the question about the global quality of a product can not be completely answered by a SMQM. This is mainly caused by the lack of requirement information, which can not be extracted from the source code only. Additional inputs which cover this issue are needed to reach a higher level of quality assessment.
7.2 Future Work

The field of software quality measurement is still a fresh discipline. Neither exists a general definition of what software quality really means, nor a well defined basis of measures. We have started the quality assessment from the bottom, i.e., the source code metrics, and formed a SMQM. We state that this SMQM has to be extended by accordingly quality models to assess other quality aspects of a product i.e., a requirement based quality model which allows to relate the code with its requirements.

7.2.1 Source Code Metric Based Quality Modeling

We conclude that source code metric based models can provide important information for design decisions on higher level of abstraction. But the benefit of this information stands and falls with the validity and expressiveness of the underlying metrics, metric combination and definition of the quality properties/attributes. There is not yet a general accepted methodology available to validate metrics because of their empirical character. Also the weights of the different used formulas from Table 5.3 are not proved. Further case studies are needed to adjust them.

In object oriented programming there are several ways to design/implement the same functionality. Each option has advantages and disadvantages. A SMQM can help to find a decision but to do so the pros and contras have to be well ranked. Further knowledge of best design practice and metric validity has to be accumulated in order to allow this, i.e., a clear specification/definition what a good design is, enables a more concrete measurement of quality attributes.

7.2.2 Quality Assessor Tool

The Quality Assessor Tool implements the basic functionality to analyze source code with a SMQM. There are a number of features which can improve the tool.

- Importers for other metric data sources. This feature makes the information base broader and allows the user to replace suboptimal metrics by better, more valid ones.

- The weights to calculate the formulas at the moment are hard coded. But this weights are everything else but fix. With an increasing number of case studies they have to be adjusted to new cognitions. A flexible setting will address this issue.

- Up to now only a comparison on the system level is possible. A breakdown of the results to module or package level will bring more information how the different units changes.

- Introduction of new design properties/attributes and metrics. A replacement or redefinition of the quality design properties/attributes can lead to more expressive ones.
Appendix A

Contents of CD-Rom

On the enclosed CD-Rom one can find the following contents:

• Abstract.pdf
• Zusammenfassung.pdf
• Thesis.pdf
• Eclipse Plugin: ch.unizh.ifi.qualityAssesor
• XML-Database of the measured releases of the “Azureus”-Project.
References


