

Static Software Analysis for Safety-Critical Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Franz-Josef Katzdobler

Matrikelnummer 0425195

an der
Fakultät für Informatik der Technischen Universität Wien
ausgeführt am Institut für Computertechnik

Betreuung
Betreuer: O. Univ. Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
Mitwirkung: Dipl.-Ing. Dr.techn. Andreas Gerstinger

Wien,

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung

Franz-Josef Katzdobler
Hagenmüllergasse 33/303
1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift)

Kurzfassung

Softwaremetriken waren in den letzten Jahrzehnten durchgehend ein Thema zur Unterstützung in der Abwicklung von Softwareprojekten. Viele verschiedene und teils widersprüchliche Ergebnisse liegen vor. Einige Studien kommen zu dem Ergebnis, dass aus statischer Softwareanalyse auf die Qualität eines Softwareproduktes geschlossen werden kann. Andere hingegen behaupten das genaue Gegenteil. Diese Arbeit konzentriert sich vor allem auf den Aspekt der Wartung im Bereich von sicherheitskritischen Systemen. Es ist belegt, dass bis zu 80 Prozent der Gesamtkosten eines Projektes auf Wartung zurückzuführen sind. Unterschiede zwischen verschiedenen Paradigmen und Programmiersprachen wurden untersucht und Metriken auf die praktische Anwendbarkeit geprüft. Dafür wurden Daten von erfahrenen Programmierern im Umfeld von sicherheitskritischen Systemen gesammelt und auf Korrelation mit den Daten von statischen Softwareanalysetools untersucht, wobei Wartbarkeitsmodelle auf der Basis des ISO 9126 Standards definiert wurden. Außerdem erfolgte eine Implementierung eines Prototypensystems, das es ermöglicht, Metriken aus mehreren Tools zu benutzerdefinierten Wartbarkeitsmodellen zu kombinieren. In den Ergebnissen werden zwei Wartbarkeitsmodelle präsentiert, wobei eine Trennung von prozeduralen und objektorientierten Sprachen eingeführt wird. Außerdem zeigen die Experimente, dass der Wartbarkeitsbegriff zwar teils unterschiedlich interpretiert wird, es sich aber doch ein Konsensus zwischen den Entwicklern bilden lässt. Zusammen mit dem entwickelten Prototypensystem, durch das die Messung und Aggregation der benötigten Metriken erleichtert wird, zeigt diese Erkenntnis die Sinnhaftigkeit des Einsatzes von Metriken in der Softwareentwicklung.

Abstract

With the aim of supporting the software development life cycle, software metrics have been subject of research for many years. Different and contradictory results were obtained concerning the correlation between the measured properties and the quality of a software product. As maintainability is an important criterion and contributes up to 80 percent to the software development cost, this work focuses on this property in the field of safety-critical systems. Essential differences between paradigms and languages were considered and metrics were examined to be useful or not in this area. The work was validated by collecting data from experienced programmers and correlating them with the data from the static measurement process performed by a tool. Models based on the ISO 9126 standard were constructed for this purpose. Furthermore, a prototype system was implemented which allows combining metrics from different tools to user defined maintainability models. The experiments and results show that the individual developers share a slightly different notion of maintainability. Furthermore, the results show that it is possible and necessary to establish consensus about the maintainability notion. Together with the implemented prototype system which supports the measurement and aggregation of the necessary metrics, this finding demonstrates the usefulness of applying metrics in software development.

Acknowledgements

I am very grateful for the cooperation and interest of my supervisor and the participating developers from the industrial field. It would not have been possible to achieve the results without their help. I am particularly grateful to my family, friends and study colleagues for their support and inspiration during my years of study. The great community and conditions in the dormitory *Salesianum Don Bosco* did a huge contribution to the successful time in Vienna as well. Further thanks go to the *University of Coimbra*, where I could spend one semester abroad and got the basic idea to work with the topic of this thesis.

Table of Contents

1	Introduction	1
1.1	Notions of Quality and Maintainability	2
1.2	The Importance of Measuring Maintainability	4
2	Related Work	6
2.1	Motivating Results	6
2.2	Contradictory Results	9
2.3	Problem Statement	10
3	Software Metrics	11
3.1	Theory of Measurement	11
3.1.1	Fundamentals	11
3.1.2	Representational Theory	12
3.1.3	The Impossibility of a Single Number for Complexity	14
3.2	Weyuker Properties	15
3.3	Differences between Languages and Paradigms	17
3.3.1	The Procedural Paradigm	17
3.3.2	The Object Oriented Paradigm	21
3.4	Metrics for Maintainability	24
3.4.1	How to Measure Maintainability?	24
3.4.2	A Maintainability Model for the Procedural Paradigm	28
3.4.3	A Maintainability Model for the OO Paradigm	28
3.5	Comparison of the Approaches	29
4	Integrating and Selecting a Software Metrics Tool	31
4.1	Perspectives	31
4.2	Defense Position of Developers	31
4.3	Aspects of Analysis Tools	33
4.3.1	General Architecture	33
4.3.2	Three Key Criteria for Success	33
4.3.3	Different Conclusions with Different Tools	35
4.4	Selected Tools	35
4.4.1	Resource Standard Metrics (RSM)	35
4.4.2	OOMeter	36
4.5	The Implemented System	36

4.5.1	Architecture	37
4.5.2	Modeling the DBS	42
4.5.3	Integration of Existing Tools in a Portal	43
5	Experiments	47
5.1	Validating Software Measures	47
5.2	Weights Assignment to the Proposed Models	47
5.3	The Analytical Hierarchy Process in Detail	48
5.4	Validation of the Procedural Model	48
5.4.1	Description of the Source Code Modules	50
5.4.2	Criticism of the Experiments	51
6	Results	52
6.1	Weights for the Procedural Model	52
6.2	Defining Thresholds for the Procedural Model	52
6.3	Weights for the OO Model	55
6.4	Defining Thresholds for the OO Model	56
6.5	The Weights for the ISO 9126 Characteristics	58
6.6	Criticism of the Former Results	58
6.7	Results of Validating the Procedural Model	58
6.7.1	Functional View	58
6.7.2	The Overall View	61
6.8	Agreement and Disagreement of Developers	61
7	Conclusion	65
7.1	Future Work	66
7.2	Back to the Problem Statement	68
7.3	Discussion	68
	Literature	70
	Internet References	73
	A Appendices	74

Abbreviations

AHC	Analytic Hierarchy Process
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CBO	Coupling Between Object classes
CC	Cyclomatic Complexity
CLOM-CK	Lack Of Cohesion of Methods
CRUD	Create, Read, Update, Delete
DBS	Data Base System
DIT	Depth Of Inheritance Tree
DOC	Comment Frequency
DOM	Document Object Model
DRY	Don't Repeat Yourself
EJB	Enterprise Java Beans
GUI	Graphical User Interface
IDE	Integrated Development Environment
ISO	International Standards Organization
Java EE	Java Enterprise Edition
JPA	Java Persistence API
JSC	Johnson Space Center
JSF	Java Server Faces
LOC	Lines Of Code
MI	Maintainability Index
ML	Machine Learning
ND	Nesting Depth
NOC	Number Of Children
NOI	Number Of Input Parameters
NOM	Number Of Methods
NOMV	Number Of Member Variables
NOR	Number Of Return Points
OO	Object Oriented
ORM	Object Relational Mapping
QBL	Quality Benchmark Level
RSM	Resource Standard Metrics
SAX	Simple API for XML
UML	Unified Modeling Language
W3C	World Wide Web Consortium
WMC	Weighted Methods per Class
XML	Extensible Markup Language

1 Introduction

Software development plays a central role in the delivery and application of information technology. Therefore, managers try to improve the process in the software development area. Several approaches are possible, Object Orientation (OO) is one of the most prominent examples. With the focus on process improvement also software metrics gained increasing interest and are subject of research for many years. Any software metric is an attempt to measure or predict attributes of a software product. In this work, the focus is on the attributes of the source code. More formally, a software metric can be defined as a mathematical function mapping the entities of a software system to numeric metric values [LLL08].

Considering the different phases of software development, software measurement can take place in all of them. The earlier measurement is applied, the more useful it is. The problem is that in the earlier phases, measurement is difficult and most measures are defined for the coding phase. In this work, the focus is on the source code metrics, addressing the implementation phase. Even so, measurement can also be successfully applied in the phases requirements and analysis, specification and design.

Contradictory results were obtained by different studies. While some authors claim that there is a correlation between the internal measured software metrics and external quality attributes, others state the opposite. It is undoubted that to confirm the results of the studies, more data from the industrial field have to be collected. This is one ambition of this thesis. Another interesting point is the difference between different paradigms. In the procedural approach, the focus has to be set on different properties than in the OO one. More specifically, differences between programming languages should be considered. In this work, the experiments are implemented in a safety-critical context with data obtained from the company Frequentis [4].

As maintainability is an important criterion and maintenance contributes up to 80 percent to the software development cost, it would be helpful to point out which metrics are especially useful to determine the maintainability of a software product. The fact that software is subject to continuing change was realized already in the 1970's, when Lehman formulated the laws of program evolution [Leh80]:

1. *Continuing Change*

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version.

2. *Increasing Complexity*

As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.

3. *The Fundamental Law of Program Evolution*

Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariance.

4. *Conservation of Organizational Stability (Invariant Work Rate)*

During the active life of a program the global activity rate in a programming project is statistically invariant.

5. *Conservation of Familiarity (Perceived Complexity)*

During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

The first law originally expressed that large programs are never completed. The message of the second law should be intuitively clear. The third law describes the observed fact that the number of decisions driving the process evolution (feedback paths, human interaction, . . .) all combine to yield a statistically regular behavior. Laws number four and five describe the management of well-established organizations. They try to avoid dramatic change. The number of people involved and the time delays in implementing decisions all work together to prevent drastic change [Leh80].

The first two laws confirm that actions have to be taken to assure and measure maintainability. To construct a model fulfilling this purpose is not a trivial task. What is missing are reasonable values for the weights of the metrics in use. To solve this problem, an experiment where subjective opinions of expert developers are correlated with a maintainability model containing proposed software metrics is performed in this thesis.

1.1 Notions of Quality and Maintainability

Many different definitions can be found to define the quality of software. The International Standards Organization (ISO) gives the following one [11]:

“The totality of features and characteristics of a product or service that bear on its ability to satisfy specified or implied needs.”

From the IEEE a different one is given [IEE90]:

“The degree to which a system, component, or process meets specified requirements and customer or user needs or expectations.”

The problem when evaluating the quality of software is that different developers and managers share different notions about quality. The ISO 9126 standard is an attempt to address this problem. As depicted in Figure 1.1, the standards gives six main characteristics.

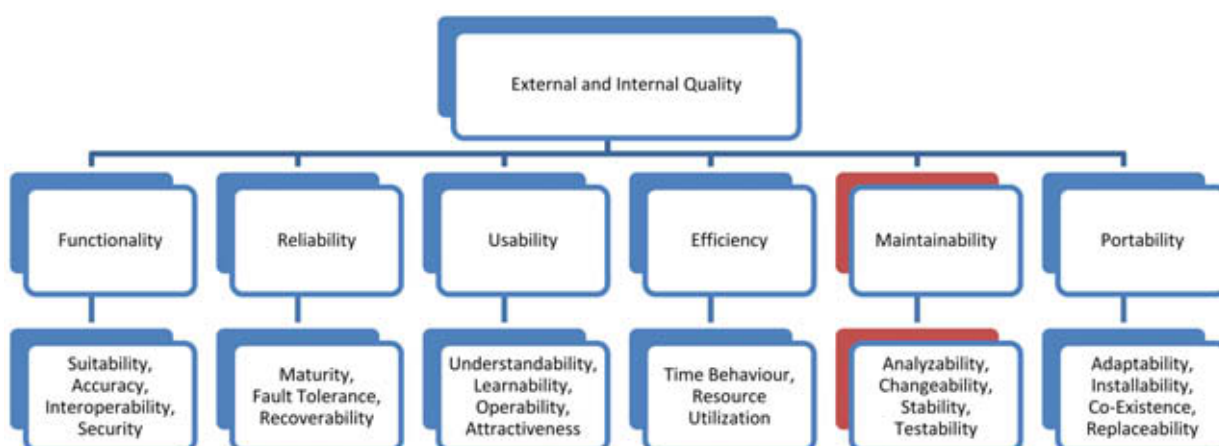


Figure 1.1: ISO 9126 Quality Model

These are further divided into sub characteristics. Note that inside the sub characteristics the compliance property is not stated here. Compliance refers to certain industry (or government) laws and guidelines. This topic is not covered by this thesis.

The focus in this work is on the maintainability characteristic, which consists of the following listed properties and is highlighted in Figure 1.2.

- Analyzability describes how easy or difficult it is to diagnose the system for deficiencies or to identify the parts that have to be modified.
- Changeability describes how easy it is to adapt a system.
- Stability describes the possibility to keep the system in a consistent state during modification.
- Testability describes how easy the system can be tested after and during the modification.

The idea is to measure each one of the sub-characteristics with metrics obtained from the source code. The standard is not undisputed [AKCK05]. Especially the absence of exact advices which metrics should be used with which weights is a weakness that has to be solved. As will be shown in this work, this is not a trivial task and different attempts can be found to attack this problem.

To distinguish between *Maintenance* and *Maintainability* and to avoid confusion, the IEEE standard definitions are listed here [IEE93].

“Maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to changed environment.”

“Maintainability is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.”

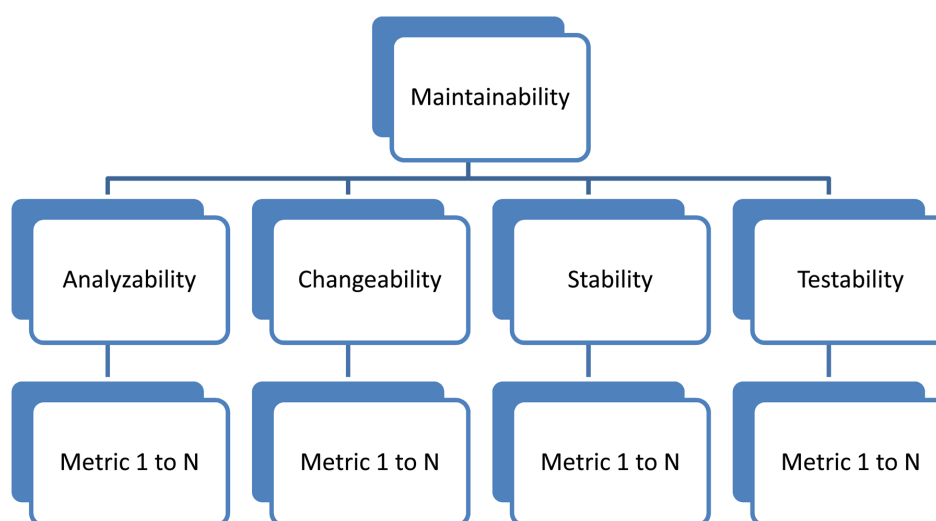


Figure 1.2: ISO 9126 Maintainability Hierarchy

1.2 The Importance of Measuring Maintainability

Estimations show that up to 80% of the overall software development costs are related to maintainability activities [PO95, NAS04]. To improve the maintenance process, it is necessary to be able to measure software maintainability. Many metrics exist for this task. The problem is that most of them are hard to apply in practice. Therefore, even if the importance of maintainability is identified, it is a challenging task to find and establish a measurement tool that supports the software development group. Furthermore, improving maintainability means effort and cost. This simple fact also prevents companies from investing in actions to improve the maintainability of their software.

As the benefits of maintainability improvement actions have no immediate effect, it is hard to convince developers of their importance. The proper recognition of the cost benefits is a precondition to establish metrics for measuring maintainability. The author of [Mun81] already discovered the problem in 1981:

“In this age of automation, we seem to be waiting for maintainability to be automatically produced from our automated systems. It will in fact come about only as a result of the proper recognition of the cost benefits and the successful implementation of these first elementary efforts to achieve it.”

The cost benefit is not subject of this thesis, many literature about this topic exists. But what is shown is that maintainability can be measured in an understandable, inexpensive way.

After these introductory aspects and before studying related work, a short overview about the organization is given. After the introduction in Chapter 1, related work and some contradictions are considered in Chapter 2. Chapter 3 gives an overview about the most important software metrics for the procedural and the object oriented (OO) paradigm. Also the question how to measure maintainability is examined in detail. Chapter 4 considers different aspects concerning

the selection of a software metrics tool and examines if there are significant tool dependent differences in the measurement results. Furthermore, the implemented prototype system is introduced. The experiments are described in Chapter 5. Together with the presentation of the results in Chapter 6 this is probably the most valuable work as the collected data were obtained from the industrial field in a safety-critical environment. The work is summarized in Chapter 7, where ideas for future work are given as well.

2 Related Work

After the introduction in the preceding chapter, some relevant findings related to the topic of this thesis are considered. There is a vast amount of literature available. The results are contradictory and need to be examined carefully.

2.1 Motivating Results

A study with respect to software metric tools can be found in [LLL08]. Two important questions are answered:

1. Given the same metric and the same input, will different tools calculate different metric values?
2. If the metric values are tool dependent, do they lead to different conclusions or are they irrelevant inaccuracies?

The result shows that different tools differ in their results for the same measured metric, meaning that depending on the used tool, different (possibly wrong) conclusions will occur. This result is important as it shows that the tool selection process is not trivial and automatically produced metric values need to be considered critically.

Another important aspect in [LLL08] is the proposal of a maintainability model based on the ISO 9126 standard. The model makes it possible to create a ranking of several software classes with respect to maintainability. This approach will be found in this work as well.

In [Kie06] an experiment with two safety critical and two standard applications was performed. The tool in use was *CTM++* which can be used for C/C++. The authors conclude that there are significant differences in the metrics between safety critical and standard applications and therefore the proposed metrics can be used to evaluate the quality of software during the development life cycle. Considering this work, some questions arise. In the comparison of the metrics a maintainability index appears, which is almost equal for the safety critical and the standard applications (see Figure 2.1). But there is no explanation how this value is calculated or derived, although this is important to interpret the value. It can just be assumed that this value represents the Maintainability Index (MI) that will be examined in Section 3.4.1.1. Furthermore, the definitions of mental effort and program difficulty level are complex. This can lead to problems

as will be described in Section 3.4.1.2. Another weakness might be the statistical significance, as only two applications of the different fields were considered.

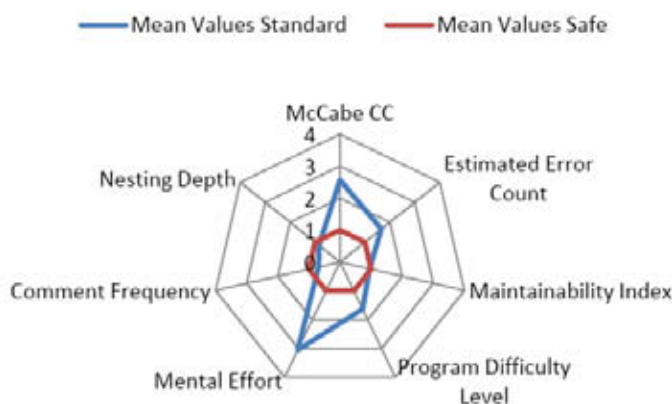


Figure 2.1: Results obtained by [Kie06], Kiviati graph reproduced by given data

Chahal and Singh [CS09] propose metrics to study symptoms of bad software design. Basically they use the metric suite introduced by Chidamber and Kemerer in 1994 [CK94] and give correlation coefficients for the different metrics. The idea is that once these coefficients are calculated for a high quality OO design, these reference values can be used to find design mistakes in other investigated projects.

The state of metrics in software industry is examined in [OH08]. Practices in the organizations HP, Motorola, NASA and Boeing are considered. The conclusion from the authors is that software measurement is lacking behind, although the importance of metrics is known since the early 1970's. The discussed experiences indicate that

1. Software metrics can be used at each phase in the development life cycle.
2. Metrics can be used to find problems that can be potential sources for errors in the system. Discovering them helps to decrease development cost.
3. Applying software metrics need not to be a time consuming effort.

Especially interesting in this work are the practices from NASA. A tool was developed by the NASA Software Assurance Technology Center which helps to determine error-prone modules using measures for complexity, size and modularity of source code. They found that large modules with the highest complexity tend to have the lowest reliability. Moreover, considering the maintainability, these modules are difficult to change or modify. They also declare that the metrics in use for quality analysis lack industry guidelines. NASA developed a metrics data program, named IV&V. This programs assists in gathering, verifying, sorting out, storing and distributing software metrics data. Additional to that, NASA offers project non-specific data available in its repository [2]. The association between the error data and metrics data in this database gives the software community the possibility to investigate the relationship of metrics and the quality of a software product. Note that the repository data is available at no cost. The only drawback of this repository is that the source code is not available. This would have been of special interest in the experiments part.

The authors of [SSM06] suggest a model for code quality management based on ISO 9126. Many interesting ideas are proposed. The basic idea is the introduction of Quality Benchmark Levels (QBL). The higher the level, the more indicators are considered. Moreover, a cost-benefit analysis is incorporated and the idea of applying thresholds (the higher the QBL, the stricter the threshold) is given. The reference values for these thresholds come from a repository with more than 120 industrial projects which were analyzed. The aim of the QBL is to capture the overall quality notion. This is not a trivial task, as many contrary aspects have to be considered. Therefore different benchmark levels seem to offer a possibility to overcome this problem. Note that in this thesis, the focus is on maintainability and not on capturing the overall quality notion. Figure 2.2 depicts the model, which can support in the following situations:

- Quality assurance in outsourced projects
- Acquisitions where source code has to be evaluated
- Product comparison

Especially outsourced projects are risky to fail. Estimations show that up to 50% of all outsourcing projects are not completed successfully. The reason in most cases is the absence of proper quality in the source code delivered. Therefore, the effort to maintain the product in-house afterwards is higher than the estimated one. The QBL model offers a tool to prevent this scenario.

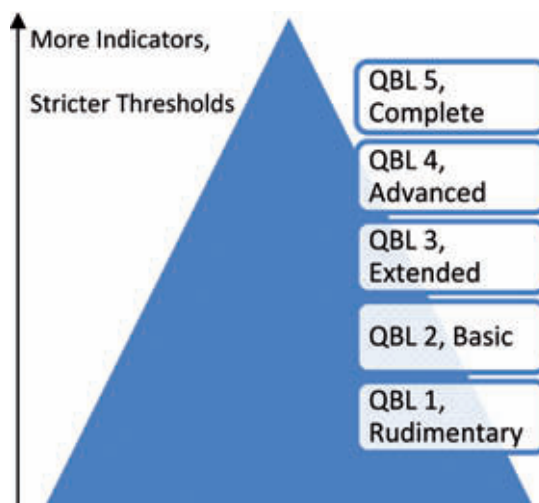


Figure 2.2: Quality Benchmark Levels proposed in [SSM06]

A necessary scientific basis for software measurement is provided by [Fen94]. The most valuable work is the proof of the theorem, that the perceived subjective complexity of a program can not be expressed with a single real number. This question was subject of research for many years. Furthermore, typical pitfalls are examined that can lead to invalid results and conclusions. Fenton's work can help to avoid common mistakes. Many experiments can be found where the expertise of [Fen94] is simply ignored.

2.2 Contradictory Results

In [MR07] the correlation between internal software metrics and dependability metrics was examined. Therefore a collection of C/C++ programs was used, extracted from the programming competition *Online Judge* (see [6]). The conclusion of the experiment is that there is no correlation between the examined internal metrics and the introduced dependability metrics. This is a surprising conclusion as software metrics were used to support the overall development life cycle of software products over the last decades. Therefore it should be explained why the results obtained in [MR07] are ambiguous.

As a first reason it should be mentioned that the languages C and C++ should be investigated separately. In the object oriented (OO) paradigm (C++) different considerations have to be made than in the procedural one (C). Although the statement *mixing programming languages in this research might invalidate the results, or at least complicate their interpretation* is given by the authors, the research is done for C and C++ programs together.

A second weakness in the investigation is that the used programs were produced mainly by non-professional programmers. Therefore, it cannot be inferred that these results are also valid in the industrial field.

As a last point of criticism, the evaluation of the number of defects in a program should be mentioned. The method is defined in a way, that the first correct (as defined by the authors) submission of an author has no defects and for every prior submission of the same author the number of defects is increased by one. This definition misses scientific background.

A different investigation in safety related context for C# was performed in [Ger07]. The opinion of expert developers was correlated with metrics collected by an analysis tool. The danger of this approach is that opinions are subjective. The obtained results are that there is significant correlation between

- Readability and Comment Density
- Code Clarity and Quality Notices ¹
- Code Clarity and Return Points

whereas no significant correlation could be found for

- Internal Quality and Cyclomatic Complexity
- Internal Quality and Method Size
- Internal Quality and File Size

The most surprising fact here is that there is no correlation between internal quality and the Cyclomatic Complexity (CC). Usually the CC is considered to be one of the most influencing factors on maintainability. Therefore it would be interesting to confirm this result for other programming languages and for a larger population of programs. The cause for the absence of correlation between internal quality and file size can be explained by the use of modern development environments. Modern tools allow to open or close complete methods in one file, yielding to a better overview even in large files.

¹Quality Notices are violations against established programming rules

2.3 Problem Statement

The aim of this thesis is to develop an applicable model to capture the property maintainability of a software system. Key challenges are to choose convenient metrics and to weigh them. The model will be specified for different paradigms and in the field of safety critical systems. Furthermore, experiments are needed to prove the benefit when introducing metrics in the software development life cycle. Also, a prototype has to be implemented which allows the definition of a user specific maintainability model. The metrics to evaluate the model are delivered by several standard tools and are processed by parsers to fill a database.

3 Software Metrics

Before applying measurement, it is necessary to identify an *entity* and a specific *attribute* of it. A lack of clear definitions can lead to misinterpretation and different meaning for different people. Therefore in this chapter the metrics used later on are defined.

Basically the entities that can be considered in software measurement are the process, the product and the resource. A process is any activity related to software development, a product any artifact produced and resources can be hardware, people or software itself that is needed for the processes.

If the attributes of an entity are considered, it can be distinguished between the *internal* and the *external* ones. An internal attribute can be measured only based on the entity and is referred to as direct measure. An external attribute can be measured by incorporating the relation to the environment and is therefore referred to as indirect measure. As an example for an external attribute think of the reliability of a program, which not only depends on the program itself. Also the compiler, the hardware, in a nutshell, the overall system is influencing this attribute. Whereas size is an internal attribute that can be applied to any software document.

In the next part of the work a necessary scientific basis for software measurement given by [Fen94] is considered. Although this work helps to prevent common pitfalls, measurement theory is often neglected in software industry when applying metrics. A survey and discussion of software metrics incorporating software measurement theory can be found in [Rig96].

3.1 Theory of Measurement

The science of measurement is relevant to software metrics. To evaluate proposed metrics, fundamental notions and theories are of importance. In software industry, the general notions of measurement theory are generally not well understood. This can lead to wrong conclusions when experiments are carried out, which do not follow the concepts of measurement theory.

3.1.1 Fundamentals

As a first step the term measurement itself needs to be defined [Fen94].

“*Measurement* is defined as the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.”

The terms attribute and entity were already introduced. An important question in this definition is what is meant by *the numerical assignment describing the attribute*. This is where *Representational Theory* comes into play that will be examined below, making this statement more precise. “Informally, the assignment must preserve any intuitive and empirical observations about the attributes and entities” [Fen94]. If the height of humans is considered, bigger numbers are assigned to taller people. The height of human is well understood. But even so, there may be different intuitive meanings. Whereas some people would say that hair height should be included, others would not agree with that. Therefore a *model* needs to be defined, reflecting a specific viewpoint. The definition of models is especially of importance in the software community. As will be seen, even for the simplest code metric Lines of Code (LOC), this can be a problem. In this particular case there is no consensus if comment lines should be counted or not for example.

3.1.2 Representational Theory

The questions that should be answered by measurement theory are the following ones [Fen94]:

- What is and what is not measurement?
- Which types of attributes can and can not be measured?
- Which type of scale can be used?
- How to define the measurement scale?
- How do we know if we have really measured an attribute?

To answer these questions, further terms are introduced. The first one is the *Empirical Relation System* (C, R) where C is the set of entities and R is the set of empirical relations. The empirical relations are formed by the intuitive understanding of the attribute. If the attribute height of humans is considered again, the empirical relations could be *is tall, taller than, ...*

The next term considered is the *Representation Condition*. A mapping M into a numerical system is needed to measure the attribute that is characterized by (C, R) . The numerical relation system is denoted with (N, P) . Entities in C are mapped to numbers in N and relations in R are mapped to numerical relations in P . The most important fact here is that the mapping has to be performed in such a way, that all empirical relations are preserved. This is called the *Representation Condition*, the mapping M itself is called the representation. Note that the representation condition is two way. Formally, if M maps the binary relation \prec to the numerical relation $<$, then $x \prec y \Leftrightarrow M(x) < M(y)$.

Identifying the empirical relations in advance is an important step in software metrics work. This avoids the definition of poorly understood numerical assignments. A classical example is the definition of the complexity of a program with the cyclomatic complexity of McCabe. In the next subsection it is highlighted that it is impossible to define a single number for the complexity of a program.

Other important aspects to consider are the *scale type* and the *meaningfulness*. In general there may exist many ways of assigning numbers satisfying the representation condition. This can be seen by the wealth of software metric definitions and it seems that the creativity in creating new ones is unlimited. Considering the height of people and two persons denoted with A and B , if person A is taller than person B , then $M(A) > M(B)$. This is independent from the unit used for the measure M , which could be feet, inches or centimeters. It can be seen that there are many different measurement representations for the normal empirical relation system for the attribute of height of people. For this special case it can be stated that any two representations M and M' can be related to each other in a specific way. By introducing a constant $c > 0$ one can state $M = cM'$. This transformation is called an *admissible transformation*. If the scale type for an attribute has to be defined, it is the class of admissible transformations that has to be examined. For the height of people, every admissible transformation is a scalar multiplication. This special scale type is called *ratio* and describes a very rich empirical relation system.

A common pitfall is to assume a ratio type a priori. For many software attributes there is no exact or a very crude definition of the empirical relation system. Considering a classification of software failures depending on their criticality, different classes of failures and a binary relation *is more critical than* could be introduced. If any two representations are related by a monotonically increasing transformation like in this example, the class of admissible transformations forms an *ordinal* scale type. This type is the one of interest in this thesis. Later on, statements like *function A is more maintainable than function B* referring to software functions should be possible. The best known scale types according to [Fen94] in order of sophistication illustrated in Figure 3.1 are:

- Nominal
- Ordinal
- Interval
- Ratio
- Absolute

Defining *meaningfulness* may sound extraordinary. Formally, in measurement a statement is meaningful if under any admissible transformation its truth or falsity remains. Referring to the human height example, it can be meaningful to say a person is twice as tall as another person. It does not matter if the height is measured in centimeters or in inches. This is not valid for the classification of software failures depending on their criticality. The statement *Failure A is twice as critical as Failure B* is not meaningful if an underlying ordinal scale is assumed. This is because one could define a measure M with $M(A) = 2$ and $M(B) = 1$ and another measure M' with $M'(A) = 10$ and $M'(B) = 7$. Under M the given statement would be true whereas under M' the statement is false.

The definition of meaningfulness is also useful to find out which kind of operations can be used on a measure. Let's consider as an example the mean of a data set. If the data set was measured on a ratio scale the operation makes sense. But if the data set was measured on an ordinal scale the median is the correct operation. A common pitfall is to use the average instead of the median for data which is only ordinal.

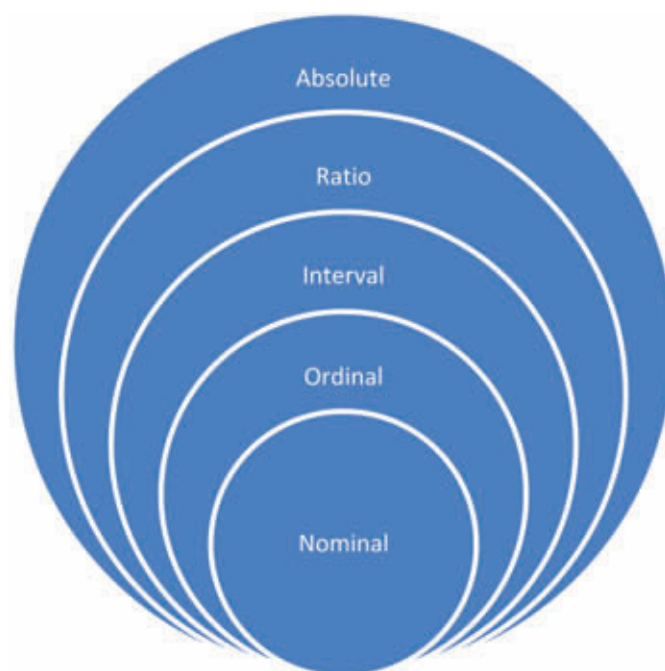


Figure 3.1: Scale Types and Sophistication

As a last point of measurement theory it should be mentioned that the formal mathematical background is the basis to find theorems which assert conditions under which a certain scale is applicable for a certain relation system. To give an example, *Cantor's Theorem* states conditions for real-valued ordinal-scale measurement assuming a countable set of entities in C and a binary relation b on C :

The empirical relation system (C, b) has a representation in $(R, <)$ if and only if b is a strict weak order. The scale type is ordinal if such a representation exists. The meaning of b being a strict weak order is:

1. Asymmetry: xRy implies that yRx is not the case.
2. Negative Transitivity: xRy implies that for every $z \in C$ either xRz or zRy .

So far it was shown that measurement theory can serve as a formal basis when examining metrics and creating new ones. To show the usefulness of this tool, an impressive fact about finding a single number to express software complexity is shown. Proven already in 1994, this fact is still not well known. Before, it may be useful to read Subsection 3.3.1.2 in advance, especially if the reader is not familiar with the definition of the Cyclomatic Complexity (CC).

3.1.3 The Impossibility of a Single Number for Complexity

To express complexity in software engineering by a single real number was subject of many studies. The author of [Fen94] showed that this is impossible.

Two hypotheses are implicit in much of the work dealing with complexity.

1. Let C be the class of programs. Then the attribute control flow complexity is characterized by an empirical relation system which includes a binary relation b denoted with *less complex than*. More detailed, $(x, y) \in b \Leftrightarrow$ if there is a consensus that x is less complex than y .
2. The proposed measure $M : C \rightarrow R$ is a representation of complexity in which the relation b is mapped to $<$.

The problem is with the second hypothesis. Considering Figure 3.2, three control flow graphs are depicted with CC values two for a), three for b) and two for c). Formally $M(X) = 2, M(Y) = 3, M(Z) = 2$. According to the representation condition, if M is really a measure of complexity, it states that Z is less complex than Y . But there is no consensus about that. Asking programmers they would agree that X is less complex than Y , but not on the fact that Z is less complex than Y . The perceived complexity of software is different for each individual developer. It is a subjective measure and depends on many different aspects. To find consensus about the meaning of software complexity is already a challenge as many different definitions can be found. Therefore, if talking about complexity inside a software development team, it is necessary to clarify what is meant by complexity and to find consensus between the participating persons.

The detailed formal proof of the theorem can be found in [Fen94].

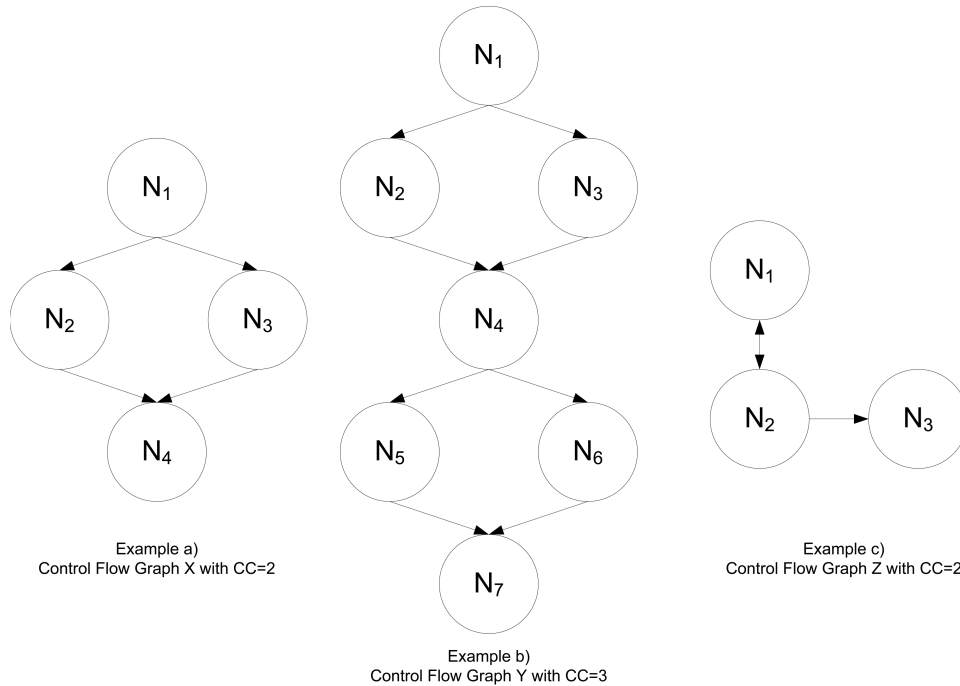


Figure 3.2: Complexity Relation, from [Fen94]

3.2 Weyuker Properties

A set of properties to clarify the strengths and weaknesses of complexity measures was proposed by [Wey88]. P, Q and R denote program bodies. $|P|$ denotes the complexity of P assuming a hypotheticalal measure.

1. $(\exists P)(\exists Q)(|P| \neq |Q|)$
2. Let c be a nonnegative integer, then there are only finitely many programs of complexity c .
3. There are distinct programs P and Q such that $|P| = |Q|$.
4. $(\exists P)(\exists Q)(P \equiv Q \ \& \ |P| \neq |Q|)$
5. $(\forall P)(\forall Q)(|P| \leq |P; Q| \ \text{and} \ |Q| \leq |P; Q|)$.
6. (a) $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \ \& \ |P; R| \neq |Q; R|)$
(b) $(\exists P)(\exists Q)(\exists R)(|P| = |Q| \ \& \ |R; P| \neq |R; Q|)$
7. There are program bodies P and Q such that Q is formed by permuting the order of statements of P , and $|P| \neq |Q|$.
8. If P is a renaming of Q , then $|P| = |Q|$.
9. $(\exists P)(\exists Q)(|P| + |Q| < |P; Q|)$.

The first property states that a measure by which all programs are considered as equally complex is not a useful measure. The second property formalizes the problem of a measure which is too coarse grained. Such a measure would divide all programs into just a few complexity classes. The third property allows different programs with the same complexity, meaning the measure should not be too fine and assign to every program a unique complexity. The fourth property says that two programs computing the same function, can have different implementations with different complexity. The fifth property states that components of a program are no more complex than the program itself. Property six states that assuming a program body R with a fixed complexity in isolation and a program body P with which it is concatenated, R may not interact at all with P , while R might interact with Q in ways which affect the complexity of the resulting program body. The seventh property says that program complexity should be responsive to the order of statements. Property eight states that renaming should have no influence on the complexity. Finally, property nine states that the complexity of a program formed by the concatenation of two program bodies is greater than the sum of their individual complexities (at least in some cases). [Wey88]

Within the representational theory of measurement, Weyuker's axioms are contradictory [Fen94]. This can be shown by using the Weyuker properties five and six. Property five asserts that adding code to a program cannot decrease its complexity. Assuming this, it can be concluded that size is a key factor. It can also be concluded, that comprehensibility is not a key factor, in certain cases a program can be understood more easily as more of it is seen. A size type complexity measure should satisfy property five, a comprehensibility complexity measure cannot satisfy property five. A contradiction appears. Property six asserts that two program bodies with equal complexity, concatenated to a same third program, lead to different complexity. This property refers to comprehensibility and not to size. Properties five and six are relevant for incompatible views of complexity. Therefore, they cannot be satisfied by a single measure.

3.3 Differences between Languages and Paradigms

The programming behaviors available in the object oriented paradigm are different from that of the procedural paradigm. One example is the creation of classes in the object oriented paradigm. Therefore one has to distinguish between these concepts when analyzing code and inferring values for certain metrics. In the following sections, different concepts are studied and metrics to capture these concepts are given.

3.3.1 The Procedural Paradigm

Although the metrics presented in this section can also be applied to the OO paradigm, their origin is in times of the procedural one. Therefore, in this work the paradigms are distinguished. In literature experiments can be found where source code from different program languages are mixed. I don't agree with that, as the origins of the following metrics go back to times where the OO paradigm was not present.

3.3.1.1 Lines of Code (LOC)

A widely used and one of the simplest software metric is the number of lines of code (LOC¹). It is simple and easy to apply. But using this metric, several disadvantages should be kept in mind. First, the goodness of the code is not taken into account, meaning that a short and well designed program may be punished. Secondly, differences between program languages are not considered. Thirdly, there is no agreement on the definition. Should comment lines be counted? What about more than one instruction in the same line? These questions cause dispute.

On the other hand, there are studies that demonstrate the usefulness of the metric. Some demonstrate that LOC is at least as good as any other metric for some software attributes. Some others claim that there exists a direct relationship between LOC and the Cyclomatic Complexity (CC), that will be presented in the next subsection. Considering such studies, one should examine the used data. Many of them are not taken from the industrial field (expert programmers) and to apply them in industry is therefore not advisable. Others mix data from different paradigms which can lead to wrong results. Also the statistical significance has to be examined.

Considering measurement theory, LOC is a valid measure for the length of the program, because the empirical relation *is shorter than* is represented by the relation $<$ between LOC, formally: x is shorter than $y \Leftrightarrow LOC(x) < LOC(y)$.

3.3.1.2 Cyclomatic Number by McCabe

The Cyclomatic Number by McCabe (CC) [McC76] concentrates on the complexity attribute. The idea is to use the number of paths in the control graph of a program as a complexity measure. More specifically, the number of independent paths is considered. Otherwise, even for a simple software, the number of paths is infinite if there is at least one cycle in it. Independent paths are complete paths going from the starting node to the end node of the graph. Considering

¹In this thesis the LOC metric refers to lines of code without comments

the example in Figure 3.3 in a first step four different paths can be determined: $\langle aceg \rangle$, $\langle acfh \rangle$, $\langle bdeg \rangle$ and $\langle bdfh \rangle$. Now the paths are represented as a vector. The vector consists of eight positions (equal to the number of edges). This leads to $\langle 1, 0, 1, 0, 1, 0, 1, 0 \rangle$, $\langle 1, 0, 1, 0, 0, 1, 0, 1 \rangle$, $\langle 0, 1, 0, 1, 1, 0, 1, 0 \rangle$ and $\langle 0, 1, 0, 1, 0, 1, 0, 1 \rangle$. From these four vectors only three are independent. It follows that the number of independent paths is three. A theorem in graph theory says that in a strongly connected graph (a graph where each node can be reached from any other) the number of independent paths is given by $e - n + 1$ where e is the number of edges and n is then number of nodes. This formula can not be applied directly because the control flow graph of a program could be not strongly connected. To make it strongly connected one edge is added from the end to the start node of the program. The number of edges is increased by one, the number of independent paths is the cyclomatic number as defined by McCabe then:

$$v(G) = e - n + 2 \quad (3.1)$$

Applying the formula to Figure 3.3 we get $8 - 7 + 2 = 3$. An interesting property is that if the number of choice points is denoted with d the CC can also be calculated with $v(G) = d + 1$. The definition can also be extended to programs containing procedures. The procedures are represented as separate flow graphs then. The metric was validated experimentally by McCabe and an empirical rule was derived that the CC of a module should not exceed the value 10.

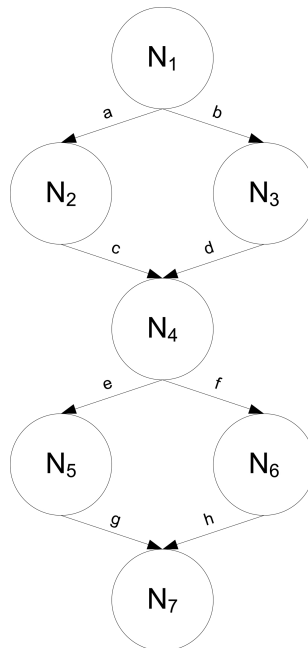


Figure 3.3: Example graph for CC with 3 independent paths

As already mentioned, from the viewpoint of measurement theory the CC is not a valid measure for the complexity of a program. But it is a valid measure for the number of independent paths, which can be used for the property testability for example.

3.3.1.3 Halstead Metrics

While McCabe was focusing only on the complexity attribute, the intention from Halstead was to build a complete theory starting with simple elements and combining them. The fundamental definitions are described here before giving the derived equations [Hal77]:

- Every variable or constant in the program is referred to as *operand*.
- *Operators* are symbols or combinations of them which influence the operands. These can be arithmetic symbols such as $+$, $-$, \times , $/$, keywords such as *if* or *while*, special symbols such as $:=$ or braces and also function names.
- n_1 describes the number of distinct operators.
- n_2 describes the number of distinct operands.
- N_1 describes the total number of the use of operators.
- N_2 describes the total number of the use of operands.

With these basic definitions several equations are derived:

- The *length* of a program is the total number of symbols in the program and defined as $N = N_1 + N_2$. If we compare this measure to LOC, this gives a different possibility to measure the program length.
- The *vocabulary* of a program is defined as the number of distinct operators and distinct operands, $n = n_1 + n_2$.

Now the Halstead Volume (HV) is introduced. The idea is that the number of bits necessary to represent each element is $\log_2(n)$. To represent the whole program we get

$$HV = N \log_2(n). \quad (3.2)$$

For the definition of the programming effort E , the potential volume V^* is introduced. V^* is the minimum possible volume for a given algorithm. The programming effort E is defined as

$$E = \frac{V}{V^*}. \quad (3.3)$$

Obviously, the potential volume is difficult to compute. To determine the programming effort E in practice, an approximation is frequently used [Wey88]:

$$\hat{E} = \frac{n_1 N_2 (N_1 + N_2) \log_2(n_1 + n_2)}{2n_2} \quad (3.4)$$

One major point of criticism is that there is no general agreement among researchers on how to count operators and operands. The counting is language dependent and may be interpreted differently. The reason for this is that the metrics from Halstead were developed in the context of algorithms and not programs. Even so, Halstead metrics are important as they were the first

attempt to define a complete measurement tool for software. Even if lacking a strong theoretical base, they were in use for years as comparison for new metrics.

The presented metrics LOC, CC and HV are used to form the maintainability index (MI), that will be presented later when focusing on the maintenance process. In table 3.1 the differences between the CC and the programming effort from Halstead concerning the Weyuker properties are summarized.

Weyuker Property Number	CC	Halstead's E
1	YES	YES
2	NO	YES
3	YES	YES
4	YES	YES
5	YES	NO
6	NO	YES
7	NO	NO
8	YES	YES
9	NO	YES

Table 3.1: Weyuker Properties: CC vs Halstead's E

3.3.1.4 Nesting Depth (ND)

The Nesting Depth (ND) metric is defined on methods. Informally, the ND of an element is the number of decisions in the control flow that are necessary to reach this element. A rule of thumb is that methods with ND higher than four are hard to understand and maintain. If the ND exceeds eight, it is recommended to split the method into smaller ones. This metric is not obvious to see in the source code. Considering Listing 3.1, a simple source code with ND of three [8]. Here it might be simple to understand. But when it is coming to complex if-statements, the situation becomes different. Considering listing 3.2 there is only one if-statement but at the same time the ND is three as well.

Listing 3.1: Nesting Depth of 3

```
public static void Method(List<string> list) {
    for (int i = 0; i < 10; i++) {
        if (i > 5) {
            foreach (string s in list) {
                Console.WriteLine(s);
            }
        }
    }
}
```

Listing 3.2: Nesting Depth with Complex Statement

```
public static void Method(string s) {
```

```
    if (s != null && s.Length > 0 && s.Contains("a")) {  
        Console.WriteLine(s);  
    }  
}
```

Based on discussions with developers this metric was chosen to be part of the procedural maintainability model presented in Section 3.4.2. The developers agreed that keeping the value of ND small helps to assure analyzability.

3.3.1.5 Interface Complexity

The term interface complexity refers to the external behavior of a function. The straight forward definition is composed by the Number Of Return Points (NOR) and the Number Of Input Parameters (NOI). It is considered as a good practice that a method should have only one exit point. Of course there may be some special cases, in which it may be preferable to allow more than one return point. Even so, the maintenance process is eased with only one. The NOI metric is an indicator for design problems. Functions with a large NOI can be subject of parameter ordering errors, poor design or too much functionality. The interface complexity was assessed to be useful by the developers and is therefore also part of the proposed procedural maintainability model.

3.3.2 The Object Oriented Paradigm

Many OO design methodologies can be found in literature. Reflecting the essential features Booch [Boo94] presents four major steps involved in the OO design process.

1. Identification of Classes and Objects
2. Identify the Semantics of Classes and Objects
3. Identify Relationships between Classes and Objects
4. Implementation of Classes and Objects

In the first step, key abstractions are identified and labeled as possible classes and objects. In the second step the meaning of the previously identified classes and objects is described. Step three deals with interactions as inheritance and visibility. The last step consists of the detailed internal view, the definition of the methods and their behavior. The design step is central in the OO paradigm.

Chidamber and Kemerer [CK94] propose six design metrics on a fundamental theoretical base, the CK-Metrics Suite. Basically it can be distinguished between complexity metrics, coupling metrics and cohesion metrics. Complexity Metrics try to capture the complexity of an entity (which is not a trivial task). Coupling metrics describe how software artifacts are dependent on others. They can be calculated on subsystems, packages or classes. Two artifacts are coupled, if one references the other. Note that dependencies that emerge during execution are not captured with statical analysis. Finally, cohesion metrics describe the coupling inside an element. To get an imagination what is meant by the measures, Figure 3.4 will be used to get an intuition of the terms.

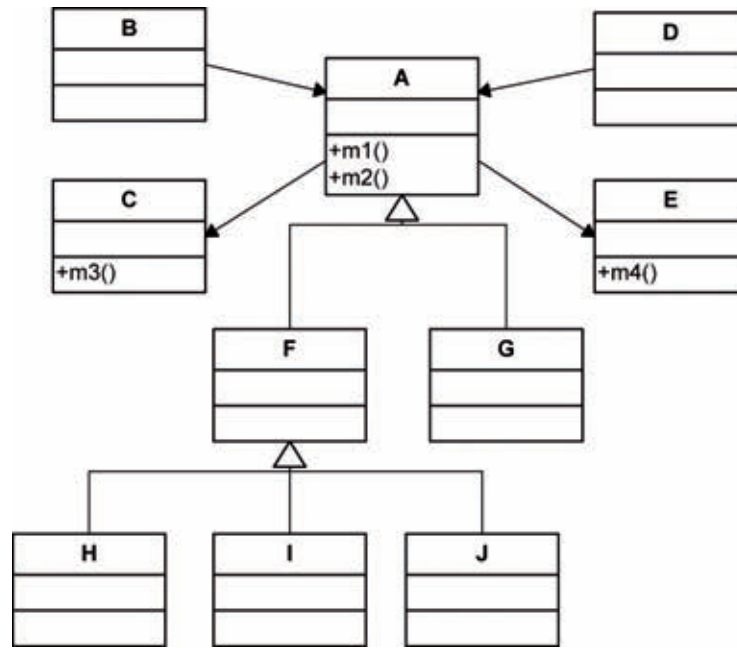


Figure 3.4: UML Diagram, Coupling and Inheritance

1. WMC (Weighted Methods per Class) is defined as

$$WMC = \sum_{i=1}^n c_i \quad (3.5)$$

where c_i is the complexity of the methods. In order to allow the most general application of this metric, complexity is deliberately not defined in a more specific way. Three viewpoints should be considered: First, the number of methods and the complexity involved is a predictor of how much effort it will take to develop and maintain the class. Secondly, a large number of methods indicates a strong impact on children (assuming children will inherit the methods defined in the class). Thirdly, a large number of methods is limiting the possibility of reuse and may be more application specific. A special definition is to set the complexity of one method to one. Then the WMC is equal to the number of methods (NOM). In the class diagram in Figure 3.4 $NOM(A) = 2$.

2. DIT (Depth of Inheritance Tree) describes the maximum length from a class to the root class, where the depth of the root class is zero. Again some viewpoints to consider: First, a deeper class in the hierarchy is likely to inherit more methods. Therefore it may be complex to predict its behavior. Secondly, deep trees indicate high design complexity. Thirdly, the potential of reuse of inherited methods grows for deep classes. In the class diagram in Figure 3.4 $DIT(H) = 2$.
3. NOC (Number Of Children) is defined by the number of immediate subclasses subordinated to a class in the class hierarchy. Viewpoints: First, inheritance is a form of reuse, therefore the greater the NOC, the greater the reuse. Secondly, a large number of children may be an indicator for the misuse of sub classing. Thirdly, the number of children states the potential influence of the class on the overall design. If this influence is high, more effort should be done in testing. In the class diagram in Figure 3.4 $NOC(A) = 2$.

4. CBO (Coupling Between Objects) is the count of the number of other classes to which it is coupled. Viewpoints: First, the more independent a class is, the better is the reuse ability. Secondly, if a class shows high coupling, it may be hard to maintain. Note that this property is especially interesting in this thesis! Thirdly, a high coupling may cause rigorous testing. In the class diagram in Figure 3.4 $CBO(A) = 4$.
5. RFC (Response For a Class) is defined as

$$RFC = |RS| \quad (3.6)$$

where RS is the response set for the class. It is a set of methods that can potentially be executed caused by a message received by an object of that class. Viewpoints: First, high RFC will complicate testing and debugging. Secondly, high RFC indicates high complexity. Thirdly, introducing a worst case value for this metric can assist in the allocation of testing time.

6. LCOM (Lack Of Cohesion in Methods) is given by the following definition: Consider a class C_1 with n methods denoted by M_1, M_2, \dots, M_n . Further let I_j be the set of instance variables used by method M_i . This will result in n such sets $\{I_1, \dots, I_n\}$. Now two sets P and Q are introduced where $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$.

$$LCOM = |P| - |Q| \quad (3.7)$$

for $|P| > |Q|$ or 0 otherwise.

Another simple metric outside the CK Metrics Suite is the Number Of Member Variables (NOMV). This metric is used as indicator for maintainability later on. The developers agree that a high number of member variables influences the characteristics analyzability and testability for example. Member variables in a class are also denoted as *fields*. In listing 3.3, a class with three member variables of type private is given. It is common to access the variables through public getter-methods.

Listing 3.3: Number of Member Variables

```
public class Bicycle {  
  
    private int cadence;  
    private int gear;  
    private int speed;  
  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    public int getCadence() {  
        return cadence;  
    }  
}
```

```
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    // further getter and setter methods ...
}
```

3.4 Metrics for Maintainability

In the field of maintenance, software and hardware differ completely. Degradation or wear out is not a topic here, but other reasons lead to the importance of maintainability:

- Correction of defects, which are discovered during operation or which were previously known
- Hardware change
- Changes of other software components where a module is integrated in
- To increase the functionality

Note that even if software does not wear out, it may degrade due to program errors or memory leaks.

A motivation for studying maintainability in more detail is given by the Johnson Space Center (JSC). There it is estimated that in large systems

1. Software life cycle costs exceeds hardware
2. 80-90% of the total system cost go into software maintenance [NAS04]

A variety of metrics can be collected to rate the quality of a software product from a maintainability point of view. Before applying the OO paradigm these were CC, LOC and the comment frequency denoted with DOC. Inside the OO paradigm the CK-Metric suite offers proper possibilities.

3.4.1 How to Measure Maintainability?

In [LH93] the maintenance effort is measured by the number of lines changed per class. The convenience of this method is not convincing. Imaging a source code, where a programmer has to change a lot to get a desired functionality. Is this good maintainability? Or is it better if the functionality is reached with a low rate of change? In addition the time should be measured in which the amount of lines were changed to judge on the maintainability of a class or a module. This data collection may be time consuming. Furthermore these measures are based on the interaction between the product and its environment. Desirable would be a model where the measurements are based on direct observation of the source code, this is where metrics come into play.

3.4.1.1 The Maintainability Index

An index to evaluate the maintainability of a system, denoted with Maintainability Index (MI), was developed in [CALO94]. Based on the HV, the CC, the average number of LOC per module and optionally the percentage of comments per lines per module. The higher the value of the MI, the better the system is to maintain. Formula 3.8 was developed by collecting data from a large number of systems and expert opinions applying polynomial assessment.

$$171 - 5.2 \ln(HV) - 0.23(CC) - 16.2 \ln(LOC) + 50 \sin\left(\sqrt{2.46COM}\right) \quad (3.8)$$

Several limitations of the MI were discussed in literature. The most important ones are given in [HKV07]:

- The MI is a composite number. Therefore it is very hard (or impossible) to determine the reason for a particular number of it.
- The use of the CC is seen as a flaw. Especially for systems using OO technology. As an example think of Java setters and getters. The CC will be one for all of them.
- The HV is difficult to define and to compute, especially for Java and C#.
- The use of the number of lines of comment is controversial too. More documentation for a module may indicate a complex piece of code which is difficult to maintain.
- The formula is hard to understand, especially the constants and operations like sin. Therefore it is hard to communicate.
- Developers feel a lack of control on the MI, therefore there is a high risk of non-acceptance.

In [PO95] a case study can be found, where several problems of the MI were shown in practice. The conclusion describes the pros and cons in a nutshell:

“One of the nicest things about measuring maintainability with a simple model like the MI is that it gives you a single index of maintainability. This single value is useful in tracking the effects of maintenance changes on different versions of the code over time, including intermodule comparisons of complexity and comparing pre- and post-change software quality. The single index is less volatile than any of the individual metrics from which it is constructed; that is, fluctuations in one metric dimension do not inordinately change the MI, making it more stable. But calculating a single value could also be one of the model’s most serious failings, because by looking at a single value you miss the detailed information provided by the raw metrics which permit you to understand the nature of the change(s) that took place.”

In this quotation one important aspect denoted as *Root-Cause Analysis* was stated. Given one value about maintainability of a software module, does not enable to fix the problem automatically. The possibility to track the reason should be conserved.

Incorporating these limitations other models evolved, many of them build on the ISO 9126 standard. Before highlighting these models, a minimal set of requirements for a maintainability model is given in the next subsection.

3.4.1.2 Minimal Requirements for a Practical Model

After realizing problems with existing maintainability measures like MI, some requirements can be given that should be fulfilled by a model. They are derived from [HKV07] and [LLL08] here:

- Technology independence should be fulfilled in order to apply the model in systems with various kinds of languages and architectures.
- The metrics in use should be well defined and computable with low effort.
- Simplicity and understandability should be kept in focus. These properties help to convince non-technical staff and management. Furthermore the model is easier to communicate.
- Root-Cause Analysis should be possible. It should be possible to find the reason for a high or low maintainability measure. This makes it possible to act on basis of the model. In Figure 3.5 this is shown in detail. System quality characteristics can be caused by properties of the source code and these can be measured by metrics. The metrics in turn indicate source code properties and influence the system quality characteristics.
- The metrics in use need to be tool independent. The more tools available measuring the used metrics, the easier to change a part of the tool chain.

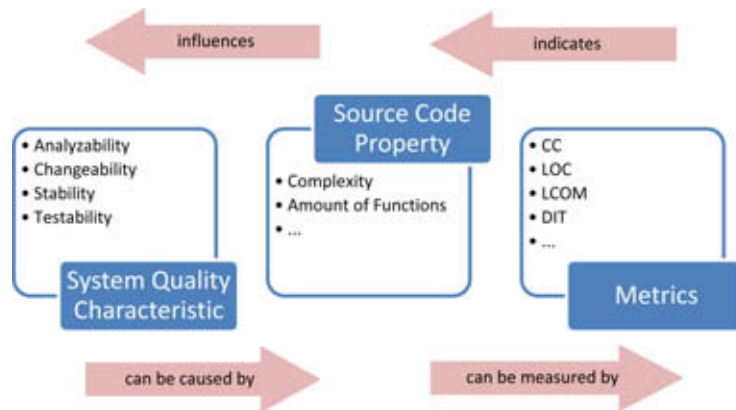


Figure 3.5: Mapping system level characteristics to metrics, assuming that metrics indicate properties and properties in turn influence the quality characteristics

3.4.1.3 Approaches based on ISO 9126

An interesting approach how to measure maintainability as already mentioned in Chapter 2, was proposed in [LLL08]. The approach is based on the ISO 9126 using the criteria Analyzability, Changeability, Stability and Testability, each of them with equal weight. Each criteria is measured by the five chosen metrics CBO, DIT, LCOM (denoted with LCOM-CK), NOC and NOM (Number Of Methods). The first four are taken from the CK-Metric Suite. The NOM metric was not mentioned so far and is defined as the count of all defined methods in a class.

The metrics are weighted different for each criterion. The proposed model is depicted in table 3.6.

Maintainability																			
1					1					1					1				
Analyzability					Changeability					Stability					Testability				
2	2	2	1	2	2	2	2	2	2	2	1	2	1	1	2	2	2	1	2
CBO	DIT	LCOM-CK	NOC	NOM	CBO	DIT	LCOM-CK	NOC	NOM	CBO	DIT	LCOM-CK	NOC	NOM	CBO	DIT	LCOM-CK	NOC	NOM

Figure 3.6: ISO 9126 based Quality Model, from [LLL08]

Based on this model, software classes can be ranked according to its maintainability. The relations and weightings of metrics to criteria was done arbitrarily. This is a weakness, as values derived from practice are needed to apply the model. This is also an important criticism of the ISO 9126 standard which is examined in [AKCK05]:

ISO/IEC 9126 provides no guidance, heuristics, rules of thumb, or any other means to show how to trade off measures, how to weight measures or even how to simply collate them

The authors of [PAT07] propose a data mining approach. In a first step the Analytic Hierarchy Process (AHP) is adopted for the weights assignments. This process reduces complex decisions to a series of one-on-one comparisons utilizing the human ability to compare single properties of alternatives. This is done with a pairwise comparison matrix where the values in the matrix represent the importance of the objectives compared to the others. The computed eigenvalues of the matrix are used afterwards as the weights for the model. In a second step clustering is applied to assist the software evaluator in drawing conclusions. One aspect that could lead to problems in applying the proposed strategy could be the use of 9 metrics. The requirements simplicity, understandability and tool independence are not addressed. Therefore it may be useful to apply the AHP on the model proposed in [LLL08] or on a simpler one.

The process of choosing the metrics for a model and how to weigh them will be examined in Chapter 5 more detailed in the experiments. With applying knowledge from expert-programmers it will be tried to get these values in the field of safety-critical systems. Basically there are three methods to interpret metrics for the model, each one with its advantages and disadvantages:

- Search for reference values in the literature
- Evaluate many projects and take the mean or median as guidelines
- Correlate the subjective opinion of expert developers with metrics to find out which one are the most crucial ones

The problem with the first approach is that there is only sparse information related to special fields in industry. Most of the experiments were performed with Open Source software or on code that was developed by non-experts. The second approach suffers from the fact, that the mean or median of the analyzed projects may not be sufficient. Even so, a valuable reference with a repository from 120 projects and a quality indicator catalog can be found in [SSM06].

As already mentioned in Chapter 2, the QBL are determined using this repository. The third approach suffers from the inherent subjectivity.

Despite of the problems how to find the weights for the model, problems usually occur when assigning metrics to system quality characteristics:

- Which metrics are convenient to capture a certain quality characteristic?
- It may happen that the necessary metrics can be measured only with high effort.
- How to deal with exceptions? These are measures that indicate problems but in fact are not (false positives).
- How to deal with dependencies between the metrics in use?

3.4.2 A Maintainability Model for the Procedural Paradigm

The model proposed in Figure 3.7 is the result of a discussion with expert developers. Keeping the amount of the considered metrics as low and understandable as possible, the suggestion was to use the measures LOC, NOI, NOR and ND. This model is proven with experiments later on. Note that no model can capture every aspect and every model will be subject of ongoing change and improvement.

Maintainability															
W ₁				W ₂				W ₃				W ₄			
Analyzability				Changeability				Stability				Testability			
W ₁₁	W ₁₂	W ₁₃	W ₁₄	W ₂₁	W ₂₂	W ₂₃	W ₂₄	W ₃₁	W ₃₂	W ₃₃	W ₃₄	W ₄₁	W ₄₂	W ₄₃	W ₄₄
LOC	NOI	NOR	ND	LOC	NOI	NOR	ND	LOC	NOI	NOR	ND	LOC	NOI	NOR	ND

Figure 3.7: Proposed Procedural Model for Maintainability, $W_{i,j}$ describing the weights for the different criteria

3.4.3 A Maintainability Model for the OO Paradigm

As can be seen in Figure 3.8, the model proposed consists of four metrics again in the bottom level. Each of this metric is weighted individually for each quality characteristic, these weights are determined in an experiment described in Chapter 5. The metrics in use cover the characteristics coupling (CBO), inheritance (DIT and NOC) and size (NOMV). The cohesion aspect is not covered, because the amount of the metrics should be kept as low as possible to guarantee understandability and simplicity. Furthermore, using more metrics complicates the process to assign the weights. It is clear that the model is not complete with the chosen measures, but to be applicable in practice, a trade off between the used metrics and the completeness has to be accepted. Also experiments from [BD04] give advices that the LCOM metric is not a good predictor for testability, an important part of the maintainability model. From the authors point of view the four metrics used in the OO model capture as much aspects as possible keeping the effort to apply the model at a moderate level.

Maintainability															
W ₁				W ₂				W ₃				W ₄			
Analyzability				Changeability				Stability				Testability			
W ₁₁	W ₁₂	W ₁₃	W ₁₄	W ₂₁	W ₂₂	W ₂₃	W ₂₄	W ₃₁	W ₃₂	W ₃₃	W ₃₄	W ₄₁	W ₄₂	W ₄₃	W ₄₄
CBO	DIT	NOC	NOMV	CBO	DIT	NOC	NOMV	CBO	DIT	NOC	NOMV	CBO	DIT	NOC	NOMV

Figure 3.8: Proposed OO Model for Maintainability, $W_{i,j}$ describing the weights for the different criteria

Considering Figure 3.9, the preceding step was the construction of the model, based on theories in software measurement and discussions with expert developers. To generate a model for maintainability, further steps are necessary. The parameters (in this case the weights) will be determined and the model will be validated. As can be seen the arrows are both way, meaning that the model can change. Based on experiences one may have to go back from the parameter stage to the construction step. Also results from the validation step may influence the parameters. It is important to realize that the suggested model can assist the developers to produce maintainable software. It is not the only perfect solution. The maintainability model itself will be subject of permanent maintenance.

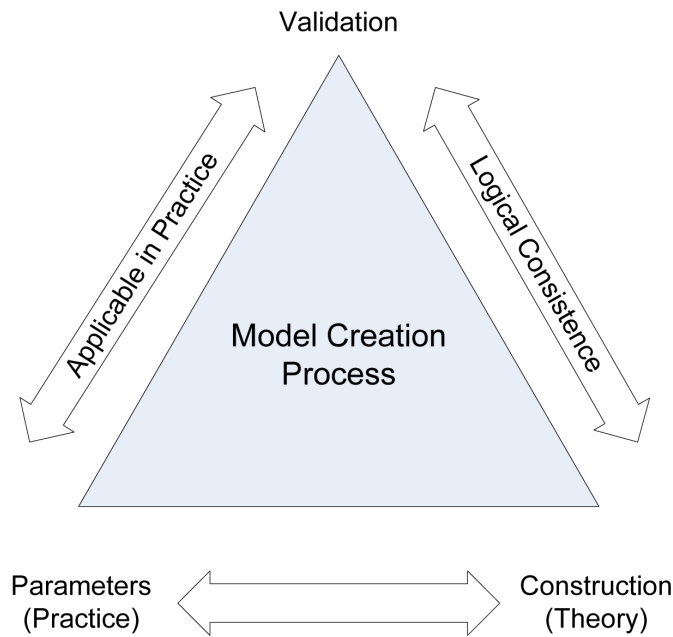


Figure 3.9: Model Construction Process

3.5 Comparison of the Approaches

Recalling the MI, a major point of criticism was that is not possible to track the reason for the values calculated by the measure. To say it with one word, root-cause analysis is not possible in a simple way. The approaches based on the ISO 9126 standard allow to perform root-cause analysis and at the same time the advantage of the MI is preserved. On the upper level of the ISO 9126

approaches, there is still a single value to depict the development of maintainability over time. The difference is that the developer can go one level down and detect what should be changed, allowing concrete actions. Also concerning understandability the ISO 9126 approaches are clearly the better choice. Even for non-programmers and non-experts the models are understandable and easy to communicate. Beside these advantages, there are also challenging tasks. The questions which metrics to use and how to weight them are the key issues. But these issues have to be solved for any other metric-based approach as well.

4 Integrating and Selecting a Software Metrics Tool

To introduce a metrics program can be a challenge for an organization. The effort to collect data for calculating useful metrics is often a barrier. It takes time, cost and resources. From the developers point of view a commonly cited reason against the use of a metrics program is the misuse of the data against themselves. Therefore, managers and team leaders have to take several considerations into account to successfully adopt a metrics program. Even if the team is convinced by the importance of applying metrics, it is a non-trivial task to choose a convenient tool.

4.1 Perspectives

In Figure 4.1, the different roles and views that occur in the software development life cycle can be seen. A developer will be interested in the code and design level. The focus of the project coordinator is more wide and for the manager the high system level view will be of interest. To satisfy all these rules and views is not the focus in this thesis. Even so, in section 4.5.3 the idea how to address these issues is outlined. In this work the focus is on the code and design level.

4.2 Defense Position of Developers

To benefit from the introduction of metrics the developers have to be convinced of the usefulness. For overcoming the defense positions of the team members, several strategies were pointed out in [SS05]. In a first step the defense positions that usually occur should be considered more detailed:

- Optimism
- Delegation
- Automation
- Specialty

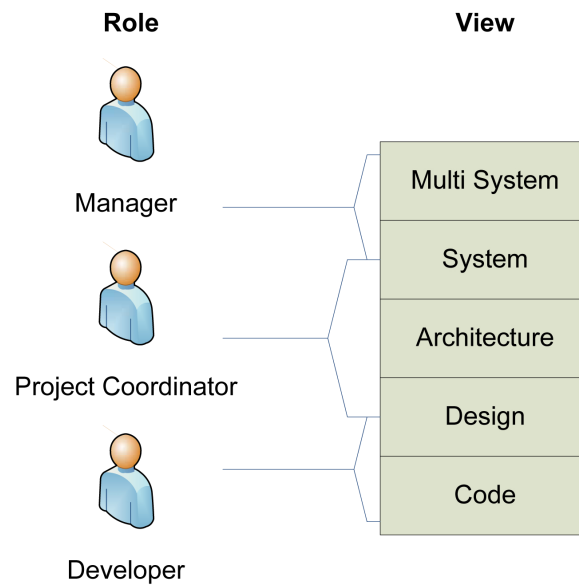


Figure 4.1: Roles and Views

- Already Established

Optimism occurs in situations in which the maintainability of a software system is not a problem. The team is still complete and the software is a new developed one. Therefore the effort to determine metrics is doubted. Developers claim that they produce a basically high quality (we are experts) and use a high quality process (we are using agile approaches) and the newest technologies.

Delegation refers to the sub systems which were developed out of house. It may occur that developers claim that any quality assurance actions should address these systems before the own ones. In some cases this may lead to the situation that nobody of the team is responsible for the produced code quality and as a consequence quality management is thought to be useless.

In modern systems typically code generators are in use. There are different types of code generators describing how much of the code is produced automatically. Only in a few cases it will be possible to develop the overall source code automatically. Therefore also in these systems quality assurance actions make sense.

Specialty can occur in projects that make exhaustive use of standard technologies, tools and special solutions. Quality assurance actions are rejected with the argument that these actions do not work in this specific context. An example therefore might be that the analysis tools can not handle the code because of special programming language constructs.

It can also occur that project members share the opinion the suggested Code Quality Management (CQM) techniques are already established in own developed tools. This is usually not the case, as the core business of the organizations is not in this area.

To avoid the previously listed problems and make profit of the measurement process, a cooperative working atmosphere is of interest. The developers should be involved in the development of the models to avoid a defense position. Furthermore, applying a maintainability model (or a general quality model) offers the possibility to objectively proof that the produced source code is of high

internal quality. If problems are really in out of house produced code, then the model helps to show this.

4.3 Aspects of Analysis Tools

Searching for a convenient analysis tool is a challenge. Especially because of the wealth of existing systems, each of them coming with pros and cons. In this section, the general architecture and aspects to choose and compare tools are examined.

4.3.1 General Architecture

As can be seen in Figure 4.2, the input for the static software analysis system considered in this work is the source code. Different tools force different restrictions on the input, for example only compilable code is accepted or included references are taken into account and have to be available. The tools also vary in the amount of measured properties. In practice a trade off between accuracy and cost has to be found. On the one hand, enough metrics to build an adequate model for a certain quality characteristic have to be available. On the other hand, if too much metrics or functionalities are considered, the tool license cost may increase.

It may be useful to introduce additional external tools. This can be the case if there is no single tool that can capture all necessary metrics. Another reason might be that with combining several free tools license costs can be saved. If applying this practice it should be kept in mind that the integration process will be more complicated. One intention of the implemented prototype system is to offer the possibility to combine multiple tools.

Also an important characteristic is the presence of a Data Base System (DBS). For custom models, the database will have to be designed and filled manually. A DBS allows to formulate queries that can be used to specify quality models. It can also serve as an interface between the data extraction process and the analysis process (in Figure 4.2 denoted with Part 1 and Part 2). Furthermore, a DBS can deal with a huge amount of data and offers a persistent way of storage. Also the combination of several analysis tool is easier, as the DBS can be used as a central repository for further data processing.

Various differences also occur in the analysis component. Some tools offer no graphical interface. This can be an advantage when the integration into an automatic build environment is intended. Then the specific quality model has to be defined manually and extracted from the DBS to process the data for the reports.

It will be almost impossible to find one tool fulfilling all desired requirements. Manual processing will be necessary to benefit from applying the expertise of metrics, especially when transferring the data measured by the individual tools to the common database. The next part of the thesis defines criteria which should be fulfilled by an individual tool to allow easy integration.

4.3.2 Three Key Criteria for Success

Auer et al. [AGB03] name three key criteria concerning integration capabilities:

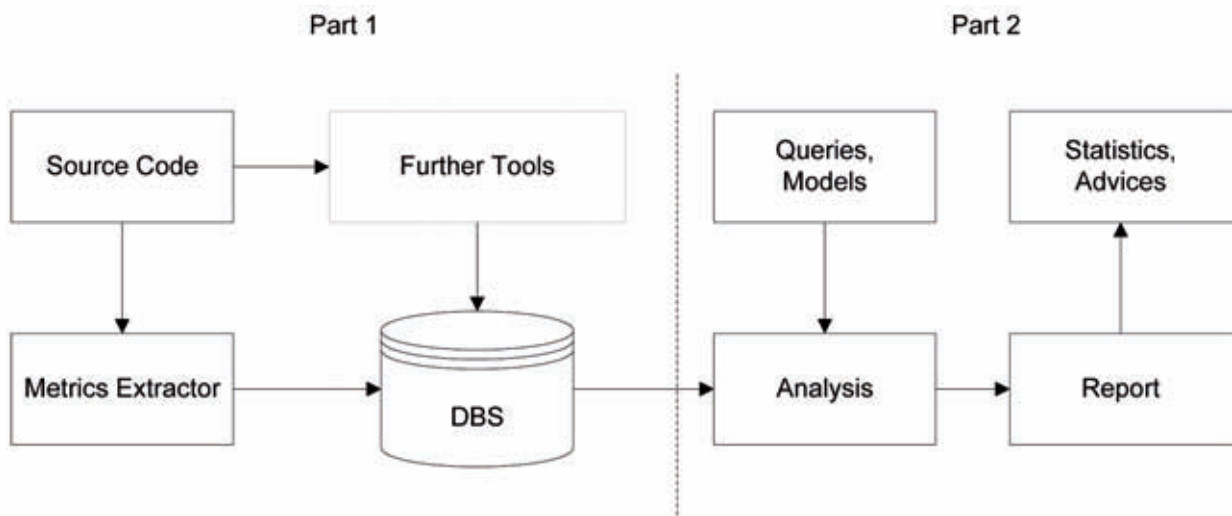


Figure 4.2: Typical Model of a Static Software Analysis System

- Platform: This criterion refers to the operating system but also to the database. Some tools are only available on one operating system. This can lead to problems if server components are in use which are mostly Unix based. Other tools only work with specific database engines. This can lead to problems if there is a particular database server already established in the company.
- Input/Output: Various tools are restricted to a few input file formats. Also the output for further processing can be limited to proprietary file formats.
- Automation: Another key aspect of metric data processing is the data collection process. It should be kept as simple as possible and be done in an automated way to avoid additional effort for developers. Manual data input usually leads to errors and is seen as a flaw. To counteract, the static analysis should be integrated into the *Nightly Build* procedure.

The evaluation of different tools in [AGB03] shows that the seamless integration is far off. The support for automation is rare. Missing interfaces and platform dependencies further complicate the situation. Moreover, tool evaluation is cost intensive due to the variety of products on the market. Considering all these aspects it becomes clear that the objective of a tool integration process has to be restricted to a certain area. The aspects relevant in this thesis are:

1. Operating system support for Windows and Unix systems.
2. The programming languages addressed are C, C++, C# and Java.
3. The objective is to implement a model that supports the maintenance process, therefore convenient metrics for this purpose have to be captured.
4. A structured output that can be used for further processing.

4.3.3 Different Conclusions with Different Tools

Another interesting point of consideration should be if using different tools leads to different conclusions assuming the same metric and the same input. In [LLL08] this question was subject of research. The answer is yes, meaning that the definition of the measured metric has to be studied carefully. Following problems can arise:

- Vendor specific definition of a metric.
- Wrong implementation of the measurement process of a well defined metric.
- Metrics denoted with the same name as well defined metrics but meaning something else.
- Poor documentation leading to the impossibility of comparison.

Note that before choosing a tool, it is of importance to know the objectives of the measurement activities. Defining the objectives makes it easier to know which tool should be considered in detail. A list of questions that can help might be the following one:

- Is the measurement activity for assessment or prediction?
- Which entities are the subject of interest?
- Which characteristics and attributes should be measured?
- What software metrics should be used?

The answers to the questions in the context of this thesis were already given. The maintainability measure is intended to be used for prediction. Software modules on the source code level will be considered. The question of how to weight the metrics is another challenge as will be shown in the next chapters. Note that most software metrics extractors don't offer the possibility to define models or to weigh them. This is a major difference to the implemented prototype system, where this feature is offered to the user.

4.4 Selected Tools

Keeping the aspects from the former section in mind, the next step is the selection of the tools. In [LLL08] a comparison of several tools is given that can assist in this question. The tools used in this work are considered more detailed in the next subsections.

4.4.1 Resource Standard Metrics (RSM)

Considering RSM [7], the key features are the support of Windows and Unix systems and also the support of multiple programming languages. Furthermore, xml-outputs can be generated which allows parsing for further processing. These capabilities cover three of four criteria required and described above. The only drawback of the tool is the restriction to the following metrics in the OO paradigm (considering class level):

- Number of Data Attributes (public, private, protected)
- Number of Methods (NOM) (public, private, protected)
- Number of Children (NOC)
- Lines of Code (LOC)
- Number of Input Parameters
- Number of Return Points
- Cyclomatic Complexity

Not all metrics that would be interesting for the maintainability model are captured, especially coupling and cohesion metrics are missing. There are two possibilities to implement the model. Either use only the metrics captured by RSM or integrate further tools to capture the missing ones and collect the data in a database. As cutting the cohesion and coupling metrics would lead to an inconsistent view on the maintainability model, it is inevitable to search for further tools which capture the missing metrics. Note that it is a common pitfall to develop models based on the metrics available from one special tool.

4.4.2 OOMeter

With OOMeter [ARK05] it is possible to analyze Java and C# source code. Considering the supported programming languages, C++ is not covered. Even so, there are further reasons for using this tool:

- Multi platform support, meaning that OOMeter can be used on all platforms that support the Java 1.2 standard or higher runtime environment.
- Generation of xml outputs which allows further processing.
- Processing of UML models (may be useful for future work).

4.5 The Implemented System

After selecting the metrics extraction tools, a system is needed which further proceeds the produced data. Combining multiple available tools is an important requirement as the developed models may need data that can't be delivered from only one tool. The given implementation shows that this is possible. User defined quality models can be created and processed. Note that this offers a wide range of functionality that is not limited on the maintainability characteristic.

4.5.1 Architecture

As already highlighted in Figure 4.2, a DBS can serve as a common interface. The prototype system makes use of this fact in the way, that combining multiple tools is done by generating xml outputs of them and afterwards storing the results in the DBS applying a parser. The consequence is that individual parsers have to be implemented if a new tool is taken into the system. But this cost is very low, considering that all the other functionality (creating user specific models, ...) is based on the DBS, not on a special tool. Furthermore with this approach it is possible, to decrease license costs. It may occur that combining multiple open source tools achieve the same result as existing solutions with high license cost.

The requirements to the prototype system are

- Creating user defined models based on characteristics and metrics.
- Define the weights for the metrics and characteristics in use.
- Assess source code based on the user specific models.
- Combination of multiple metrics extraction tools should be possible.

The prototype system is developed with web access in mind. The users can use the system via a standard web browser. In section 7 it will be shown which advantages come with the distributed approach.

Developing a web based application from scratch is time consuming. Many configuration files have to be created and the CRUD (Create, Read, Update, Delete) functionalities have to be implemented to access the database. To get the different technologies working together is a challenge. Usually this work is similar in many web based applications. This is where Seam [All09] comes into play.

Seam is a powerful open source development platform for building rich Internet applications in Java. Seam integrates technologies such as Asynchronous JavaScript and XML (AJAX), JavaServer Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0),... into a unified full-stack solution, complete with sophisticated tooling.

Before Seam, developers felt that they spent more time with solving technology problems than implementing the real business logic. Seam overcomes this gap. The principle is depicted in Figure 4.3. The technologies needed for building an application are in a Seam container, this principle offers the possibilities to refer to objects where they are needed, at the same time guaranteeing the autonomy of the layers. The interaction of the User Interface (in this prototype JSF) and the Business Logic (in this prototype EJB) is enhanced. The necessity of creating additional connection layers is omitted. At the same time many different technologies are supported. The motivation for this approach is the DRY (don't repeat yourself) principle, which means redundancy should be avoided when developing applications.

The built application is not bound to a specific server. As can be seen in 4.3, the communication is done with the JCA (J2EE Connector architecture) and JTA (Java Transaction API). The server in use for the prototype is JBoss. Together with the tooling support within Eclipse (JBoss Tools plugin), this combination offers a powerful framework for the development. All the former technologies may be overwhelming, but as can be seen later on, it is worth using them. Anyway, the time consuming effort to get known to the technologies can be a barrier.

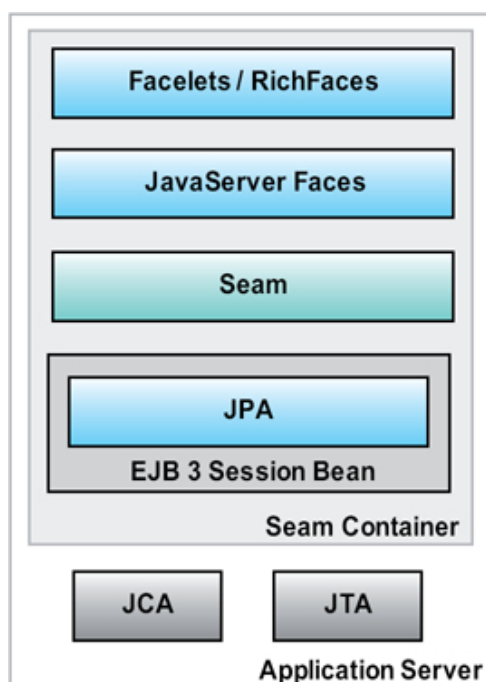


Figure 4.3: A Cross Section of the Technologies Incorporated in the Seam Stack, from [All09]

4.5.1.1 The Java Persistence API (JPA)

How to persist data is a central question to almost every enterprise application. Entities have to be translated between the Java runtime environment and a relational database. Java persistence is the mechanism for this task. For many developers this is the most popular technology of the JAVA Enterprise Edition (EE) platform.

Coming from the perspective of the database, Java persistence performs read and write operations like any other database client. On the other hand, from the perspective of the programmer, it is much more than only this. One of the reasons why it was created, was to extract the SQL out of the code by replacing it with object manipulation. Instead an object representation of the database is offered to the programmer. This is clearly an advantage as almost all modern applications are dealing with object oriented paradigms. To create this abstraction from SQL and offering the object representation, four concepts are introduced:

- Entities (Classes)
- Persistence Unit
- Persistence Manager
- Transactions

The relationship between these concepts and how they work together can be seen in Figure 4.4. The task of the *Persistence Unit* is to organize the metadata. By metadata entities are mapped to the database. The *Persistence Manager Factory* uses the managed metadata from the Persistence Unit and with this information *Persistence Managers* can be created. The Persistence Managers

are implementing the moving process between the Java runtime and the database. This process is denoted by *Entity Life Cycle*. It is recommended that operations performed by the persistence managers are within the scope of one *Transaction*.

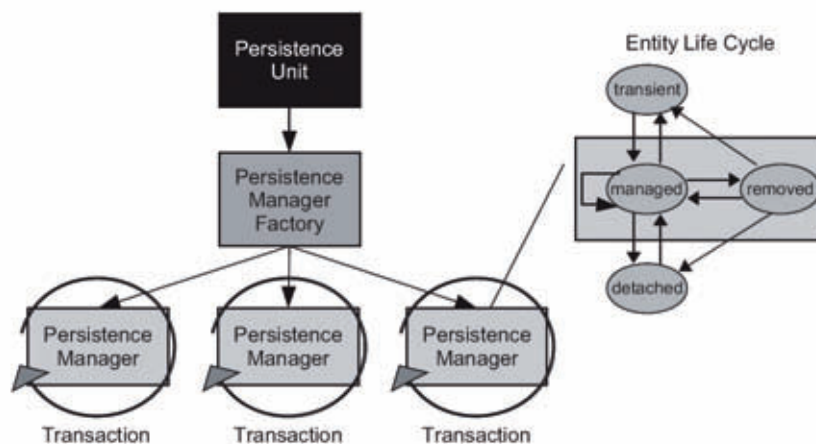


Figure 4.4: Java Persistence: Concepts and Life Cycle, from [All09]

Entities in an Object Relational Mapping (ORM) are the point of contact between the application and the database. The transportation of the data is done by these central elements. One could have the impression that they are just a space for holding data. But especially within SEAM they are much more. As an example SEAM allows to bind entity classes directly to the user interface level to get form data. What is not implicit included in the entities is the database schema. For this purpose metadata for the mapping is included. On the one hand this can be done by ordinary xml-files, but on the other hand there is a much more elegant possibility by using annotations. What does this mean? Consider the source code in Listing 4.1. It is a part of the Metric class used in the implemented prototype system. Especially consider the annotations beginning with an @. With these annotations we say that Metric should be mapped to a database table METRIC. Furthermore, the id is defined with the @ID and with the @GeneratedValue it is expressed that it should be auto-generated. The id attribute should be mapped to the table METRIC_ID and furthermore unique, not nullable and have a precision of 18. All this is specified with the @Column and the options of it. This metadata can now be used to create the database automatically. On the other hand it is possible to create entities out of an existing database schema, called reverse engineering. This is all offered by Java Persistence. But SEAM goes even one step further. The entities can be used directly to create a GUI that is mapped already to the entities, bringing the capabilities of Java persistence to its full potential.

Another important aspect that can be implemented with annotations is the relationship between the entities. Looking at the @ManyToOne there may arise questions. What is specified with this annotation? The relation is depicted in Figure 4.5. One Metric-Object can have exactly one BasicMetric-Object. On the other hand, one BasicMetric-Object can belong to several (0 to N) Metric-Objects.

With the option `fetch = FetchType.LAZY` it can be expressed that related entities should only be loaded on demand. This is called *lazy loading*. Placing the annotation before the `getBasicmetric()` method tells Java persistence that the relationship refers to the BasicMetric-Object.

To complete a bidirectional Many-To-One Relationship the BasicMetric-Object also has to know about its Metric-Object. The corresponding annotations are shown in listing 4.2. Placing the

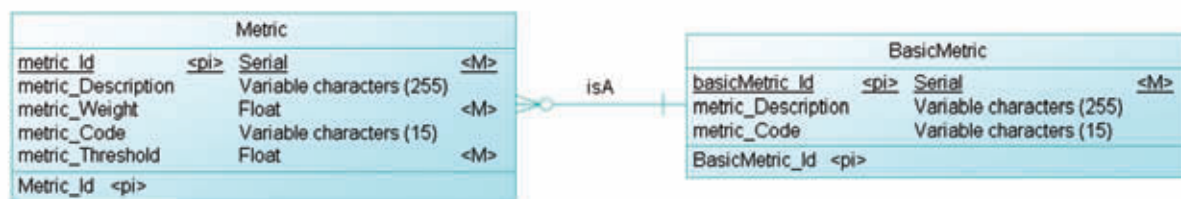


Figure 4.5: Many To One Relation, Example

annotation before the `getMetrics()` method, Java persistence assumes automatically that the relationship refers to `Metric-Object`. The key element in `BasicMetric.java` is the **mapped-by** option. It tells Java persistence that there exists a method `getBasicMetric()` in `Metric.java` and the relationship is completely specified with these attributes. Note that this was only a small example. Any kind of relation can be specified with annotations.

Listing 4.1: Annotations in `Metric.java`

```

@Entity
@Table(name = "METRIC")
public class Metric implements java.io.Serializable {

    // some variables ...

    public Metric() {
    }

    @Id
    @GeneratedValue
    @Column(name = "METRIC_ID", unique = true,
            nullable = false, precision = 18, scale = 0)
    public long getMetricId() {
        return this.metricId;
    }
    ...
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "BASICMETRIC_ID", nullable = false)
    @NotNull
    public Basicmetric getBasicmetric() {
        return this.basicmetric;
    }
    ...
}
  
```

Listing 4.2: Annotations in `BasicMetric.java`

```

@Entity
@Table(name = "BASICMETRIC")
public class Basicmetric implements java.io.Serializable {
  
```

```
//some variables...

public Basicmetric () {
}

@OneToMany(cascade = CascadeType.ALL,
           fetch = FetchType.LAZY, mappedBy = "basicmetric")
public Set<Metric> getMetrics () {
    return this.metrics;
}
...
}
```

4.5.1.2 Enterprise JavaBeans Technology

Enterprise JavaBeans (EJB) are standardized components within the Java-EE servers. The purpose is to simplify the development of multi-tier, distributed systems. In business applications concepts like security, naming and transactions are of importance. When working with SEAM, it is advisable to understand the basics of the EJB technology, as the business logic is usually executed in an EJB-Container. They can be accessed either *remote* or *local*.

There are different versions of EJB's:

- Entity Bean
- Session Bean
- Message Driven Bean

Entity beans are a model of the persistent data of the system. Session beans are modeling the processes performed by the user. Usually they make use of the entity beans. Two types of session beans can be distinguished, *stateless* and *stateful* ones. A stateful session bean owns a memory. It is aware of his history. This means that it can save information from one method call and offer this information to the next method calls. Contrary, in a stateless session bean all parameters have to be defined that are necessary for the successful execution of a method call. As a consequence stateless session beans have no identity, whereas stateful beans do. The third type are the message driven beans used for asynchronous communication.

4.5.1.3 Implementation of the Parser Functionality

The central issue in this subsection is to offer a tool to fill the database in our prototype system. As many existing metrics extractors create xml-reports, it is obvious to use existing parsers for this task. Inside the parser it is possible to persist the previously created measurement results. It would be possible to integrate the parsers directly into the web-based architecture. But several disadvantages are coming with this approach:

- The integration into a nightly build procedure is complicated.
- The reports generated by the existing metrics extractors have to be loaded manually by a user. An action would be needed to parse the report then (at least a click on a button).

Considering the former aspects, it was decided to implement the parser functionality as a separate module that can be integrated into the nightly build procedure with a single line of code. The invocation of the parser module therefore should allow to specify options for the type of the report (indicating which metrics extraction tool was used to create the report) and for the path of the input file (indicating where the report is stored). A typical invocation is given in Listing 4.3. The option `-f` gives the possibility to specify the input file (`report.xml` in this case) and the option `-t` specifies the tool (Resource Standard Metrics).

Listing 4.3: Invocation of the Metric Parser

```
java -jar MetricParser -f ./report.xml -t RSM
```

Different types of parsers can be distinguished. In an object parser, the whole xml-tree is built and stored in memory. As a consequence, every element can be accessed any time. The disadvantage coming with this approach is that for large xml-files (which usually applies to static software analysis) the demand for memory can exceed the available resources. Also the parsing can become very slow in this case. The Document Object Model (DOM) is an example for this paradigm.

Beside DOM based parsers there are push and pull parsers as well. SAX (Simple API for XML) is the standard for push parsers, where the parser itself is controlling the process. Note that the SAX standard is no W3C recommendation. These parsers are event driven. The whole document is traversed only once sequentially. The parser fires events (if syntactic units are identified). For each event a call back function can be registered from the programmer to react on the event. These parsers are the better choice for large files as the memory requirements are constant! Furthermore, this paradigm is faster. On the other hand, if there is more than one access on a xml-node, the application programmer is responsible for buffering.

Although SAX was originally developed for Java, DOM and SAX are platform independent. For DOM, the implementation even does not need to be in an OO programming language. The choice of technology depends on the available resources and the expected amount of data to be parsed.

4.5.2 Modeling the DBS

The first step is to create a conceptual model. In a conceptual model the setup is not bound to a specific database technology. From the conceptual model a physical model can be created automatically. In the physical model the database technology is specified. In the prototype system *Microsoft SQL Server 2005* is used. Note that changing the underlying database technology is supported with low effort using conceptual models. Once the database is set up, the most impressive tool of Seam can be applied. Seam allows to create an entire project, complete with a build script, environment profiles, a compatible set of libraries and configurations that is ready to be deployed. Furthermore, once the DBS is implemented, reverse engineering can be used to create an application prototype including the CRUD operations out of the DBS. This fact makes it worth to invest the necessary time to get known to Seam.

An excerpt of the conceptual database model is given in Figures 4.8 and 4.9. In the left part it can be seen that a model is formed by characteristics. A characteristic is formed by metrics. In the right part a module is split into files, a file into classes and a class can contain several functions. A measure can be created for a class or a function and measures a certain defined metric. This approach makes it possible to apply different quality models on the same functions or classes. The names for *class*, *file* and *function* are substituted by the German equivalents *Klasse*, *Datei* and *Funktion* as the English names are occupied internally by the used programmes.

Mandatory fields are marked with $\langle M \rangle$ and the primary identifiers with $\langle pi \rangle$. Note that the modeling step of the database has to be performed carefully and correct as it is the input to the reverse engineering capabilities of seam-gen.

In Figure 4.6 a sample screen of the implemented system is depicted. This screen offers the possibility to define metrics, define thresholds that should not be exceeded and relate metrics to a specific characteristic. Note that the complete screen was generated with the seam-gen functionality. Seam-gen tries to create the functionality out of the generated database scheme with reverse engineering. Even error messages appearing in real time are already included and mandatory fields are marked with a star. Implementing these functionalities manually would be time consuming, therefore SEAM offers a powerful tool after the effort is done to become familiar with the SEAM technology.

After the project was created with seam-gen, the application has to be enhanced. So far only the CRUD functionalities were created. To develop in a comfortable way, it is desirable to import the existing project into an Integrated Development Environment (IDE). Also the support for this step is given. The created project already includes the files necessary for Eclipse. This makes it possible to import the project and continue developing with Eclipse and the JBoss Tools plugin.

Edit Metric

metricCode:

metricDescription:

metricThreshold*:

metricWeight*: ✘ 0.2a* muss eine Zahl zwischen 4.9E-324 und 1.7976931348623157E308 sein. Beispiel: 1999999

* required fields

Save Delete Done

basicmetric* characteristic*

characteristicId	characteristicCode	characteristicDescription	characteristicWeight
1	ANA	Analyzability	0.27

Select characteristic

Figure 4.6: Sample Screen of the Implemented System

4.5.3 Integration of Existing Tools in a Portal

The system so far was already created with *portal* character. A portal in the context of quality analysis offers the possibility for a centralized data storage, data analysis and presentation. It is

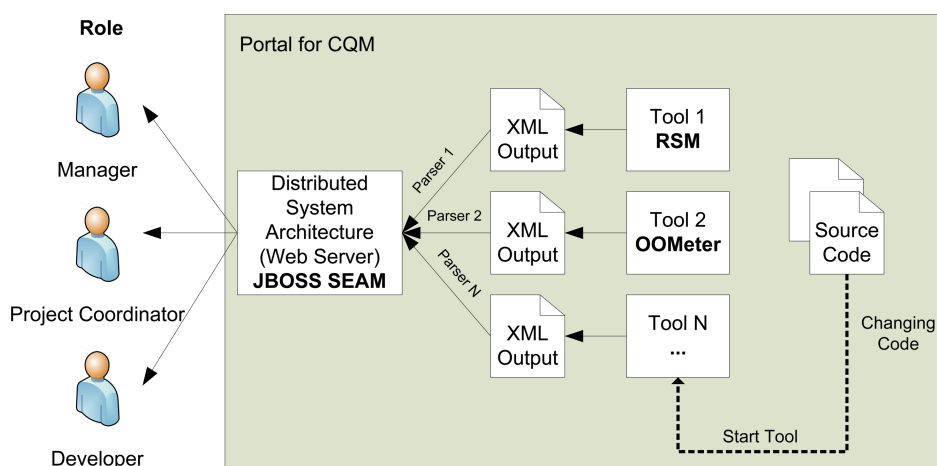


Figure 4.7: Portal Architecture, adapted from [SSM06]

common that a standard web browser is used to access the portal. Furthermore, if security mechanisms are installed, view from any where can be granted. With the login possibilities personalized areas can be offered and roles (see Figure 4.1) can be assigned to certain persons. Especially it could be possible, that the project manager defines a certain quality model for a specific project and the developers are assessing their source code following this model. Also the code quality can be monitored over a certain period of time referring to a specific model. Note that the analysis can be integrated in the nightly build procedure and the produced output can be integrated into the DBS without high effort, once the system installed. From the business perspective portals are attractive because the collected data can be used multiple times minimizing resources.

In this thesis, an exemplary maintainability model was created. Note that the implemented system can be used to define any other model. The necessary steps will be the determination of the metrics that need to be measured, the definition how to weigh them and the integration of the tools to measure these metrics. The portal architecture makes this approach possible.

In Figure 4.7, the overall possibilities should become clear. If source code changes, the analysis tools are started, creating reports. These reports are processed by parsers and stored. The different persons can access the portal depending on their role in the system. The used technologies for the implementation of this work are highlighted.

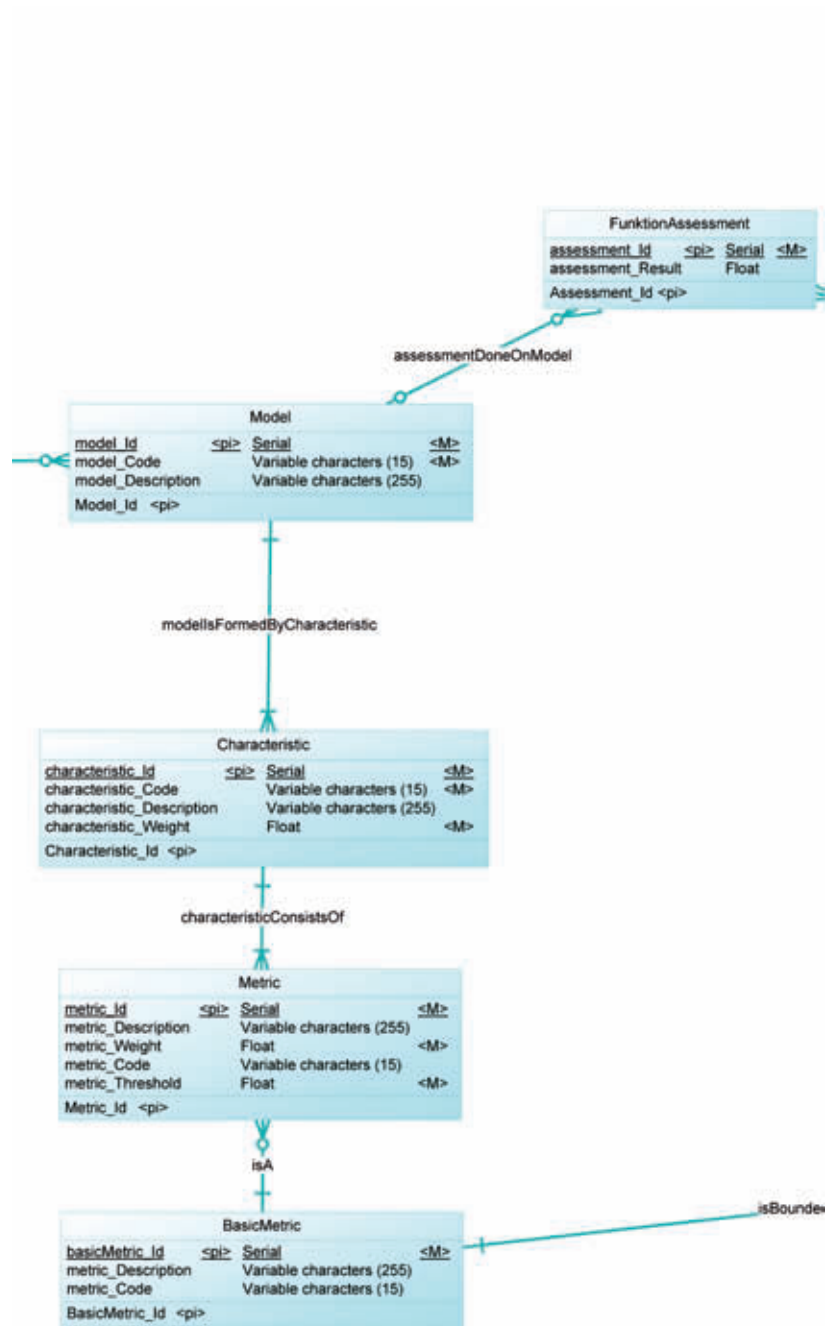


Figure 4.8: Database Conceptual Model, Part 1

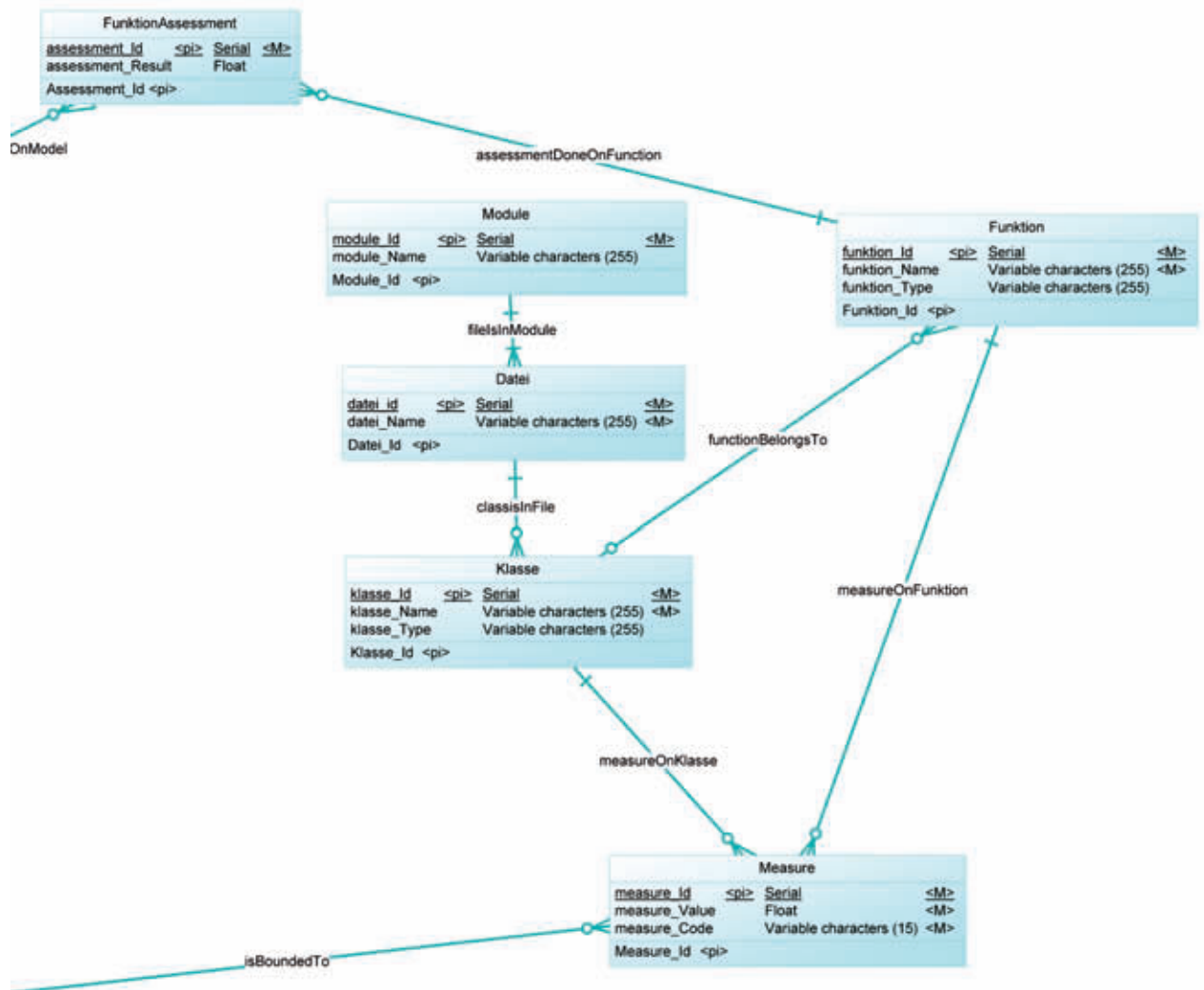


Figure 4.9: Database Conceptual Model, Part 2

5 Experiments

To prove if software metrics can assist or not in the software development life cycle, experiments and data from the industrial field are needed. In this chapter the experiments are described. The results are given in Chapter 6 separately.

5.1 Validating Software Measures

Before starting with experiments, it is important to examine the process how to validate a software measure. According to [Fen94], validation has to adhere to the following steps:

Validating a software measure in the assessment sense is equivalent to demonstrating empirically that the representation condition is satisfied for the attribute being measured. For a measure in the predictive sense, all the components of the prediction system must be clearly specified and a proper hypothesis proposed, before experimental design for validation can begin.

Many studies related to software metrics can be found where the expertise from measure theory is ignored. In most cases a measure is proposed and validated by showing that it correlates with some other existing one. There is neither a specification of the required prediction system nor a specification of the experimental hypothesis.

5.2 Weights Assignment to the Proposed Models

To find the weights for the proposed models presented above a workshop was performed. Seven developers were asked to perform the Analytical Hierarchy Process (AHP). It is a structured technique for solving problems consisting of complex decisions ([3]). The process is examined more detailed in the next section. The technique helps the decision makers (in this case the developers) to find the solution that best suits their needs and their understanding of the problem. This is exactly what is wanted in a software development team. In a first step the attendants were asked to apply the AHP on the higher level, considering the sub characteristics analyzability, changeability, stability and testability. The terms were defined based on ISO 9126 to guarantee

a common notion of them. In a second step, every sub characteristic was considered in detail separately once for the procedural and once for the OO paradigm. The AHP process was applied on the four chosen metrics then. The developers came from the field of safety critical systems and were employees of Frequentis [4]. The templates used for the AHP processed filled by the developers are depicted in the appendix. The results are presented and interpreted in Chapter 6.

5.3 The Analytical Hierarchy Process in Detail

The technique developed by Thomas L. Saaty is a structured technique for dealing with complex decisions. The basis are the fields of mathematics and psychology. The process makes use of a hierarchy to define less complicated subproblems. Each one of the subproblems is analyzed independently then. Typical fields where the process is applied are health care, government and education. But the process may be useful for software engineering as well. The ISO 9126 model already defines a hierarchy for quality properties and is therefore well suited to apply the process.

After building the hierarchy, one-to-one comparisons are performed. In the case of the maintainability model, on the upper level, one example for a comparison is between analyzability and testability. Note that with an increasing amount of properties the number of the comparisons grows exponential. After performing all comparisons, the AHP converts them into a numerical value. The result is a numerical weight for each element in the hierarchy. Furthermore, for each individual participating person a consistency ratio can be calculated. This specific value shows conflicts in the one-to-one comparisons. Assume that one person states the following facts:

1. Analyzability is more important than testability.
2. Testability is more important Changeability.
3. Changeability is more important than analyzability.

Can this be a consistent assessment? No, because in the third point there is a contradiction. The value of the consistency ratio describes the trust into the assessment. Therefore the value can assist to find inconsistent assessments.

Note that the number of properties in the experiment was kept low, four properties on the upper level and four properties on the lower level were used. To guarantee simplicity and understandability of the model it is necessary to keep this in mind.

In the experiment a scale from 1 to 9 was used, in two directions. Figure 5.1 describes the used scale. If property *A* is considered more important than property *B*, the participants can express this with a positive value, where 2 means slightly more important and 9 means much more important. 1 is considered as equally important. If property *B* is considered to be more important, the same values are used with negative sign.

5.4 Validation of the Procedural Model

As described above, the weights assignment was done by a group of seven developers. The next step in the model construction process is the validation. For this purpose an experiment with



Figure 5.1: Weighing Process Description

another *different* developer group was performed. The experiment is depicted in Figure 5.2. Different pieces of source code (two modules in C and two other modules in C++) were presented to the developers together with a questionnaire they had to fill. The pieces of source code were chosen in a way, that they lead to different results applying the procedural model presented including the weights in 6. The questionnaire was constructed in a way that the developers had to assess the maintainability by investigating the source code. In the first step they had to consider the overall module and assess the sub characteristics presented in the ISO 9126 model and the overall maintainability. In a second step four functions per module were chosen and the exercise was to rank these functions. The developers were also asked to comment on their decisions. Especially these comments give hints on the aspects that can't be covered by static software analysis as can be seen later on. The aim of this experiment was to prove the following hypothesis:

The model will rank the chosen source code functions in the same order as the developers do by filling the questionnaire, in the context of maintainability.

Furthermore, it is possible to investigate if the developers themselves agree on the same order or if there are significant different perceptions of maintainability. The used questionnaire can be found in the appendix. The results are presented and discussed in Chapter 6. In the next subsection, the used source code modules are examined in detail.

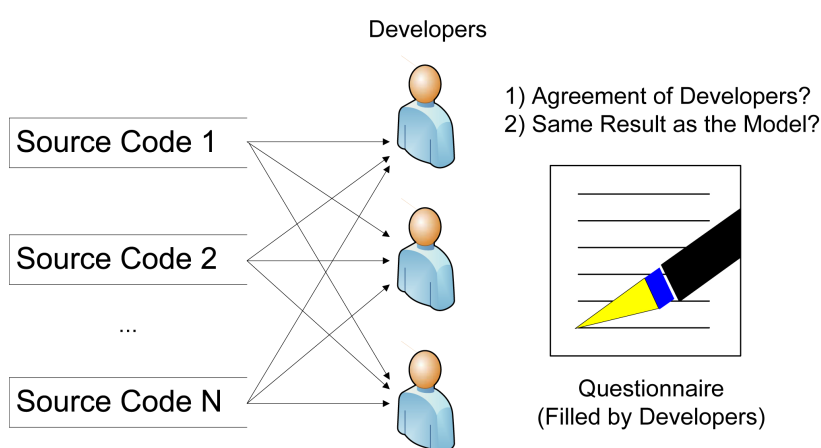


Figure 5.2: Depiction of Validation Experiment

5.4.1 Description of the Source Code Modules

The experiment was performed with four different source code modules. Two of the modules written in C and two of them written in C++. The modules written in C were independent from each other.

The *tsearch* module can be found online [5] and offers a tree search for red/black trees. These trees are binary trees in which the edges are colored either red or black. *bleConverter* is a module for handling the little/big endian conversion of mails, respectively data blocks. Note that these two modules implement *different* functionality.

SockManager1 and *SockManager2* are especially adequate for this experiment as they implement the *same* functionality. The second module was developed with the intention to improve the maintainability of the first one. Therefore, this experiment also shows if the improvement process was successful or not. The modules are part of a library for message communication.

After choosing the modules, the examination for functions that could be ranked in the modules was done. This was done in a way, that they differed from each other considering the metrics used in the proposed procedural model. In Tables 5.1, 5.2, 5.3 and 5.4 the selected functions are given with their metrics. Also the value for maintainability applying the proposed procedural module (MM) and the ranking using the proposed procedural model is stated (MR). Note that some functions are ranked as equal by the model ¹. How to calculate these values will be shown in the results in section 6. To perform the calculation, thresholds for each metric will be needed. Furthermore, the weights obtained from the former presented experiment will be used.

Note that the Nesting Depth (ND) could not be measured directly by the tools in use. Only a threshold could be given to the metrics extractors. Therefore, it was only obtained if the ND was above or below a certain value (in this concrete case the value five).

Function	LOC	NOI	NOR	ND	MM	MR
<i>tsearch</i> ₁	57	6	1	≥ 5	0.44	3
<i>tsearch</i> ₂	40	3	3	< 5	0.70	1*
<i>tsearch</i> ₃	176	3	4	≥ 5	0.12	4
<i>tsearch</i> ₄	16	3	3	< 5	0.70	1*

Table 5.1: Characteristics of selected Functions in Module *tsearch*

Function	LOC	NOI	NOR	ND	MM	MR
<i>bleConverter</i> ₁	23	5	1	< 5	1	1*
<i>bleConverter</i> ₂	45	3	1	≥ 5	0.77	3
<i>bleConverter</i> ₃	69	4	2	< 5	0.70	4
<i>bleConverter</i> ₄	23	5	1	< 5	1	1*

Table 5.2: Characteristics of selected Functions in Module *bleConverter*

¹marked with *

Function	LOC	NOI	NOR	ND	MM	MR
<i>SockManager1</i> ₁	14	1	1	< 5	1	1*
<i>SockManager1</i> ₂	43	3	1	< 5	1	1*
<i>SockManager1</i> ₃	60	1	1	≥ 5	0.77	3*
<i>SockManager1</i> ₄	37	1	1	≥ 5	0.77	3*

Table 5.3: Characteristics of selected Functions in Module *SockManager*₁

Function	LOC	NOI	NOR	ND	MM	MR
<i>SockManager2</i> ₁	33	2	1	< 5	1	1
<i>SockManager2</i> ₂	66	4	1	≥ 5	0.77	3
<i>SockManager2</i> ₃	77	2	1	< 5	0.86	2
<i>SockManager2</i> ₄	153	0	1	≥ 5	0.63	4

Table 5.4: Characteristics of selected Functions in Module *SockManager*₂

5.4.2 Criticism of the Experiments

The assigning of the weights to the metrics was the consensus of a special developer group. As already mentioned, the validation experiment was done with a second, *different* developer group. Now there is no reason to assume that the application of the model, created by the first group, should lead to the same result as the validation experiment with the second group. The second group could have created a different model with different weights which would represent their viewpoint. This criticism is true. Every created model can only be valid in a certain environment. The first step is to reach consensus about what a certain characteristic means to the involved persons. To give an example in the context of maintainability, some developers could say that more return values are preferable, others could say that less return values are preferable. The problem has to be discussed and one way has to be chosen. Everybody has to adhere to the decision after.

Even so, it is interesting to see if the viewpoint of the second group is completely different or similar. Furthermore, inside the second group, it can be seen if the common perception of maintainability is similar or if everybody shares a different notion of maintainability.

6 Results

In the former section, several experiments were described. In this section, the results are considered, beginning with the obtained weights for the metrics and characteristics of the proposed models. It will be shown how to assess functions using the procedural model by introducing thresholds for each metric in use. Furthermore, the outcome of the validation experiment will be examined.

6.1 Weights for the Procedural Model

In Figure 6.1, the detailed results of the Analytical Hierarchy Process (AHP) for each individual developer can be seen for the procedural model. In Figure 6.2 the resulting model is given.

For the changeability characteristic the Nesting Depth (ND) is a significant measure. This seems plausible. The mental effort to alter a function with a high ND is higher, as the scenarios to reach a certain element has to be captured by the developer. The developer also has to keep in mind that he could influence elements with higher ND than the one he changes.

For the characteristics stability and testability the interface complexity is the meaningful measure. Also this is intuitive. From the viewpoint of testing, considering a function as a black box, the only relevant information is the interface. From the viewpoint of stability, changing a function's interface may lead to undesired behavior in other parts of the system.

For the analyzability characteristic the LOC and NOR metrics are the important ones. This seems plausible as well. Imagine to find the code that has to be altered in a system to reach a new functionality. The bigger the system, the harder the relevant code to identify. If the relevant functions are identified, the next step is to determine how the return value of this function should be to reach the new desired behavior. If the exit point of the function is not clear, this may indicate more effort to change the original function.

6.2 Defining Thresholds for the Procedural Model

What we have so far is the value of metrics that can be measured by tools and the values to weight them. But to apply the model in practice, one step is missing. This step is the interpretation

	D1	D2	D3	D4	D5	D6	D7	Avg	Stdv
Analyzability									
LOC	12.14%	39.82%	19.80%	12.91%	20.83%	62.21%	12.47%	25.74%	18.77%
NOI	24.57%	21.79%	25.08%	41.41%	43.99%	6.07%	4.45%	23.91%	15.35%
NOR	59.43%	8.51%	30.08%	29.19%	23.61%	10.43%	43.04%	29.18%	17.88%
ND	3.86%	29.88%	25.04%	16.50%	11.57%	21.29%	40.04%	21.17%	11.98%
CR	13.04%	7.96%	30.76%	22.17%	13.83%	10.04%	0.47%		
Changeability									
LOC	5.05%	44.28%	7.11%	9.54%	5.88%	24.31%	5.60%	14.54%	14.75%
NOI	59.76%	20.37%	28.98%	48.47%	35.31%	11.81%	21.53%	32.32%	16.90%
NOR	29.98%	6.65%	12.69%	28.51%	10.29%	17.36%	52.82%	22.61%	15.99%
ND	5.21%	28.70%	51.22%	13.49%	48.52%	46.53%	20.06%	30.53%	18.50%
CR	14.91%	14.95%	14.71%	25.16%	4.75%	44.44%	2.95%		
Stability									
LOC	8.85%	9.73%	4.60%	8.39%	6.90%	6.34%	4.68%	7.07%	2.01%
NOI	44.27%	53.62%	29.34%	49.92%	36.29%	42.70%	15.20%	38.76%	13.17%
NOR	31.77%	12.92%	32.90%	29.72%	15.09%	44.37%	63.94%	32.96%	17.43%
ND	15.10%	23.73%	33.16%	11.98%	41.72%	6.60%	16.17%	21.21%	12.46%
CR	12.96%	14.52%	22.74%	11.58%	7.89%	0.14%	14.50%		
Testability									
LOC	7.18%	15.20%	4.07%	7.90%	4.33%	5.30%	3.84%	6.83%	4.01%
NOI	41.32%	45.81%	19.53%	37.22%	44.75%	41.49%	18.86%	35.57%	11.52%
NOR	41.32%	21.82%	40.79%	37.22%	6.16%	38.86%	60.07%	35.18%	16.97%
ND	10.19%	17.17%	35.61%	17.65%	44.75%	14.35%	17.23%	22.42%	12.68%
CR	3.25%	35.73%	16.47%	0.29%	3.72%	19.02%	17.38%		

LOC Lines of Code
 NOI Number of Input Parameters
 NOR Number of Return Points
 ND Nesting Depth
 CR Consistency Ratio, Specific Value of the Analytical Hierarchy Process

 Consistency Ratio above 15 Percent
 Avg Average
 Stdv Standard Deviation
 D Developer

Figure 6.1: Detailed Results for the Individual Developers, Procedural Model

of the metrics itself. What is a *good* value for *LOC*? What is a *good* value for *NOI*? The easiest way to do this is a black/white mapping. This means that a value under a certain threshold is considered as good, others as bad. The values used for the procedural model in the context of this thesis is summarized in Table 6.1. If for example a function has less than 70 *LOC* and *NOR* of one, but *NOI* of more than six and a *ND* of more than five, the analyzability would result in $0.26 + 0.29 = 0.55$. Note that the range of the aggregated value for each characteristic is from 0 to 1. Applying the weights (which sum up to 1 as well) after by multiplication, the overall value for maintainability can only vary between 0 and 1 as well.

The following general formula describes the assessment process:

Maintainability															
0.27				0.21				0.40				0.12			
Analyzability				Changeability				Stability				Testability			
LOC	NOI	NOR	ND	LOC	NOI	NOR	ND	LOC	NOI	NOR	ND	LOC	NOI	NOR	ND
0.26	0.24	0.29	0.21	0.15	0.32	0.23	0.30	0.07	0.39	0.33	0.21	0.08	0.35	0.35	0.22

Figure 6.2: Proposed Procedural Model for Maintainability with Weights Obtained by the AHP

$$\sum_{i=1}^{N_c} W_i \times \sum_{j=1}^{N_i} W_{i,j} \times t(M_{i,j}(x)) \quad (6.1)$$

where

- N_C is the number of characteristics in the model
- N_i is the number of metrics for an individual characteristic
- W_i is the weight for each individual characteristic
- $W_{i,j}$ is the weight for a metric inside a characteristic
- $M_{i,j}$ is the measured value for the metric in use inside a characteristic
- x is the considered function
- t is the threshold function, mapping values above (under) the threshold to 0 respectively 1

LOC	NOI	NOR	ND
< 70	< 6	≤ 1	< 5

Table 6.1: Mapping (Interpretation) of Metrics for the Procedural Model

The consequence of this coarse grained approach is that minor differences in functions are not captured, resulting in the same value of maintainability. If a function is just under the threshold for all metrics, it will give the same value like a function that is significantly under the threshold. Introducing more intervals would give a more fine grained approach on the cost of a more complicated model. For now, we stick to the coarse grained approach keeping the disadvantage in mind.

Another problem is to choose the thresholds. The chosen values are based on experts opinions. A different idea based on algorithms from Machine Learning (ML) will be presented in Section 7, where ideas for future work are given.

	D1	D2	D3	D4	D5	D6	D7	Avg	Stdv
Maintainability									
Analyzability	61.02%	17.21%	44.73%	19.68%	17.53%	11.13%	15.05%	26.62%	18.73%
Changeability	13.81%	16.77%	33.26%	8.41%	11.22%	47.92%	15.69%	21.01%	14.30%
Stability	19.56%	58.42%	12.85%	63.50%	50.78%	32.78%	47.43%	40.76%	19.44%
Testability	5.62%	7.60%	9.16%	8.41%	20.47%	8.17%	21.83%	11.61%	6.62%
CR	74.23%	16.50%	7.65%	11.53%	20.97%	17.47%	13.54%		
Analyzability									
CBO	44.82%	24.20%	18.10%	40.80%	36.84%	22.22%	60.91%	35.41%	15.11%
DIT	14.74%	22.09%	35.84%	14.67%	26.86%	44.44%	13.90%	24.65%	11.85%
NOC	6.28%	12.73%	31.99%	23.31%	21.06%	22.22%	16.79%	19.20%	8.24%
NOMV	34.16%	40.98%	14.07%	21.23%	15.24%	11.11%	8.40%	20.74%	12.31%
CR	5.68%	34.71%	5.06%	54.82%	25.34%	32.92%	6.23%		
Changeability									
CBO	19.53%	50.02%	28.41%	47.05%	36.35%	13.02%	54.44%	35.55%	15.89%
DIT	37.49%	6.77%	44.62%	32.15%	15.90%	16.15%	16.12%	24.17%	13.90%
NOC	32.49%	29.18%	18.26%	12.86%	22.76%	22.40%	18.61%	22.36%	6.71%
NOMV	10.50%	14.04%	8.71%	7.94%	24.99%	48.44%	10.83%	17.92%	14.64%
CR	10.66%	24.89%	23.34%	14.01%	13.35%	6.43%	40.94%		
Stability									
CBO	10.32%	50.12%	14.03%	32.33%	11.71%	39.54%	47.61%	29.38%	17.25%
DIT	31.97%	16.96%	42.03%	38.29%	40.33%	11.45%	9.45%	27.21%	14.18%
NOC	51.65%	25.89%	39.97%	21.29%	33.98%	24.75%	32.83%	32.91%	10.43%
NOMV	6.06%	7.03%	3.97%	8.09%	13.99%	24.26%	10.11%	10.50%	6.85%
CR	11.38%	24.58%	21.67%	10.25%	1.39%	24.66%	14.20%		
Testability									
CBO	48.19%	43.18%	56.28%	43.75%	16.67%	53.70%	70.57%	47.48%	16.49%
DIT	6.81%	15.12%	20.10%	29.00%	33.33%	25.58%	10.51%	20.06%	9.81%
NOC	6.81%	35.47%	20.10%	19.36%	33.33%	5.75%	13.17%	19.14%	11.80%
NOMV	38.19%	6.23%	3.53%	7.89%	16.67%	14.97%	5.75%	13.32%	12.01%
CR	1.58%	7.18%	12.97%	14.61%	0.00%	42.42%	5.53%		

CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
NOC	Number of Children
NOMV	Number of Member Variables
CR	Consistency Ratio, Specific Value of the Analytical Hierarchy Process

	Consistency Ratio above 15 Percent
Avg	Average
Stdv	Standard Deviation

Figure 6.3: Detailed Results for the Individual Developers, Overall Maintainability and OO Model

6.3 Weights for the OO Model

In Figure 6.3, the detailed results of the AHP process for the OO model and the overall maintainability characteristic is shown. In Figure 6.4 the resulting model is given.

Considering the result of the experiment, the first observation is that the CBO metric is of particular importance. In three of the four categories it is to be considered as the most significant measure. Especially in the category testability the distance to the other metrics is very high. This can be explained by considering Figure 6.5. To test a component (Device Under Test) of a system that is coupled to other components, simulators are needed. The simulators simulate the

Maintainability																															
0.27				0.21				0.40				0.12																			
Analyzability				Changeability				Stability				Testability																			
CBO	0.35	DIT	0.25	NOC	0.19	NOMV	0.21	CBO	0.36	DIT	0.24	NOC	0.22	NOMV	0.18	CBO	0.29	DIT	0.27	NOC	0.33	NOMV	0.11	CBO	0.48	DIT	0.20	NOC	0.19	NOMV	0.13

Figure 6.4: Proposed OO Model for Maintainability with weights obtained by the AHP

behavior of the coupled components that are not implemented yet. Therefore the more coupled elements the DUT has, the more effort has to be spent to create the simulators. For testing, also configuration and data are necessary which are ideally separated. Furthermore, a component that drives the test (the test driver) is needed.

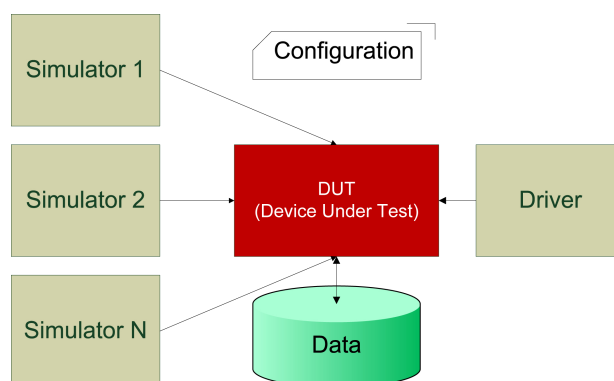


Figure 6.5: Necessary Elements for a Component Test

As mentioned above, in three of four categories the CBO measure is the most significant metric. Only concerning stability, the NOC metric is rated as the most meaningful measure. This is also what intuition tells us. Altering a class with a high number of children will potentially influence them, leading to instability.

The NOMV metric is ranked on the last position in every category, except in analyzability. This seems to be plausible as well. In classes with a high amount of member variables, it may take more time to identify the variables that have to be modified.

6.4 Defining Thresholds for the OO Model

The same step done for the procedural model is done for the OO here as well. Thresholds to interpret the metrics have to be defined. The familiarity with OO metrics is usually low. Therefore it is hard to ask developers for values which they would consider as advisable. The question is where to get reference values. In [NAS01] a study can be found to assist in answering this question. Three systems were analyzed for this purpose summarized in table 6.2.

The last two rows were obtained by a specific analysis tool and the intention is to describe the quality.

System	A	B	C
Lines of Code	50k	300k	500k
Number of Classes	46	1000	1617
Language	Java	Java	C++
Type of Application	Commercial	NASA	NASA
Code Construct	OO	Excellent OO	Good OO
Quality	Low	High	Medium

Table 6.2: Analyzed Systems by [NAS01]

To get a feeling for the metrics in use by the proposed OO model, a statistical analysis performed by [NAS01] is used. In tables 6.3, 6.4 and 6.5 the minimum, maximum, mean and standard deviation are summarized.

	Minimum	Maximum	Mean	Standard Deviation
CBO	0	11	2.48	2.93
NOC	0	2	0.07	0.32
DIT	0	2	0.37	0.53

Table 6.3: Statistical Values for System A

	Minimum	Maximum	Mean	Standard Deviation
CBO	0	22	1.25	2.01
NOC	0	21	0.35	1.66
DIT	0	4	0.97	0.69

Table 6.4: Statistical Values for System B

	Minimum	Maximum	Mean	Standard Deviation
CBO	0	16	2.09	2.05
NOC	0	116	0.39	3.22
DIT	0	6	1.02	0.98

Table 6.5: Statistical Values for System C

As system *B* is considered as the best quality system, the focus is set on table 6.4, keeping in mind that this is a special classification by NASA. Note that no values for the NOMV metric were analyzed in the given study. Therefore, the threshold for the NOMV metric is an estimation of the author.

The values given in Table 6.6 are based on the reference values from above (except NOMV). It is obvious that they will change based on experiences. But to start the analysis with the proposed OO model, it is necessary to start with estimations.

CBO	DIT	NOC	NOMV
< 12	< 5	< 10	< 10

Table 6.6: Mapping (Interpretation) of Metrics for the OO Model

6.5 The Weights for the ISO 9126 Characteristics

A surprising fact is that testability is considered of low importance with a factor of 0.12. This is due to the reason, that the developers mainly test code written by themselves. Therefore it seems not import to them to develop code that can easily by tested by others. The perception is *I know my code, so I will be able to test it*. This fact can lead to problems as there will be inevitable situations in which the source code will be maintained by different developers. Furthermore, the process of testing has gained increasing attention through the last years. One reason therefore is the growing complexity of the implemented systems. This fact should be kept in mind and actions to meet this challenge should be performed. Actions like Unit Testing and Test Driven Development (TDD) are technologies that can support in these situations.

6.6 Criticism of the Former Results

After considering the former results, there are several points of criticism that should be examined. As can be seen, the results of the AHP for each specific developer are varying. This may indicate that the perception of maintainability is specific for each developer. Also the consistency ratio (CR) should be considered more detailed. It is a AHP specific measure that indicates if an assessment is consistent or suffers from contradictions. This can be explained by the fact, that this was the first time that the developers were faced with the field of metrics. Form the statistical point of view, the experiment was performed with seven developers, this sample size is surely too small to infer on the generality. But it will be hard to take the time of more developers for this kind of experiments.

6.7 Results of Validating the Procedural Model

In Section 5 the experiment for validating the procedural model was described and following hypothesis was stated:

The model will rank the chosen source code functions in the same order as the developers do by filling the questionnaire, in the context of maintainability.

In this step the results of the experiment are compared with the results from the model, beginning with the functions from the module *tsearch* in Table 6.7.

6.7.1 Functional View

MM describes the maintainability value that is calculated from the model as explained in 6.2. Note that this value can only vary between 0 and 1. *MR* is the ranking that results from the

model. EM_{Avg} is the average value of the ranking experiment from the developers. Note that this value can vary from 1 to 5, following the Austrian school grading system. As the average value is not a meaningful measure concerning the different views of the developers (consensus), also the standard deviation is given denoted with EM_{Stdv} .

Function	LOC	NOI	NOR	ND	MM	MR	EM_{Avg}	EM_{Stdv}	ER
<i>tsearch</i> ₁	57	6	1	≥ 5	0.44	3	2.43	0.98	3
<i>tsearch</i> ₂	40	3	3	< 5	0.70	1*	2.29	0.49	2
<i>tsearch</i> ₃	176	3	4	≥ 5	0.12	4	4.00	0.00	4
<i>tsearch</i> ₄	16	3	3	< 5	0.70	1*	1.29	0.49	1

Table 6.7: Comparison of Results for Module *tsearch*

What can be seen from the results is that the ranking from the experiment matches with the ranking from the model very well. As the model is not fine grained enough at the moment, some functions are ranked as equal¹. The use of a more accurate mapping than the black/white approach would solve this problem at the price of a more complicated model. Considering the values from the standard deviation all developers agree that the function *tsearch*₃ suffers of bad maintainability. Looking at the comments they justified their choice in the following way:

- Function too long
- Contains multiple returns
- Bad naming of variables
- Huge complexity and various if blocks

The first two points are captured by the model directly. The naming of the variables is surely relevant for maintainability, but how to capture this property? The semantic of a name to a developer can't be measured directly. Here we observe the limits of metrics, some properties are simply not measurable. The same is true for the complexity, it was already shown that complexity can not be put into a single number. The nesting depth can only give hints about it! Even so, the result for this module is promising.

Considering the results of the module *bleConverter* in Table 6.8 the results match well again. What can be seen from function *bleConverter*₄ is that there is a high standard deviation. Looking at the justifications, the reason is that some developers consider very special details like one reason was *No 64 Bit Support*. Another point that lead to many discussions is the quality of the source code comments. Again, the semantic of a source code comment to a developer can't be measured and is not captured by the model. Even introducing a measure for the amount of lines of comments in the source would not solve the problem. Great comments for one developer may be worthless to another one. One developer even answered that there were too many comments in the code and for him this is distracting.

Even with all the non captured properties the ranking from the experiment matches well with the ranking from the model. This may be a hint that the model provides a good compromise between complexity and accuracy.

¹denoted with *

Function	LOC	NOI	NOR	ND	MM	MR	EM_{Avg}	EM_{Stdv}	ER
<i>bleConverter</i> ₁	23	5	1	< 5	1.00	1*	1.71	0.76	1
<i>bleConverter</i> ₂	45	3	1	≥ 5	0.77	3	2.71	0.95	3
<i>bleConverter</i> ₃	69	4	2	< 5	0.70	4	3.43	0.79	4
<i>bleConverter</i> ₄	23	5	1	< 5	1.00	1*	2.14	1.35	2

Table 6.8: Comparison of Results for Module *bleConverter*

In Table 6.9 the results for the module *SockManager1* are given. In this case high standard deviations are noticeable. The different opinions are justified by the following points:

- Bad comment quality
- Complexity
- Hard coded constants

These properties are observed different for each developer. They are not measured directly by the model. This again shows us that even with the best model it is not possible to satisfy every developer's perception. Every model can only work in a certain context where the model is developed together and accepted from the participating persons.

Function	LOC	NOI	NOR	ND	MM	MR	EM_{Avg}	EM_{Stdv}	ER
<i>SockManager</i> ₁	14	1	1	< 5	1.00	1*	1.57	1.13	1
<i>SockManager</i> ₂	43	3	1	< 5	1.00	1*	2.71	1.11	3
<i>SockManager</i> ₃	60	1	1	≥ 5	0.77	3*	2.57	1.13	2
<i>SockManager</i> ₄	37	1	1	≥ 5	0.77	3*	3.14	0.69	4

Table 6.9: Comparison of Results for Module *SockManager1*

Finally the results for the functions in the module *SockManager2* are given in table 6.10. As can be seen the functions on the first and last position are ranked equal from the model and from the experiment. The function *SockManager2*₃ shows high standard deviation. The reason are the same as already mentioned in the other cases. There are different perceptions about complexity and comments. The ranking from the model is therefore slightly different from the experiment ranking.

Function	LOC	NOI	NOR	ND	MM	MR	EM_{Avg}	EM_{Stdv}	ER
<i>SockManager</i> ₂ ₁	33	2	1	< 5	1.00	1	1.14	0.38	1
<i>SockManager</i> ₂ ₂	66	4	1	≥ 5	0.77	3	2.43	0.53	2
<i>SockManager</i> ₂ ₃	77	2	1	< 5	0.86	2	2.86	1.07	3
<i>SockManager</i> ₂ ₄	153	0	1	≥ 5	0.63	4	3.57	0.79	4

Table 6.10: Comparison of Results for Module *SockManager2*

6.7.2 The Overall View

The developers in the experiment were also asked to assess the maintainability characteristics according to the ISO 9126 standard on the overall modules. The used grading scheme was again the Austrian school grading system from 1 (best) to 5 (worst). The individual gradings are given in Table 6.11, the summarized result with mean and standard deviation is depicted in Figure 6.6.

Module	D1	D2	D3	D4	D5	D6	D7
<i>tsearch</i>	3	4	3	4	4	2	5
<i>bleConverter</i>	1	3	2	1	2	2	1
<i>SockManager1</i>	4	1	3	2	2	3	2
<i>SockManager2</i>	4	3	5	3	3	3	4

Table 6.11: Individual Gradings of Developers for Overall Maintainability

An interesting phenomenon could be seen when considering the detailed results of the individual developers. They share a different offset for what is good maintainability and what is bad maintainability. What is meant by this is that if they compare two modules, they always agreed that *SockManager1* is better than *SockManager2*, but expressing this with very different grades. For example for one developer the difference in *SockManager1* and *SockManager2* was expressed by grading them with 1 and 3, whereas another one graded the modules with 3 and 5. Therefore the standard deviations can be high in this case.

Another important comment the developers gave was that some modules can not be compared, because they implement different functionality. This is the case for *tsearch* and *bleConverter*. Considering this, the consequence is that it is valid to compare different versions of the same module, but not comparing modules implementing different functionality.

Characteristic / Module	tsearch		bleConverter		SockManager ₁ (new)		SockManager ₂ (old)	
	Avg	Stdv	Avg	Stdv	Avg	Stdv	Avg	Stdv
Analyzability	3.86	1.21	1.71	0.95	2.29	0.95	3.43	0.53
Changeability	4.00	1.15	1.57	0.79	2.29	1.11	3.43	0.53
Stability	3.00	1.29	1.57	0.53	2.29	1.11	3.29	1.11
Testability	2.43	0.79	1.14	0.38	2.86	1.35	3.57	0.98
Maintainability (Overall)	3.57	0.98	1.71	0.76	2.43	0.98	3.57	0.79

Avg... Average
Stdv... Standard Deviation

Figure 6.6: Overall Maintainability, View of Developers

6.8 Agreement and Disagreement of Developers

Another question to be examined is, if the developers share a common notion of maintainability. In Table 6.11 the results of the individual rankings concerning the overall maintainability characteristic were already given, followed by some discussion. Now, a statistical evaluation is of interest. Therefore, it is necessary to find a convenient tool for analyzing the data. In [KS39] *The Problem of m Rankings* is considered. Given n objects ranked by m persons, the question if

the set of rankings shows any evidence of community of judgment among the developers has to be answered. In the former experiment, n refers to the four source code modules and m refers to the seven participating developers.

The Kendall rank correlation coefficient, more commonly referred to as Kendall's tau (τ) coefficient or a tau test, is a measure for rank correlation [Ken38]. It will take values between minus one and one. A positive correlation indicates that the ranks of both variables increase together. A negative correlation indicates that as the rank of one variable increases, the other one decreases. In the source code experiment, a positive value means that the considered developers share a similar notion of maintainability, whereas a negative value means that they share a contrary notion of maintainability.

To proceed, some definitions underlying Kendall's tau have to be introduced. If (x_j, y_j) and (x_k, y_k) are two elements of a sample from a bivariate population, (x_j, y_j) and (x_k, y_k) are *concordant* if $x_j < x_k$ and $y_j < y_k$ or if $x_j > x_k$ and $y_j > y_k$. The two elements are *discordant*, if $x_j < x_k$ and $y_j > y_k$ or if $x_j > x_k$ and $y_j < y_k$. The difference between the number of concordant pairs c and the number of discordant pairs d is denoted by S . There are $\binom{n}{2}$ distinct pairs of observations in the sample. Each pair, except ties, is either concordant or discordant. Kendall's tau is defined as

$$\tau_n = \frac{c - d}{c + d} = \frac{S}{\binom{n}{2}} = \frac{2S}{n(n-1)}. \quad (6.2)$$

For the appearance of ties, adjusted formulas were introduced (for details see [Ken38]). Kendall's tau is especially convenient for small sample sizes and for non-equidistant scales. To compute the values for each pair of developers, a convenient tool can be found in [1]. As tool input the dataset given in table 6.11 was used. The result given in Figure 6.7 shows the values for Kendall's tau for all pairs of developers.

	D2	D3	D4	D5	D6	D7
D1	-0.40	0.80	0.18	0.00	0.89	0.18
D2		0.00	0.55	0.80	-0.67	0.55
D3			0.55	0.40	0.67	0.55
D4				0.91	0.00	1.00
D5					-0.22	0.91
D6						0.00

Figure 6.7: Kendall's tau for Developer Pairs

Considering the pair $(D2, D6)$, it can be seen that the maintainability perception is highly contrary. Two further combinations result in a negative value. Four combinations show no correlation. The majority of the developer pairs show a positive value, indicating a common perception of maintainability. Note that the amount of considered source code modules ($n = 4$) is very small. Therefore, the results have to be considered cautious. It is common to give the p-value together with Kendall's tau, which describes the error probability. To give significant statements, the p-value should not exceed five percent. However, for small values of n , the p-value will exceed this guideline. In Figure 6.8 the scatterplot constructed with [1] is depicted. Note that the p-values are exceptionally high as only for source code modules were considered in the experiment.

The diagonal of the matrix shows the histogram of each data series. The scatterplots (and smooth curve) are depicted in the upper half of the matrix. Every combination of pairs is considered.

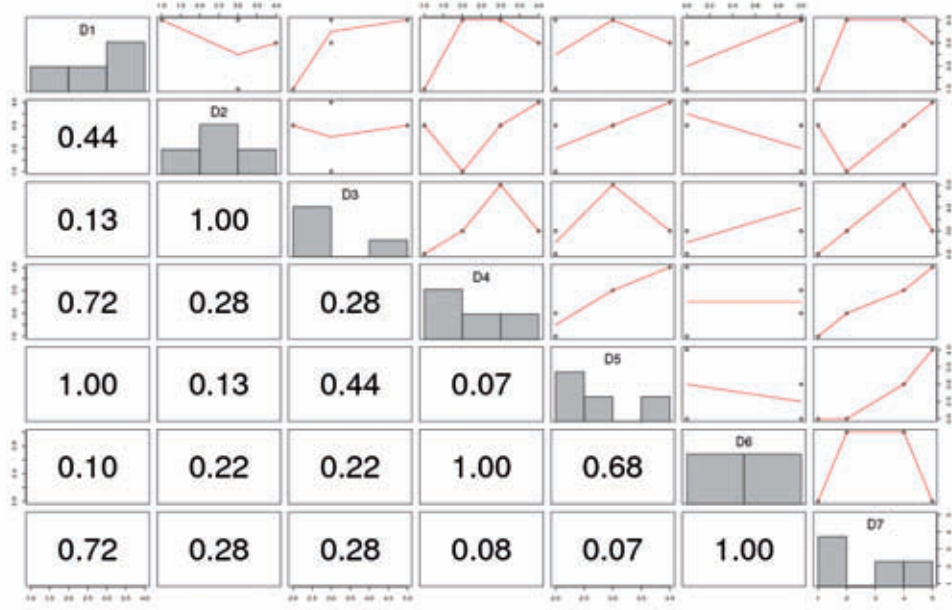


Figure 6.8: Kendall's tau: Scatterplot with Histogram, Smooth Curve and Error Probabilities

In the lower half of the matrix the p-value of the Kendall tau rank correlation is stated. The scatterplot for $(D4, D6)$ shows a line with no slope ($\tau = 0$). The scatterplot $(D4, D7)$ shows an arithmetically increasing slope ($\tau = 1$), whereas $(D2, D6)$ depicts the opposite ($\tau = -0.67$).

So far, developer pairs were analyzed. In a next step the overall trend is examined. *Kendall's Coefficient of Concordance*, also known as *Kendall's W*, can be used for determining agreement among raters. It is defined as [KS39]

$$W = \frac{12S}{m^2(n^3 - n)} \quad (6.3)$$

where m and n have the same meaning like in the definition of Kendall's tau. To define S , further terms are introduced.

Supposing object (source code module) i is given the rank $r_{i,j}$ by judge (developer, person) number j . Then the total rank for object (source code module) i is $R_i = \sum_{j=1}^m r_{i,j}$. The mean value of the total ranks is $\bar{R} = \frac{1}{2}m(n+1)$. The input is modified before calculation, the data from each ranker (developer) are ranked. Therefore, Kendall's W can be applied to scores, measurements, or ranks, and even if the scale of measurements used by different rankers are different. S is the sum of square deviations then, $S = \sum_{i=1}^n (R_i - \bar{R})^2$. The occurrence of ties reduces the value of W , therefore it is necessary to implement corrections in case of existence. The value for Kendall's W can vary between zero and one. If W is zero, then there is no overall trend of agreement among the developers. Intermediate values of W indicate a greater or lesser degree of consensus among the participating persons.

Revisiting table 6.11, Kendall's W is calculated to show the existence or non-existence of a trend in the maintainability perception. To perform the calculation, a tool which allows input transformation and calculation was used [9]. The transformed data is given in Table 6.13. Comparing

to Table 6.11, the given grades were converted to a zero based ranking and a simple correction in the case of ties was performed. In Table 6.12, guidelines for the interpretation of Kendall's W are given.

Kendall's W	Interpretation	Assuredness of Arrangement Factors
0.1	Very Weak Consensus	Not Existing
0.3	Weak Consensus	Minimal
0.5	Medium Consensus	Average
0.7	Strong Consensus	High
0.9	Very Strong Consensus	Very High

Table 6.12: Interpretation of Kendall's W , from [Sch97]

The analyzed data results in $W = 0.4978$ and $p < 0.01$, indicating a moderate level of agreement among the participating developers with an error probability of less than one percent. This result is promising. It shows that there is a common trend. Experiments with more developers and more source code modules would be needed to confirm the result. The Kendall W offers a proper tool for the evaluation of such experiments.

Module	D1	D2	D3	D4	D5	D6	D7
<i>tsearch</i>	1	3	1.5	3	3	0.5	3
<i>bleConverter</i>	0	1.5	0	0	0.5	0.5	0
<i>SockManager1</i>	2.5	0	1.5	1	0.5	2.5	1
<i>SockManager2</i>	2.5	1.5	3	2	2	2.5	2

Table 6.13: Transformed Gradings

7 Conclusion

In this chapter, ideas which arose in previous sections are summarized and approaches for future work are given. Key challenges were to choose convenient metrics and to weigh them, especially to answer these questions further research is of interest. Concerning the prototype implementation, further requirements will be stated.

The focus of the thesis was on the source code level in the implementation phase. Applying metrics is not restricted to this phase. The earlier a weakness in the development life cycle is localized, the smaller is the cost to correct it. Therefore, it may be worth to use metrics already in the requirements phase. Continuing the idea, a quality model based on metrics that is put on the overall software development process would result. The necessary additional tool support could be integrated in the presented portal. But the additional effort and the disputed usefulness of metrics may prevent companies from introducing metrics into their strategy. Many experts doubt that the benefit would exceed the cost. Thinking of projects where high maintenance effort is expected, the probability is higher that the metric approach succeeds.

The maintainability model presented is surely not the only possible solution. Because of time constraints experiments that are needed to confirm the usefulness of different models are hard to perform. Furthermore, the definition of models for other characteristics would be helpful. The ISO 9126 standard can serve as a starting point for this task.

A prerequisite for assessing software development artifacts is the possibility to measure the needed metrics. There are hardly any tools to find that can measure all desired values. In the implemented prototype this problem was solved by combining multiple metrics extraction tools based on a specific parser for each tool. The integration of further tools will be necessary to enlarge the functionality and bring the idea to its full potential.

Concerning the implemented prototype system, the possibilities for further development are manifold. Using the SEAM technology the support for login and rights management is eased. Assigning rules to different persons, the integration of further tools and the enhancement of the user interface is of interest as well. In this thesis, static software analysis was considered. The prototype system is not restricted to static analysis. It is also possible to integrate dynamic aspects from dynamic software analysis tools. The combination could be another exciting approach.

To save time and resources the integration into a nightly build procedure should be possible. In the context of this thesis, this can be done by starting the individual parsers after the specific metrics extraction tools have finished the reports. If the nightly builds are done in regularly time

frequencies, it will be interesting to see how the source code changes by supporting the developers with the feedback obtained by the metric based quality models.

7.1 Future Work

A completely different approach for the problem how to weigh different metrics and find thresholds for special quality characteristics could come from the area of Machine Learning (ML). In the literature I could not find any experiments applying techniques from ML in this context. The principle is to use a set of training data samples that are classified by experts. Based on this data other data samples are predicted. The advantages of such an approach would be the high support of classification algorithms and the high accuracy that can be achieved by them with moderate computation effort. Problems might occur again with the time needed from the expert developers for classifying the training instances. As the model is only as good as the training data, this is crucial for the quality of the produced results. To get a better idea how the procedure works, a typical example of weather data is depicted in Table 7.1. The four attributes outlook, temperature, humidity and windy are listed. Furthermore, we want to predict if it is advisable to go for playing badminton or not. The data gives experiences from the past. Based on these experiences, we want to predict the future. The play-attribute is the one that is predicted.

Outlook	Temperature	Humidity	Windy	Play
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
overcast	hot	high	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
overcast	cool	normal	TRUE	yes
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
rainy	mild	normal	FALSE	yes
sunny	mild	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	TRUE	no

Table 7.1: Weather Dataset

Different attribute types are possible, in the case of weather data there are no numerical attributes, but nominal ones. Comparing to software metrics, the attributes of interest could be different metrics. The attribute to predict could be the maintainability. One could introduce different classes of maintainability, for example from 1 to 5. The question is how to get the data from the past in the case of software maintainability. As the implemented prototype system already contains the different metrics for software modules, an idea would be to ask the developers to

classify the code. This functionality could be integrated into the existing system. This is a time consuming task, as the programmer has to be familiar with the code. To reduce this time, only software modules could be used that the programmer worked on anyway. Meaning every time a developer finishes a task, he or she could be asked to classify the source code with respect to maintainability. After collecting enough data, the algorithms from the ML field could be used to find the most meaningful attributes and also create a model for the future. The existing tool support and the wealth of developed algorithms could achieve high accuracy. To demonstrate this, the WEKA tool [12] is used for the following experiments. WEKA is a collection of machine learning algorithms for data mining tasks. There are several possibilities for using the tool, ranging from GUI (Graphical User Interface) support to an API (Application Programming Interface). Using the WEKA-API, the ML functionality could be integrated into the implemented prototype presented in this thesis. In Figure 7.1 a very understandable technique is depicted denoted by *decision tree*. Using software metrics, it would be interesting to see if there is a possibility to classify code based on a tree. In this case the decision tree for the weather data results in bad performance, as the training data are only a few instances. But for certain problems accuracy might reach more than 90%.

Decision trees are a very popular classification method. The tree leaves represent the classes. The model is predictive and descriptive. It displays relationships found in the training data. The tree consists of zero or more internal nodes and one or more leaf nodes. Each internal node is a decision with two or more child nodes. The generated rules are easy to describe and understand and the approach does not make any assumptions about the underlying probability distributions of the attributes in use [Gup09].

Bayesian classification offers a different approach. A hypothesis that the given data belongs to a particular class is assumed. Then the probability of the hypothesis to be true is calculated. This approach requires only one scan of the whole data. Bayes' theorem assumes that all attributes are independent. This is a weakness as attributes are often correlated.

There are many other algorithms, each coming with different advantages and problems. To conclude, ML could be used to find meaningful metrics for a certain quality characteristic and to define metric thresholds. These are key challenges when working with software metrics and therefore it could be interesting to analyze the ML approach more detailed.

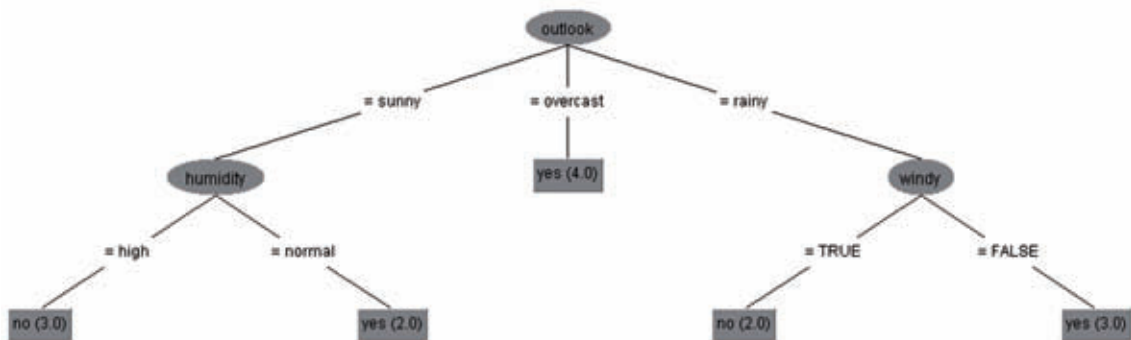


Figure 7.1: Decision Tree of Weather Data, produced with WEKA

7.2 Back to the Problem Statement

To recall the problem statement from Section 2.3, the aim of the thesis was to:

- Develop an applicable model to capture the maintainability property
- Choose convenient metrics and to weigh them
- Perform experiments based on the model
- Implement a prototype system which allows to:
 - Create user defined models based on characteristics and metrics
 - Define the weights for the metrics and characteristics in use
 - Assess source code based on the user specific models
 - Combine multiple metrics extraction tools

To solve these issues, the work began with studying related work. The Maintainability Index (MI) was examined and it was shown that key properties are not fulfilled by this measure. An approach based on the ISO 9126 standard seemed promising and puts key criteria that are necessary for successful integration of a maintainability model together. The problem with the standard was that there were no proposed metrics and weights how to capture the different characteristics. To overcome this gap, an experiment where the Analytical Hierarchy Process (AHP) was applied within a developer group coming from the safety critical field. Two different models were proposed in this way, one for the procedural and one for the object oriented paradigm. To proof the usefulness of the model, a second experiment with a different developer group was realized. The results suffer from the low sample size. Even so, the results give hints that the model makes sense.

The implemented system based on the SEAM technology supports the application of quality models. The user can create specific models for his need, define weights and characteristics and assess source code based on these models. The system was developed to a stage which shows that the architecture and the concept works. Multiple metrics extraction tools were combined to capture the necessary metrics and the design allows flexibility and extensibility. To combine multiple tools is important as there exist rarely metrics extraction tools that fulfill all desired needs of the developers. Furthermore, open source technologies assist to integrate metrics into the software development life cycle at an affordable effort.

7.3 Discussion

When introducing metrics into the software development life cycle, key challenges are to choose convenient metrics and to weigh them. Therefore, a consensus about the maintainability notion has to be found in advance. The experiments showed that there are slightly different perceptions about maintainability. Even so, the statistical evaluation confirms that a common trend exists, indicating that it is worth to invest in measuring maintainability by applying metrics.

When using metrics, it is of importance to distinguish between different paradigms like procedural and object oriented languages. Studies were shown in which this fact was ignored. This can lead to wrong conclusions.

A further challenge was to get the necessary metrics for the constructed maintainability models from a single tool. This shows the importance of integrating several tools into a portal. With this approach it is also possible to decrease license costs. Several open source tools can be used to get the desired metrics for a special quality model.

It could also be seen that several constructed models do not fulfill minimal requirements like simplicity and understandability. To establish a maintainability model, these aspects have to be kept in mind. The acceptance of a model can only be assured, if the participating parties are involved in the construction process.

The problems that could not be solved should not be hidden here. The experiments also showed that some properties simply can never be measured. Together with finding a consensus between participating persons in a developer group, this is a key issue. Therefore, the discussions about the use and usefulness of metrics will go on. The first goal should be to find consensus about the key properties that are responsible for good maintainability. Incorporating the developer groups into the model creation process can assist to reach consensus and to advance the sensibility for maintainability.

Another reason why the use of metrics is not common may be the fact, that the profit does not occur immediately in most cases. The time budget is restricted and one may think that the application of metrics is additional effort that can be skipped. If we think of how many times that source code produced by others already lead to problems to other developers or to a complete new version of a module, it may be worth to invest earlier and profit later. Or to say it with the *Meskimen's Law* [10]:

“There is never time to do it right, but always time to do it over.”

Literature

- [AGB03] AUER, M. ; GRASER, B. ; BIFFL, S.: A survey on the fitness of commercial software metric tools for service in heterogeneous environments: common pitfalls. In: *Software Metrics Symposium, 2003. Proceedings. Ninth International*, 2003. – ISSN 1530–1435, S. 144–152
- [AKCK05] AL-KILIDAR, Hiyam ; COX, Karl ; KITCHENHAM, Barbara: The use and usefulness of the ISO/IEC 9126 quality standard. In: *ISESE*, 2005, S. 126–132
- [All09] ALLEN, Dan: *SEAM in Action*. Manning, 2009
- [ARK05] ALGHAMDI, J.S. ; RUFAl, R.A. ; KHAN, S.M.: OOMeter: A Software Quality Assurance Tool. In: *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, 2005. – ISSN 1534–5351, S. 190–191
- [BD04] BRUNTINK, Magiel ; VAN DEURSEN, Arie: Predicting Class Testability Using Object-Oriented Metrics. In: *4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, Society Press, 2004, S. 136–145
- [Boo94] BOOCH, Grady: *Object-Oriented Analysis and Design with Applications*. 2nd. Redwood City, Calif. : Benjamin-Cummings, 1994
- [CALO94] COLEMAN, D. ; ASH, D. ; LOWTHER, B. ; OMAN, P.: Using metrics to evaluate software system maintainability. In: *Computer* 27 (1994), Aug, Nr. 8, S. 44–49. – ISSN 0018–9162
- [CK94] CHIDAMBER, S.R. ; KEMERER, C.F.: A Metrics Suite for Object Oriented Design. In: *IEEE Transactions on Software Engineering* 20 (1994), Nr. 6, S. 476–493. – ISSN 0098–5589
- [CS09] CHAHAL, Kuljit K. ; SINGH, Hardeep: Metrics to study symptoms of bad software designs. In: *SIGSOFT Softw. Eng. Notes* 34 (2009), Nr. 1, S. 1–4. – ISSN 0163–5948
- [Fen94] FENTON, N.: Software Measurement: A Necessary Scientific Basis. In: *IEEE Transactions on Software Engineering* 20 (1994), Nr. 3, S. 199–206. – ISSN 0098–5589
- [Ger07] GERSTINGER, A.: Interpretation of Software Quality Metrics for Safety-Critical Systems. In: *Proceedings of 25th International System Safety Conference, Baltimore, USA, August 13-17, 2007*, System Safety Society, 2007

- [Gup09] GUPTA, G. K.: *Introduction To Data Mining With Case Studies*. 2009
- [Hal77] HALSTEAD, Maurice H.: *Elements of software science / Maurice H. Halstead*. Elsevier, New York, 1977
- [HKV07] HEITLAGER, I. ; KUIPERS, T. ; VISSER, J.: A Practical Model for Measuring Maintainability. In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, 2007, S. 30–39
- [IEE90] IEEE: IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std 610.12-1990* (1990), S. 1
- [IEE93] IEEE. *IEEE Software Engineering Standards Collection*. 1993
- [Ken38] KENDALL, M.G.: A new measure of rank correlation. In: *Biometrika* 30 (1938), S. 81–93
- [Kie06] KIEVIET, Michael: *Analyse über Qualifizierungsmethoden sicherheitsrelevanter Embedded Software*, Diplomarbeit, 2006
- [KS39] KENDALL, M. G. ; SMITH, B. B. *The Problem of m Rankings*. 1939
- [Leh80] LEHMAN, M.M.: Programs, life cycles, and laws of software evolution. In: *Proceedings of the IEEE* 68 (1980), September, Nr. 9, S. 1060 – 1076. – ISSN 0018–9219
- [LH93] LI, W. ; HENRY, S.: Maintenance metrics for the object oriented paradigm, 1993, S. 52–60
- [LLL08] LINCKE, Rüdiger ; LUNDBERG, Jonas ; LÖWE, Welf: Comparing software metrics tools. In: *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–050–0, S. 131–142
- [McC76] MCCABE, Thomas J.: A complexity measure. In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1976, S. 407
- [MR07] MEULEN, Meine J. P. van d. ; REVILLA, Miguel A.: Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs. In: *ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–3024–9, S. 203–208
- [Mun81] MUNSON, J.B.: Special Feature: Software Maintainability: A Practical Concern for Life-Cycle Costs. In: *Computer* 14 (1981), nov., Nr. 11, S. 103 –109. – ISSN 0018–9162
- [NAS01] NASA: Principal Components of Orthogonal Object-Oriented Metrics. (2001)
- [NAS04] NASA. *NASA Software Safety Guidebook*. March 2004
- [OH08] ORDONEZ, Mauricio J. ; HADDAD, Hisham M.: The State of Metrics in Software Industry. In: *Information Technology: New Generations, Third International Conference on* 0 (2008), S. 453–458. ISBN 978–0–7695–3099–4

- [PAT07] P. ANTONELLIS, Y. Kanellopoulos C. Makris E. Theodoridis C. T. ; TSIRAKIS., N.: A Data Mining Methodology for Evaluating Maintainability according to ISO/IEC-9126 Software Engineering-Product Quality Standard. In: *Special Session on System Quality and Maintainability, organized in conjunction with the 11th European Conference on Software Maintenance and Reengineering, CSMR 2007*, 2007
- [PO95] PEARSE, T. ; OMAN, P.: Maintainability measurements on industrial source code maintenance activities, 1995, S. 295 –303
- [Rig96] RIGUZZI, Fabrizio. *A Survey of Software Metrics*. July 1996
- [Sch97] SCHMIDT, Roy C.: Managing Delphi Surveys Using Nonparametric Statistical Techniques. In: *Decision Sciences* 28 (1997), Nr. 3, S. 763–774
- [SS05] SIMON, Daniel ; SIMON, Frank: Das wundersame Verhalten von Entwicklern beim Einsatz von Quellcode-Metriken. In: *Metrikon 2005 - Praxis der Software Messung* (2005)
- [SSM06] SIMON, Frank ; SENG, Olaf ; MOHAUPT, Thomas: *Code Quality Management*. dpunkt.verlag GmbH, 2006
- [Wey88] WEYUKER, E. J.: Evaluating Software Complexity Measures. In: *IEEE Trans. Softw. Eng.* 14 (1988), Nr. 9, S. 1357–1365. – ISSN 0098–5589

Internet References

- [1] *Multivariate Correlation Matrix (v1.0.4) in Free Statistics Software (v1.1.23-r6)*, 2008. http://www.wessa.net/rwasp_pairs.wasp/.
- [2] *NASA Repository*, June 2008. <http://mdp.ivv.nasa.gov/repository.html>.
- [3] *The Analytical Hierarchy Process (AHP)*, 2009. <http://www.boku.ac.at/mi/ahp/ahp.pdf>.
- [4] *Frequentis AG*, 2009. www.frequentis.com.
- [5] *Koders.com*, 2009. <http://www.koders.com/>.
- [6] *Online Judge*, November 2009. <http://uva.onlinejudge.org/>.
- [7] *Resource Standard Metrics (RSM) Tool*, 2009. <http://msquaredtechnologies.com/>.
- [8] *codebetter.com*, 2010. <http://codebetter.com/>.
- [9] *Kendall's W Calculation*, 2010. http://amchang.net/StatTools/KendallW_Pgm.php.
- [10] *Software Development Laws*, 2010. <http://www.codeproject.com/KB/work/murphy.aspx>.
- [11] *SQA.net*, 2010. <http://www.sqa.net/iso9126.html>.
- [12] *WEKA*, 2010. <http://www.cs.waikato.ac.nz/ml/weka/>.

A Appendices

In the last chapter of the thesis, further information concerning the experiments are listed. Figure A.1 depicts the template that was used for the Analytical Hierarchy Process (AHP) concerning the overall maintainability aspect and the OO model. The same is shown for the procedural model in Figure A.2. Figure A.3 and Figure A.4 give an example for the questionnaire that was used for the validation of the proposed procedural model.

Maintainability

A/B	Analyzability	Changeability	Stability	Testability
Analyzability	1			
Changeability		1		
Stability			1	
Testability				1

Analyzability: How easy it is to identify the parts that have to be modified

A/B	CBO	DIT	NOC	NOMV
CBO	1			
DIT		1		
NOC			1	
NOMV				1

Changeability: How easy it is to adapt the system

A/B	CBO	DIT	NOC	NOMV
CBO	1			
DIT		1		
NOC			1	
NOMV				1

Stability: Possibility to keep the system in a consistent state during modification

A/B	CBO	DIT	NOC	NOMV
CBO	1			
DIT		1		
NOC			1	
NOMV				1

Testability: How easy the system can be tested during/after modification

A/B	CBO	DIT	NOC	NOMV
CBO	1			
DIT		1		
NOC			1	
NOMV				1

Explanation: assign weights on a scale from 1 to 9
 1 means equally important
 9 means property A is much more important than property B
 -9 means that property B is much more important than property A



 same properties are equally important
 not to fill, is the reciprocal value

Figure A.1: Template for the AHP Process, Overall Maintainability and OO Model

Analyzability: How easy it is to identify the parts that have to be modified

A/B	LOC	NOI	NOR	ND
LOC	1			
NOI		1		
NOR			1	
ND				1

Changeability: How easy it is to adapt the system

A/B	LOC	NOI	NOR	ND
LOC	1			
NOI		1		
NOR			1	
ND				1



Stability: Possibility to keep the system in a consistent state during modification

A/B	LOC	NOI	NOR	ND
LOC	1			
NOI		1		
NOR			1	
ND				1

Testability: How easy the system can be tested during/after modification

A/B	LOC	NOI	NOR	ND
LOC	1			
NOI		1		
NOR			1	
ND				1

Explanation: assign weights on a scale from 1 to 9
 1 means equally important
 9 means property A is much more important than property B
 -9 means that property B is much more important than property A

 same properties are equally important
 not to fill, is the reciprocal value

LOC Lines of Code
 NOI Number of Input Parameters
 NOR Number of Return Points
 ND Nesting Depth

Figure A.2: Template for the AHP Process, Procedural Model

tsearch

Please try to give answers to the best of your knowledge. No name will be published.
 The questionnaire is used to determine the important properties concerning maintainability of C Code.
 The grading system is from 1 (excellent) to 5 (poor).
 Thank you for your participation.

Questions / Explanations	Grade (1...5)												
<p>Q1: How would you rate the analyzability of the software module?</p> <p>E1: <i>Analyzability</i> describes how easy or difficult it is to diagnose the system or deficiencies or to identify the relevant parts that have to be modified to achieve a new desired functionality. Think also about finding bugs/failures</p>													
<p>Q2: How would you rate the changeability of the software module?</p> <p>E2: <i>Changeability</i> describes how easy or difficult it is to adapt the system (the real implementation)</p>													
<p>Q3: How would you rate the stability of the software module?</p> <p>E3: <i>Stability</i> describes how easy or difficult it is to keep the system in a consistent state during/after modification. Would you "touch the code"?</p>													
<p>Q4: How would you rate the testability of the software module?</p> <p>E4: <i>Testability</i> describes how easy or difficult it is to test the system during/after modification</p>													
<p>Q5: How would you rate the overall maintainability of the software module?</p> <p>E5: <i>Maintainability</i> is the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. This is the generic term and consists of the former attributes:</p> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <thead> <tr> <th colspan="4" style="text-align: center;">Maintainability</th> </tr> <tr> <th style="text-align: center;">W₁</th> <th style="text-align: center;">W₂</th> <th style="text-align: center;">W₃</th> <th style="text-align: center;">W₄</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">Analyzability</td> <td style="text-align: center;">Changeability</td> <td style="text-align: center;">Stability</td> <td style="text-align: center;">Testability</td> </tr> </tbody> </table>	Maintainability				W ₁	W ₂	W ₃	W ₄	Analyzability	Changeability	Stability	Testability	
Maintainability													
W ₁	W ₂	W ₃	W ₄										
Analyzability	Changeability	Stability	Testability										
<p>If you want, give comments to your grades (why did you rate a property very good/very poor?)</p> <div style="background-color: #f2f2f2; height: 40px; width: 100%;"></div>													

Figure A.3: Questionnaire for the Validation Experiment, Part 1

tsearch

Now consider the **functional level**:

Q6: Rank the functions given below in the module **tsearch** according to their maintainability, give numbers from 1 to 4 (1 for the best maintainability, 2 for the next best, etc...). No two functions can be ranked as equal. Consider the properties from above on the functional level: Analyzability, Changeability, Stability, Testability

	Rank (1 to 4)
<code>static void maybe_split_for_insert (node *rootp, node *parentp, node *gparentp, int p_r, int gp_r, int mode)</code>	<input type="text"/>
<code>void *_tsearch (const void *key, void **vrootp, _compar_fn_t compar)</code>	<input type="text"/>
<code>void *_tdelete (const void *key, void **vrootp, _compar_fn_t compar)</code>	<input type="text"/>
<code>void *_tfind (key, vrootp, compar)</code>	<input type="text"/>

Q7: Consider the function on the first position. Why did you rank the function there? Which properties were crucial?

Q8: Consider the function on the last position. Why did you rank the function there? Which properties were crucial?

Q9: Compare the functions on the first and on the last positions, are there significant differences for you concerning maintainability? Describe them

Q... Question
E... Explanation

Time Spent in hours:

Figure A.4: Questionnaire for the Validation Experiment, Part 2