Master's Thesis

Persuasive Technologies for Improved Software Quality

Sophie Steinparz

Graz, 2012

Institute for Software Technology
Graz University of Technology

Supervisor: Alexander Felfernig

Abstract

The lack of quality in software has been a great challenge since the origin of software development. This challenge is visible in the extensive amount of research about software quality attributes and software quality metrics. Despite this comprehensive knowledge, a lack of software quality is still one of the main problem areas in the field of software development. Software developers write error-prone code against their better judgement and don't pay much attention to its quality unless they are forced to by managers or users. The goal of this work is to support developers in identifying and improving weak software artifacts at the beginning of programming such that error-prone code will be avoided and the software quality increased. Persuasive elements have been integrated into the Eclipse Framework to achieve this goal. On the basis of these elements, the user knows whether further improvements need to be made for every source code element at all times.

Acknowledgement

I would like to thank everyone who supported me during the process of writing this thesis.

Sophie Steinparz
Graz, 2012

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, ______________________ ______________________

Place, Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ______________________ ______________________

Ort, Datum Unterschrift

Contents

List of Figures **xi**

List of Tables **xiii**

1. Introduction **1**

1.1. Motivation 1

1.2. Goals 1

1.3. Related Work 2

2. Software Quality Metrics **7**

2.1. Basic Quality Attributes 7

2.2. From Basic Quality Attributes To Metrics 10

2.3. Evaluation of Metrics 17

2.4. Automatic Metrics Calculation 19

2.5. Metrics Suites, Correlations and Conflicts 21

2.6. The Downside of Metrics 23

2.7. Different Interpretations of Metrics 23

2.8. Additional Literature 26

3. Persuasive Technology **29**

3.1. Persuasion and Human Behavior Change 29

3.2. Models, Frameworks and other Concepts 32

3.3. Persuasive Technology Today 36

3.4. Ethics of Persuasive Technology 39

4. Combining The Aspects **41**

4.1. Basic Idea 41

4.2. Acceptance Pre-Study 41

4.3. Design Study 43

4.4. Tool Description “Persuasive Quality Improver” 44

4.5. Chosen Persuasive Concepts 46

4.6. Implemented Metrics 49

4.7. “Persuasive Quality Improver” Implementation 50

4.7.1. Eclipse Plug-in Basics 51

4.7.2. Decorator 54

4.7.3. Markers 54

4.7.4. Metrics 55

4.7.5. Dialog 55

4.7.6. Database and Logging 56

4.8. Remarks for Functionality Extension 56

4.8.1. Add more Metrics 56

4.8.2. Suggest refactoring-tips to the user at the right time 57

4.8.3. Improved Self-Monitoring 57

4.8.4. Introduce a social factor 58

4.8.5. Praise and Reward 62

4.8.6. Increase the System Credibility 63

5. Case Study **65**

5.1. Participants 65

5.2. Chosen Assignment 65

5.3. Automated Evaluation 67

5.4. Code Review 69

5.5. Feedback 72

6. Future Work **75**

6.1. Implementing the Feedback 75

6.2. Technical Improvements 76
6.3. Offer More Information 76
6.4. More Detailed Improvement Suggestions 76

7. Conclusion **77**

Bibliography **79**

List of Figures

1.1. Hierarchical visualization of the program structure with Code Smells as leaves by Van Emden and Moonen [58] 3
1.2. CodeNose shows flaws in the source code to the user. [53] 4
1.3. KenyaEclipse shows the user how to simplify the code. [10] 5

2.1. Hierarchical structure of software characteristics [7] 9
2.2. Definition and evaluation of Boehm et al.'s metrics [7] in respect of potential benefit and correlation with quality factors. 12
2.3. Example of a McCabe metric calculation. This method has a McCabe metric value of 4. 13
2.4. Mapping between metrics and Booch's object oriented design elements [8] from Chidamber and Kemerer [12] 14
2.5. Coupling Count metrics for reusability measures: from Price and Demurjian [47] . . 16
2.6. Meaning of the Reusability metrics: from Price and Demurjian [47] 16
2.7. Extended ISO software quality model by Leung [34] 17
2.8. Correlation matrix for the Chidamber and Kemerer [13] metrics suite [5] 18
2.9. Evaluation of completeness and possibility of metrics automation from Boehm et al. [7] 20
2.10. The nine issues of the Issue-oriented approach for software quality measurement [51] 22
2.11. Table of metrics to evaluate existing components [52] 25

3.1. The "Behavior Grid" describes different behavior change categories. [28] † 31
3.2. The social feedback agent iCat from Midden and Ham [40] gives feedback through facial gesture 33

3.3. “Simple Joy” persuasive tablecloth to encourage stress-free eating [57] 35
3.4. The overlapping of Persuasion and Computers shows the area for “Captology” [22] . 36
3.5. Orino and Kitamura [45]’s process to persuade the user to buy a product different to the user’s first choice. 37
3.6. Examples of the use of ViTo, the persuasive television remote control [42] 38
3.7. Persuasive designers have to take responsibility for the outcome of their design under specific circumstances. This flow-chart diagram helps to decide when the designer has to be responsible. [6] 39

4.1. Quality feedback icons variant 1 43
4.2. Quality feedback icons variant 2 43
4.3. Quality feedback icons variant 3 43
4.4. Eclipse IDE with the developed plug-in 44
4.5. Eclipse IDE without the developed plug-in 45
4.6. Eclipse with the developed plug-in and the activated quality guide (Issuenary). 46
4.7. Suggestion: The Eclipse marker shows the user which code has to be improved. . . . 47
4.8. Suggestion: The Eclipse marker shows the user how to improve the code (textual recommendation) 47
4.9. The plug-in’s package explorer close-up with very good software quality in the right window (nearly all icons green [bright]) and worse software quality in the left window (many red icons [dark]) 48
4.10. The Eclipse Marker View close-up including some improvement suggestions for the user. 49
4.11. Correlation matrix of Chidamber and Kemerer [13]’s metrics suite by Succi et al. [55] 49
4.12. Definition of the implemented metrics 50
4.13. Part 1: Visualization of further ideas 59
4.14. Part 2: Visualization of further ideas 60

List of Tables

5.1. Lines of code per method - evaluated results 68
5.2. Depth of Inheritance Tree - evaluated results 68
5.3. Lack of Cohesion in Methods - evaluated results 68
5.4. McCabe - evaluated results 68
5.5. Code Review Questionnaire 70
5.6. χ^2 test results of questionnaire 71

Chapter 1

Introduction

1.1. Motivation

Software quality factors such as maintainability and reusability are highly important in contemporary software development. A program does not only have to fulfill predefined functionality but also has to be robust and predictable, even if it is used wrongly. Everybody who has learned how to develop software should have heard of quality factors. However, this does not mean that everybody uses this knowledge when implementing software. The question of why this happens was raised during the work on this thesis. There are many advantages for the programmer when writing code that is easy to maintain and reuse. Code refactoring is a good tool for keeping code maintainable and reusable during a complete project life cycle. The complexity of code refactoring directly relates to the complexity of the source code. This means code refactoring should be made before the source code becomes too complex.

1.2. Goals

The goal of this thesis is to help achieve a higher software quality and motivate the programmer to increase the quality of their programs from the beginning, in order to avoid long phases of refactoring and frustrated programmers, users, and customers. To achieve this goal three main major tasks where defined:

- **Major Task 1**: Conduct research on software quality measures. The question “How can quality be measured?” quickly leads to software quality metrics. This thesis gives an insight into the topic of metrics and will start by summarizing what research has already been conducted, what is important about metrics, and how to use them reasonably.

- **Main Task 2**: Conduct research on persuasive technology. This is needed to overcome the programmer's lack of motivation, to provide ideas on how to visualize certain information, and design persuasive user interfaces.
- **Main Task 3**: Combine the first two main tasks to create an integrated concept. This concept is realized as an Eclipse Plug-in prototype called "Persuasive Quality Improver" and evaluated from various perspectives. As a result of the evaluation of this prototype, ideas for improvement and issues for future work are discussed.

The tasks presented are reflected in the structure of this thesis. Chapter 2 gives insights into software quality attributes, how to measure them, and important precautions for their correct use. Persuasive Technologies are introduced in Chapter 3. Chapter 4 presents pre-studies and their results, the chosen persuasive concepts, chosen metrics, and the tool implemented within the scope of this thesis. In addition, ideas for further functionality, including a rough description on how to realize them, are provided. Chapter 5 presents the case study conducted and its results. Finally, Chapter 6 presents issues for future work.

The major contributions of this thesis are the following:

- Use Persuasive Technology in the field of software quality evaluation as a new area of application
- Development of an Eclipse plug-in to investigate whether persuasive approaches are successful
- Empirical studies that clearly show the applicability of the developed approaches

1.3. Related Work

This work is not the first one that has tried to improve software quality during its implementation. Van Emden and Moonen [58] worked on a tool that was meant to detect *"problems that are generally associated with bad program design and bad programming practices"*. Emden and Moonen took a closer look at so-called "code smells" and coding standards. According to Emden and Moonen, "code smells" are patterns in code which indicate necessary refactoring. For example, code smells could be duplicate code or methods that are too long. The two main points in their work were how to detect and present these code smells. Because in this context code smells only define static structures, they can be detected by parsing the static code. The parser will create a parse tree which then will be analyzed according to structure and relations between the program entities. Code smells will be detected during this process. These code smells were visualized as a tree in their prototype (see Figure 1.1). This tree represents "*all the packages, classes, interfaces, methods and constructors, and their attached smell nodes*"[58]. All leaves in Figure 1.1 represent code smells.

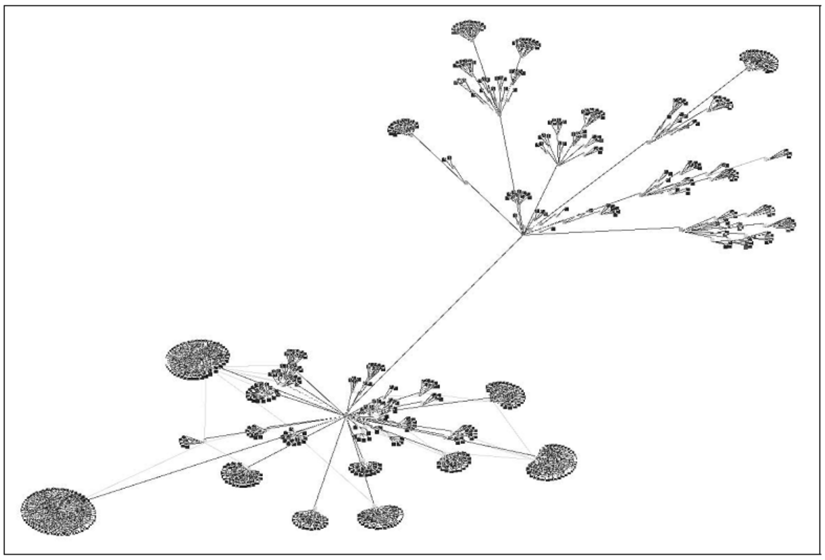


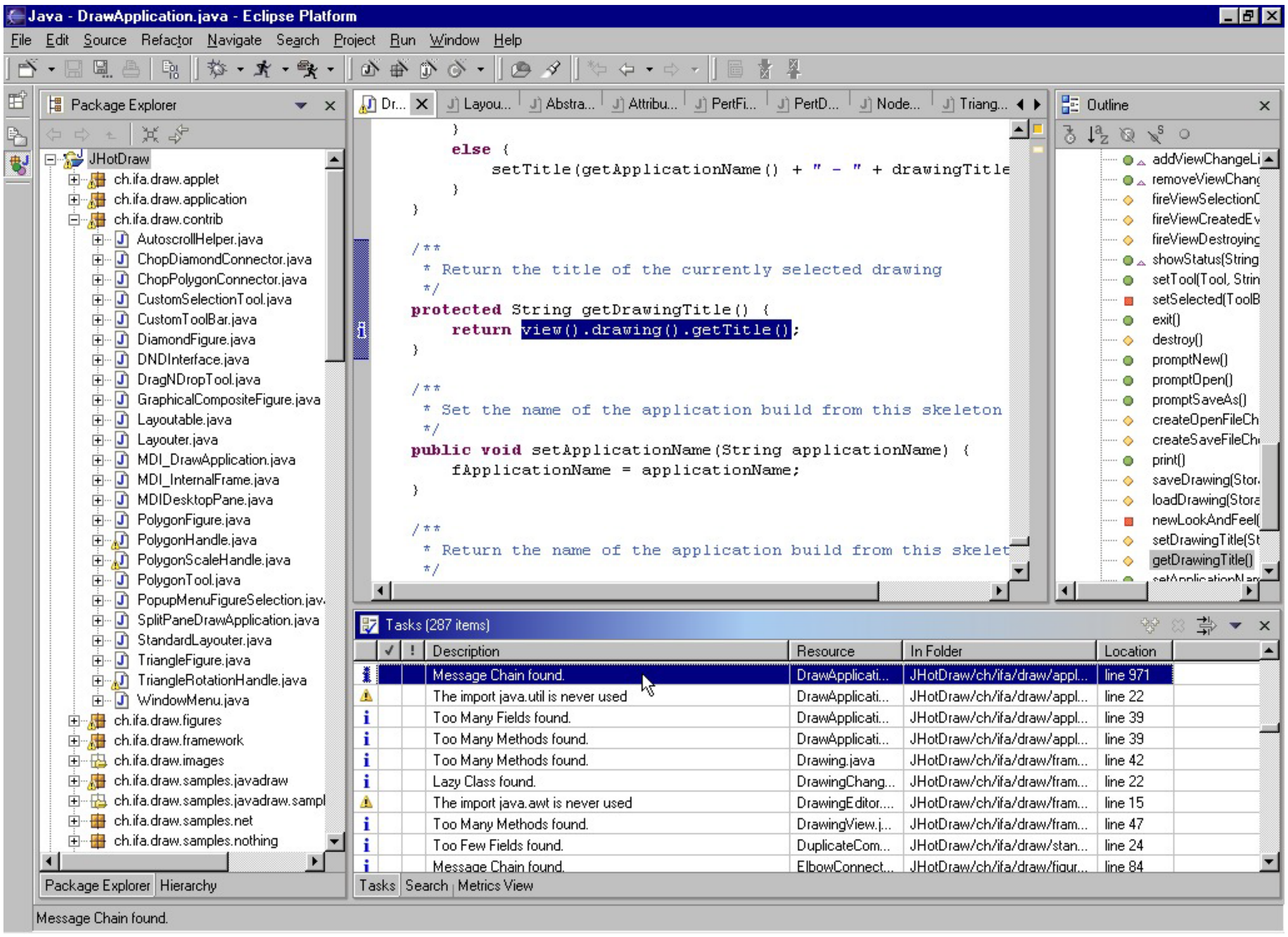
Figure 1.1.: Hierarchical visualization of the program structure with Code Smells as leaves by Van Emden and Moonen [58]

Another work on code improvement is done by Slinger [53]. Slinger's prototype works in the Eclipse Environment, similar to the prototype presented in this thesis. Again, the concept of code smells is used. Slinger has defined different code smells (e.g., switch statements, long methods, empty catch clauses) which are detected with the so-called "CodeNose" Eclipse plug-in. A nice feature of this plug-in is that the user can choose between the code smells he or she wants to detect.

A third work which has a lot in common with this thesis is the so-called KenyaEclipse by Chatley and Timbul [10][*]. This is an Eclipse Environment designed to help users to learn programming. The environment checks code style with the intention of teaching programming students how to write structured and easy-to-read code. Each structure violation detected will be shown to the user (see Figure 1.3) and the tool provides quick-fix changes the user can apply to solve the problem. In Figure 1.3 KenyaEclipse shows how to simplify the combination of boolean expressions and boolean return values to the user. With the help of the tool the user can reduce four lines of code to one. The resulting source code is: "$return(n < 0);$".

All these approaches are designed to achieve higher software quality, based on different concepts. None of them uses persuasive technology nor software quality metric suites, which are the main components in this thesis. How to integrate persuasive technologies in combination with metric suites to software development environments such as Eclipse will be discussed in detail in the following sections.

[*]The Kenya tool has its own website: http://chatley.com/kenya

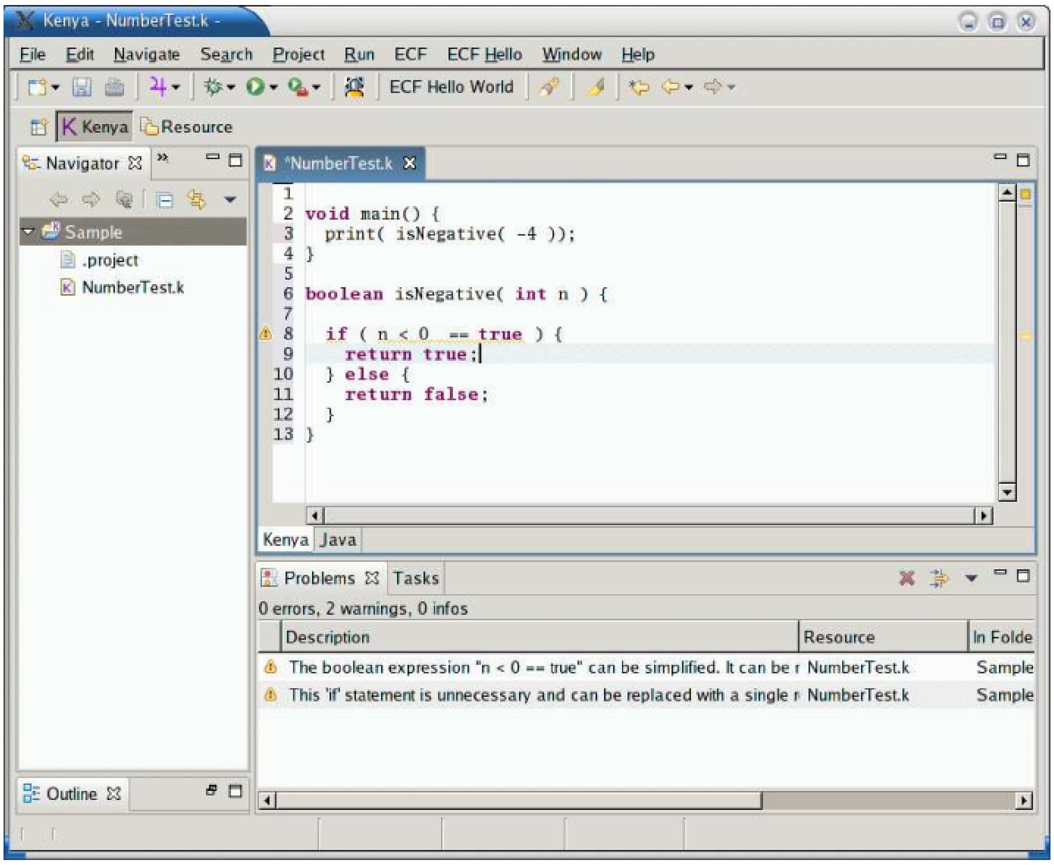


```
        }
        else {
            setTitle(getApplicationName() + " - " + drawingTitle
        }
    }

    /**
     * Return the title of the currently selected drawing
     */
    protected String getDrawingTitle() {
        return view().drawing().getTitle();
    }

    /**
     * Set the name of the application build from this skeleton
     */
    public void setApplicationName(String applicationName) {
        fApplicationName = applicationName;
    }

    /**
     * Return the name of the application build from this skelet
     */
```

Tasks (287 items)

	✓	!	Description	Resource	In Folder	Location
i			Message Chain found.	DrawApplicati...	JHotDraw/ch/ifa/draw/appl...	line 971
⚠			The import java.util is never used	DrawApplicati...	JHotDraw/ch/ifa/draw/appl...	line 22
i			Too Many Fields found.	DrawApplicati...	JHotDraw/ch/ifa/draw/appl...	line 39
i			Too Many Methods found.	DrawApplicati...	JHotDraw/ch/ifa/draw/appl...	line 39
i			Too Many Methods found.	Drawing.java	JHotDraw/ch/ifa/draw/fram...	line 42
i			Lazy Class found.	DrawingChang...	JHotDraw/ch/ifa/draw/fram...	line 22
⚠			The import java.awt is never used	DrawingEditor....	JHotDraw/ch/ifa/draw/fram...	line 15
i			Too Many Methods found.	DrawingView.j...	JHotDraw/ch/ifa/draw/fram...	line 47
i			Too Few Fields found.	DuplicateCom...	JHotDraw/ch/ifa/draw/stan...	line 24
i			Message Chain found.	ElbowConnect...	JHotDraw/ch/ifa/draw/figur...	line 84

Figure 1.2.: CodeNose shows flaws in the source code to the user. [53]



```

void main() {
  print( isNegative( -4 ));
}

boolean isNegative( int n ) {

  if ( n < 0  == true ) {
    return true;
  } else {
    return false;
  }
}
```

0 errors, 2 warnings, 0 infos

Description	Resource	In Folde
The boolean expression "n < 0 == true" can be simplified. It can be r	NumberTest.k	Sample
This 'if' statement is unnecessary and can be replaced with a single r	NumberTest.k	Sample

Figure 1.3.: KenyaEclipse shows the user how to simplify the code. [10]

Chapter 2

Software Quality Metrics

"You cannot control what you cannot measure"[16]. This chapter introduces how to make source code measurable. Software Quality Metrics (SQM) are a key element of this thesis. *"The term "metric" is defined as a measure of the extent or degree to which a product (here we are concentrating on code) possesses and exhibits a certain (quality) characteristic"* (Boehm et al. [7]). The words metric and measure are used synonymously here. SQM have been an important topic since at least 1968 [49] . Since then, many different SQM have been developed and evaluated. This chapter gives an overview of SQM to give the reader an impression of the evolution and diversity of metrics.

2.1. Basic Quality Attributes

The first step when it comes to calculating metrics is to define quality characteristics (also named quality attributes). There are many different quality attributes. One set of quality attributes can be taken from Rubey and Hartwick [49]. The first part of their work defines seven quality attributes:

- $\mathbf{A_1}$: Mathematical calculations are correctly performed
- $\mathbf{A_2}$: The program is logically correct
- $\mathbf{A_3}$: There is no interference between program entities
- $\mathbf{A_4}$: Computation time and memory usage are optimized
- $\mathbf{A_5}$: The program is intelligible
- $\mathbf{A_6}$: The program is easy to modify
- $\mathbf{A_7}$: The program is easy to learn and use

This attribute group reflects very basic but specific problems of source code quality. Another more complex attribute set is presented by Boehm et al. [7]. Boehm et al. first tried to find one overall metric which can be used to determine the quality of a program. They concluded that there probably is no such single metric. They did identify software quality characteristics. An extract of their initial set of software characteristics, including a brief description, is given here [7]:

- **Understandability**: Code possesses the characteristic understandability to the extent that its purpose is clear to the inspector
- **Completeness**: All the code parts are present and fully developed (including external references)
- **Portability**: Code can be built easily and well on computer configurations other than its current one
- **Consistency:** Code contains uniform notation, terminology and symbology within itself. For **external consistency**, the content has to be traceable to the requirements
- **Maintainability**: It must be possible to update the code to satisfy new requirements or to correct deficiencies
- **Testability**: Code facilitates the establishment of verification criteria and supports evaluation of its performance
- **Usability**: Code is reliable, efficient, and robust against human errors
- **Reliability**: Code can be expected to perform its intended functions satisfactorily
- **Structuredness**: The code possesses a definite pattern of organization of its interdependent parts
- **Efficiency**: Code fulfills its purpose with minimal time and memory usage

After a closer look at these characteristics, relations between them were found and led to a hierarchical structure (see Figure 2.1) . Many of the quality characteristics listed above can be better achieved by lowering the software complexity. A software is more easy to test and to maintain when its code is easier to understand. This is why complexity is added to the list of quality attributes even though it is not directly seen by the end-users of a software.

These characteristics all have in common that they are very unspecific and cannot be measured objectively. Nevertheless, these are the terms a user or customer may use to define quality of a product. It is not possible to quantify these quality attributes with the given characteristics and therefore the level of quality cannot be defined or approved.

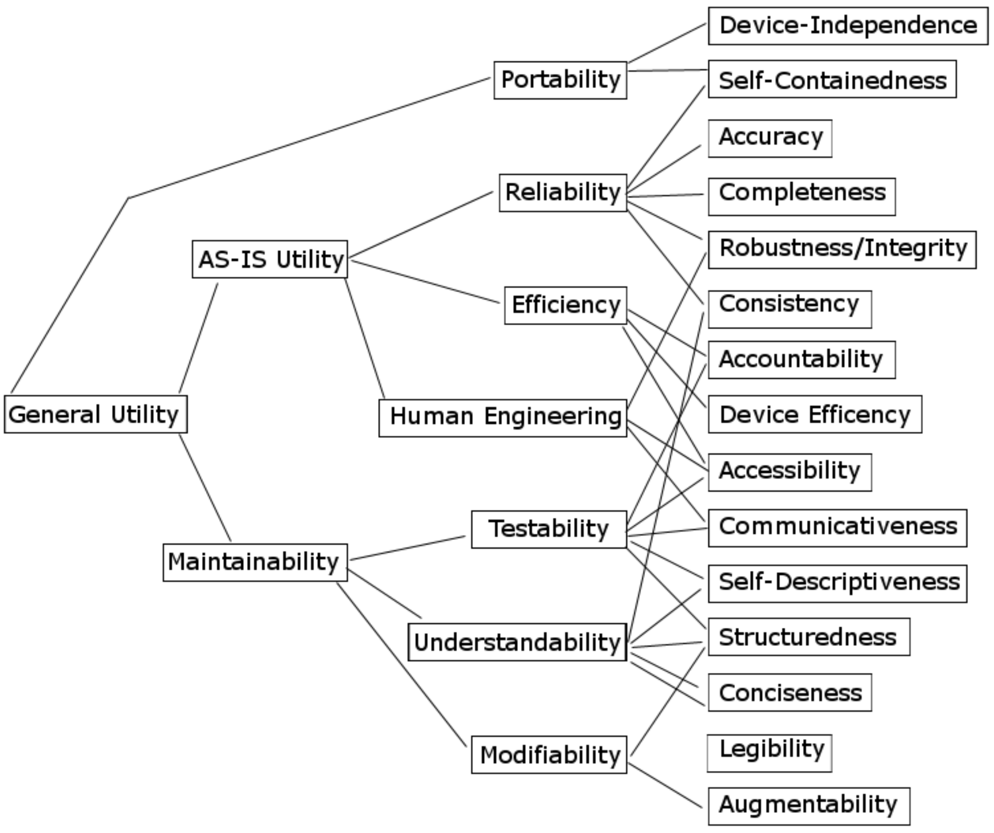


Figure 2.1.: Hierarchical structure of software characteristics [7]

2.2. From Basic Quality Attributes To Metrics

"In the absence of specific, applicable quantitative measurement tools there exists no means of defining the desired level of quality" [49]. This means defining quality attributes is only the first step in measuring and defining software quality. The next step has to be making these quality attributes measurable. Rubey and Hartwick [49] took this step in the second part of their work. They defined specific metrics to quantify quality attributes for each of the quality groups presented in Section 2.1. The following list exemplifies the quality attribute group "A_2 - The program is logically correct" with six out of twelve metrics[49]:

- $\mathbf{A_{2_1}}$: There are no open branches
- $\mathbf{A_{2_2}}$: Branches point to the correct place in the program
- $\mathbf{A_{2_3}}$: Branches do not initiate an unending loop
- $\mathbf{A_{2_4}}$: Equality comparisons between floating-point operands
- $\mathbf{A_{2_5}}$: Limit checks are provided on index tables
- $\mathbf{A_{2_6}}$: Program entities are capable of performing their required functions in less than the maximum and more than the minimum time allowed

It is important for a metric that its evaluation is possible on an objective basis. Given a metric and a specific source code, every evaluator has to come to the same conclusion. For each quality attribute group Rubey and Hartwick defined one metric as mathematical formula too. In group A_2 the mathematical formula for A_{2_6} is the following:

$$A_{2_6} = 100 \left[\sum_{i=1}^{n} (A_i + B_i) \right] /2n \tag{2.1}$$

$$A_i = \begin{cases} 1 & \text{if the maximum execution time for the i-th program entity} \\ & \text{ is less than its allowable maximum} \\ 0 & \text{otherwise} \end{cases} \quad B_i = \begin{cases} 1 & \text{if the maximum execution time for the i-th program entity} \\ & \text{ is greater than its allowable minimum} \\ 0 & \text{otherwise} \end{cases}$$

$$n = \text{number of program entities}$$

Regarding overall ratings, it is important that all metrics have a value range from zero to 100. This can be seen in the formula stated above.

The third and last part of Rubey and Hartwick's work [49] is about external factors. External factors are important because they can influence the possible result range of metrics. External factors can be: security specifications, programming specifications, computer hardware, and so forth. This means the programmers cannot be blamed for bad metrics results which are based on external factors.

The defined metrics and the external factors lead to the Q quality model proposed by Rubey and

Hartwick [49]. Q combines the absolute metric values M (e.g. $M_{Runtime} = 4s$), the normalized metric values M' (e.g. , $M'_{Runtime} = 33.3$) which take external factors into account, and the weighted metric values M" which also consider relative importance. The formulae given below show how M' and M" are calculated [49]:

$$M'_i = 100\left(\frac{M_i - M_{i_{min}}}{M_{i_{max}} - M_{i_{min}}}\right) \tag{2.2}$$

$$0 <= M_{i_{min}} <= M_i <= M_{i_{max}} <= 100 \tag{2.3}$$

where $M_{i_{min}}$ and $M_{i_{max}}$ define the practical range which could be obtained for a given set of external factors.

$$M''_i = \frac{k_i M'_i}{100} \tag{2.4}$$

where k_i are attribute weights assigned by the user, ranging in value from 0 (no importance) to 100 (maximum importance)

For example: The following numbers have been determined:
$M_{Runtime} = 4s$, $M_{Runtime_{min}} = 2s$, $M_{Runtime_{max}} = 8s$

$$M'_{Runtime} = 100\left(\frac{M_{Runtime} - M_{Runtime_{min}}}{M_{Runtime_{max}} - M_{Runtime_{min}}}\right) = 100\left(\frac{4s-2s}{8s-2s}\right) = 33.3 \tag{2.5}$$

The metric $M_{Runtime}$ has been weighted with $k_{Runtime} = 70$

$$M''_i = \frac{k_{Runtime} M'_{Runtime}}{100} = \frac{70 * 33}{100} = 23.31 \tag{2.6}$$

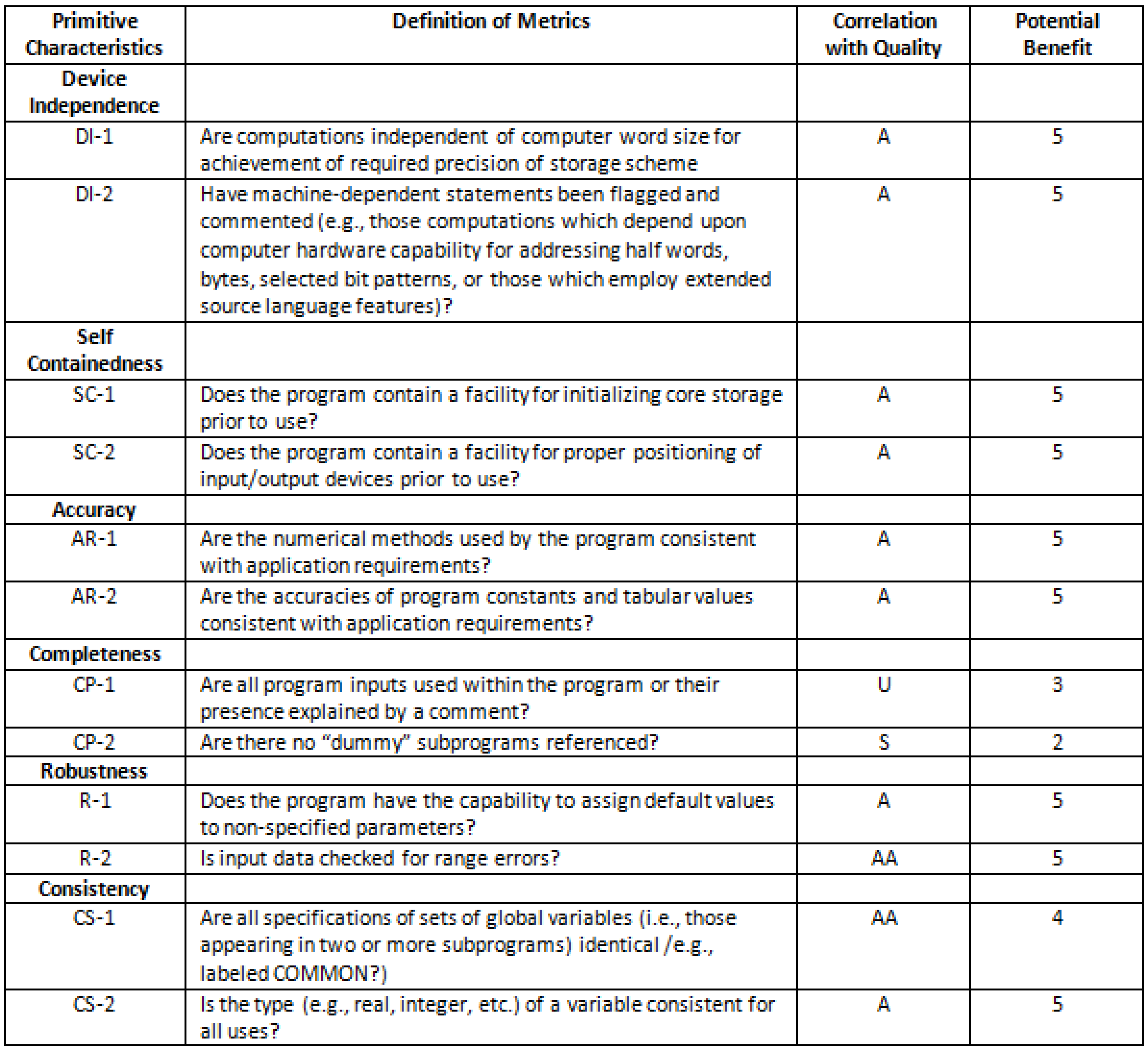
This means for the metric $M_{Runtime}$ there is a potential of 23.31% for improvement.

Another way to get from characteristics to metrics is shown by Boehm et al. [7]. After defining quality characteristics they formulated many metrics for each characteristic and then analyzed whether these metrics were really able to reflect its characteristics. This analysis and evaluation considers the potential quality benefit of an improved metric on the one hand, and the correlation with quality attributes on the other hand. The potential benefit was valued from 5 (*"Extremely important for metric to have a high score; major potential troubles otherwise"*) to 1 (*"slight incremental value for metric to have high score; no real loss otherwise"*). The correlation with quality has been valued as follows [7]:

- A - Very high positive correlation; nearly all programs with a high metric score will possess the associated characteristic
- AA - High positive correlation; a good majority (say 75-90%) of all programs with a high metric score will possess the associated characteristic

- U - Usually (say 50 - 70%) of all programs with a high metric score will possess the associated characteristic
- S - Some programs with high metric scores will possess the associated characteristic

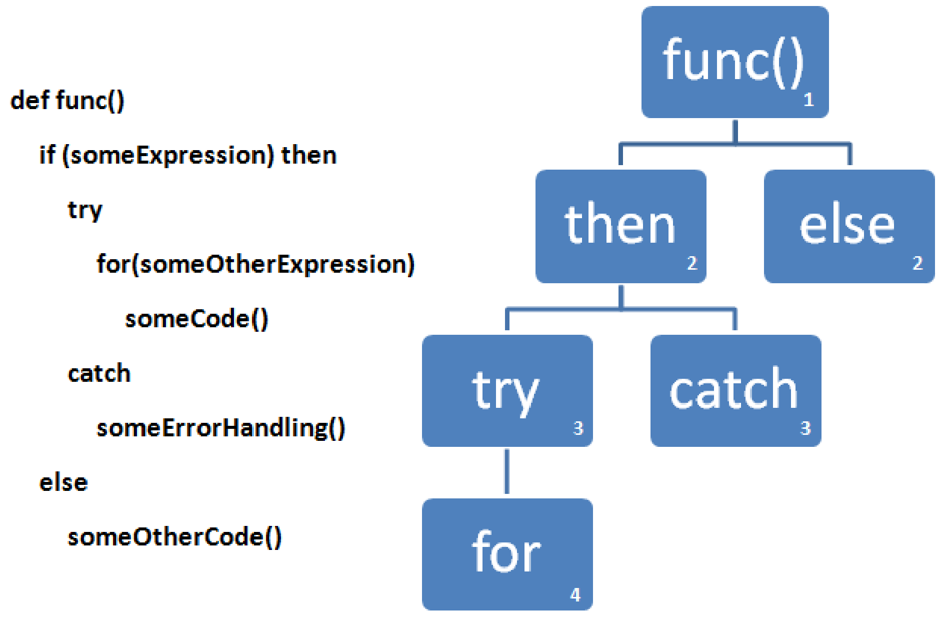
In Figure 2.2 some examples of metrics are shown. Many of them have a quality correlation of at least AA and a potential benefit of 5 and are, therefore, highly promising.



Primitive Characteristics	Definition of Metrics	Correlation with Quality	Potential Benefit
Device Independence			
DI-1	Are computations independent of computer word size for achievement of required precision of storage scheme	A	5
DI-2	Have machine-dependent statements been flagged and commented (e.g., those computations which depend upon computer hardware capability for addressing half words, bytes, selected bit patterns, or those which employ extended source language features)?	A	5
Self Containedness			
SC-1	Does the program contain a facility for initializing core storage prior to use?	A	5
SC-2	Does the program contain a facility for proper positioning of input/output devices prior to use?	A	5
Accuracy			
AR-1	Are the numerical methods used by the program consistent with application requirements?	A	5
AR-2	Are the accuracies of program constants and tabular values consistent with application requirements?	A	5
Completeness			
CP-1	Are all program inputs used within the program or their presence explained by a comment?	U	3
CP-2	Are there no “dummy” subprograms referenced?	S	2
Robustness			
R-1	Does the program have the capability to assign default values to non-specified parameters?	A	5
R-2	Is input data checked for range errors?	AA	5
Consistency			
CS-1	Are all specifications of sets of global variables (i.e., those appearing in two or more subprograms) identical /e.g., labeled COMMON?)	AA	4
CS-2	Is the type (e.g., real, integer, etc.) of a variable consistent for all uses?	A	5

Figure 2.2.: Definition and evaluation of Boehm et al.’s metrics [7] in respect of potential benefit and correlation with quality factors.

McCabe [38] introduces a metric which quantifies the quality attribute complexity. The focus of this metric concentrates on the question *“How to modularize a software system so the resulting modules* *are both testable and maintainable?”*. The result of McCabe’s [38] work is a popular and easy to use complexity metric known as the “McCabe Complexity Metric”. This metric is used as an indicator if a given code will be difficult to test or maintain. This cyclomatic metric calculates the complexity based on graph-theoretical methods. Figure 2.3 demonstrates how this metric values source code structure. McCabe and Butler [39] extend this paper with *“design complexity measures for the development of software systems”*. With this they want to provide a solution to the problem that *“while many studies dealt with the subject of measuring program complexity, few studies have concentrated on measuring complexity of development specifications.”*[39] With this motivation they introduced the following metrics: Module Design Complexity, Design Complexity and Integration Complexity. These metrics are not designed to evaluate source code but the previous step of specification. All these metrics are based on flow-graphs and, therefore, on cyclomatic complexity like the McCabe Complexity Metric.



```
def func()
  if (someExpression) then
    try
      for(someOtherExpression)
        someCode()
    catch
      someErrorHandling()
  else
    someOtherCode()
```

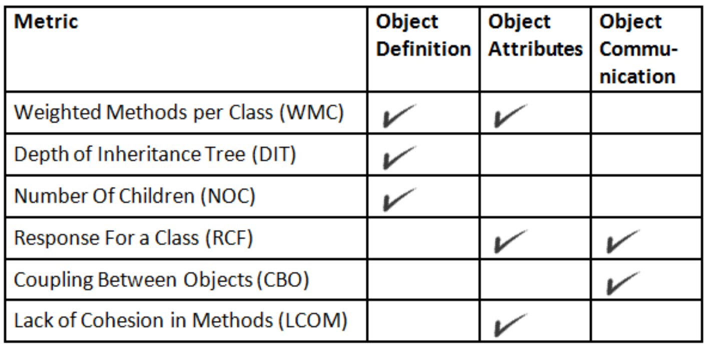
Figure 2.3.: Example of a McCabe metric calculation. This method has a McCabe metric value of 4.

Chidamber and Kemerer [12] published metrics for object oriented software design. These metrics were developed according the fundamental elements of object oriented design by Booch [8]. The fundamental elements of object oriented design are: “Object Definition”, “Attributes of Objects” and “Communication Along Objects”. All of the published metrics are complexity metrics. This work interprets complexity of an individual (e.g., a class or a method) as *numerosity of its composition*.

This means everything is as complex as the composition of its parts. Chidamber and Kemerer [12] define the following metrics according to this definition:

- **Weighted Methods Per Class (WMC)**: A class consists of its methods. Its complexity can therefore be calculated by the complexity of its methods. Each method has a weight describing its complexity.
- **Depth of Inheritance Tree (DIT)**: The inheritance tree belongs to the components of each class. The deeper the tree of inheritance is, the more complex the actual class is
- **Number of Children (NOC)**: The more classes extend the main class, the more dependencies it has. The more dependencies a class has, the more difficult it is to maintain
- **Coupling Between Objects CBO)**: Two objects are coupled to each other if they act upon each other. This means for example, one class uses methods or an instance variable of the other class. Two coupled objects, as long as they are not in each other's inheritance tree, lead to more difficult maintenance because of their interdependency
- **Response For a Class (RFC)**: *"The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class"* [13]. If this set is a large one, testing and debugging becomes more complex
- **Lack of Cohesion in Methods (LCOM)**: Cohesion leads to encapsulation of a class. This is a desirable attribute of a class

The mapping between these metrics and Booch's elements of object oriented design [8] can be seen in Table 2.4.



Metric	Object Definition	Object Attributes	Object Commu-nication
Weighted Methods per Class (WMC)	✓	✓	
Depth of Inheritance Tree (DIT)	✓		
Number Of Children (NOC)	✓		
Response For a Class (RCF)		✓	✓
Coupling Between Objects (CBO)			✓
Lack of Cohesion in Methods (LCOM)		✓	

Figure 2.4.: Mapping between metrics and Booch's object oriented design elements [8] from Chidamber and Kemerer [12]

Li and Henry [35] focused on object oriented metrics for maintainability. They based their set of metrics on the Chidamber and Kemerer [13] metrics presented above. Instead of the "Coupling Between Objects" they defined three coupling metrics of their own:

- **Coupling through inheritance**: This is a combination of the metrics "Depth of Inheritance Tree" and "Number Of Children". Both metrics address inheritance
- **Coupling through message passing**: The messages sent between two classes indicate how dependent these classes are on each other
- **Coupling through data abstraction**: A class A contains a member variable of class B. This means there is a coupling between A and B. This coupling complexity is measured by this metric

Additionally, two so-called "size metrics" are presented. Size metrics are metrics which take size attributes to measure complexity (e.g., Lines of Code). These metrics were analyzed *"to determine if the maintenance effort can be predicted from metrics"*. The conclusion of this analysis was that the prediction is possible.[35]

Price and Demurjian [47] analyzed and measured the software quality attribute of "Reusability". A main factor for reusability is coupling. *"Coupling is defined as an inter-hierarchy dependency that results when methods of one hierarchy use methods or instance variables of another hierarchy."* They found eight types of coupling depending on the level of specialization of a class and if a related hierarchy exists or not. Some types of coupling are desirable and others not. Each type of coupling corresponds to a defined coupling metric. The metrics are named Coupling Count CC_1 to CC_8 and can be seen in Figure 2.5. Figure 2.6 shows which types of coupling are desirable and which not. For a better understanding of the metrics and their meaning, CC_1 and CC_3 are explained by way of example:
$CC_1 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_j G_i$: Count all couplings between general classes which are related to each other.
$CC_3 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_j S_i$: Count all couplings from general classes to related specific classes.
According to Figure 2.6, CC_3 coupling is not desirable. There should be no coupling from a general class to a related specific class. For example, there is a general class named "Vehicle" with associated specific classes named "Car", "Motorbike", and "Bike". Besides these classes there is also a general class named "Ground" with associated specific classes named "Highway", "Grassland", and "Staircase". If "Vehicle" is coupled with "Ground" this would be a Coupling Count 1 metric, which is good for reuse. The software design is good if the vehicle generally knows how to interact with a ground. On the other hand, if the ground is coupled with a car (the class "Car" is not related to the class "Ground"; therefore this is measured by the CC4 metric) or with the grass (the class "Ground" is related to the class "Grass"; therefore this is measured by the CC3 metric) this is bad for reuse. It may be an advantage for this very special project to have this coupling, but in general this coupling is does not make any sense.

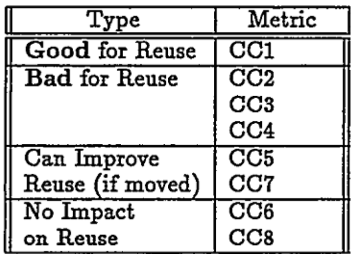
Leung [34] analyzed metrics for intranet applications. Based on an extended ISO software quality model (see Figure 2.7), he conducted a user survey study. This study identified the importance of each metric for intranet applications. Different metrics were defined for each of the top five rated

$$
\begin{array}{l|l}
CC_1 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_j G_i & CC_5 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_j G_i \\
CC_2 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_j G_i & CC_6 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_j G_i \\
CC_3 = \sum_{i=1}^{m} \sum_{j=1}^{x} G_j S_i & CC_7 = \sum_{i=1}^{m} \sum_{j=1}^{y} S_j S_i \\
CC_4 = \sum_{i=1}^{n} \sum_{j=1}^{x} G_j S_i & CC_8 = \sum_{i=1}^{n} \sum_{j=1}^{y} S_j S_i
\end{array}
$$

m	number of hierarchies which are related to this class
n	number of hierarchies which are not related to this class
x	number of General classes in this hierarchy
y	number of Specific classes in this hierarchy

$G_j G_i$	number of couplings from the j-th General class to all General classes in the i-th hierarchy
$G_j S_i$	number of couplings from the j-th General class to all Specific classes in the i-th hierarchy
$S_j G_i$	number of couplings from the j-th Specific class to all General classes in the i-th hierarchy
$S_j S_i$	number of couplings from the j-th Specific class to all Specific classes in the i-th hierarchy

Figure 2.5.: Coupling Count metrics for reusability measures: from Price and Demurjian [47]



Type	Metric
Good for Reuse	CC1
Bad for Reuse	CC2 CC3 CC4
Can Improve Reuse (if moved)	CC5 CC7
No Impact on Reuse	CC6 CC8

Figure 2.6.: Meaning of the Reusability metrics: from Price and Demurjian [47]

software characteristics ("*Availability*", "*Accuracy*", "*Security*", "*Suitability*", and "*Time behavior*"). Then these metrics were analyzed according to their importance, urgency, and costs. The following three metrics were chosen as a result of this analysis:

- **Availability AVAIL**: $\frac{\sum \text{Prime hours} - \sum \text{Outage hours}}{\sum \text{Prime hours}}$ Prime hours are those hours where the application has to be available. For an intranet application this could mean the office hours (e.g., 8am to 6pm)
- **Failure rate FR**: $\frac{\sum \text{Number Of failures}}{\sum \text{Execution hours}}$
- **Software failure rate SFR**: $\frac{\sum \text{No. of software failures}}{\sum \text{Execution hours}}$

These three metrics are strongly related to the five characteristics stated above.

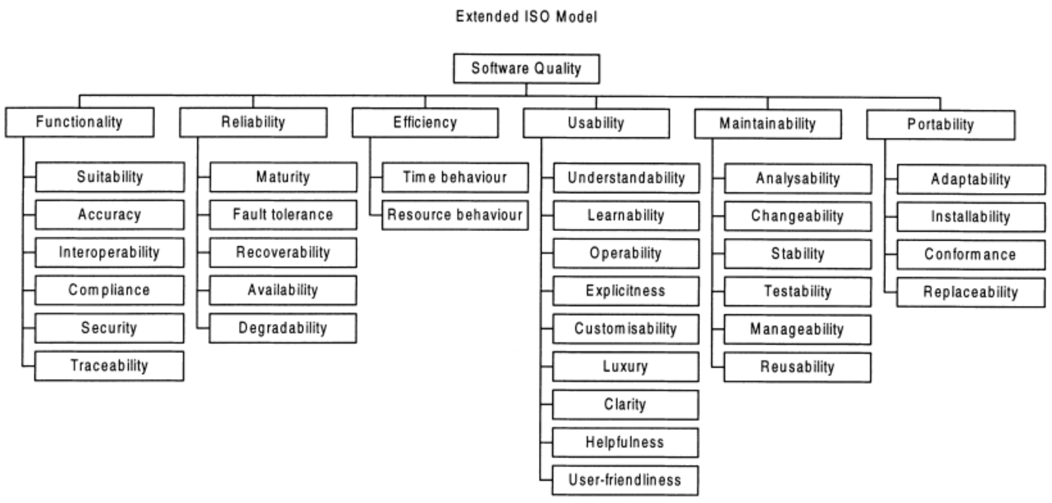


Figure 2.7.: Extended ISO software quality model by Leung [34]

2.3. Evaluation of Metrics

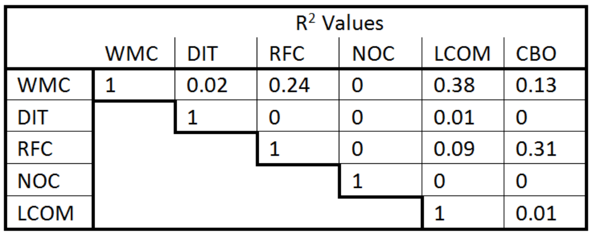
As discussed in the previous section, there are many different approaches to define metrics. The definition of a formal function for a metric does not always mean that this metric helps measure software quality attributes. Before implementing any metrics it is important to evaluate them according to their significance.

Chidamber and Kemerer [13] evaluated all of their metrics according to six properties for metrics [13].

- **Noncoarseness** : Different classes may have different results for the metric m

- **Nonuniqueness** : It must be possible that different classes can have the same results for the metric m
- **Design Details are Important**: Two classes which provide the same functionality do not have to have the same results for the metric m
- **Monotonicity**: A class which is the combination of two classes must have at least the same results for the metric *m* as each of the basic classes have alone
- **Nonequivalence of interaction**: Given the classes A, B, and C and the constraint metric $m(A) = m(B)$ this must not imply that $m(A+C) = m(B+C)$
  For example, the classes Car, Grass, and Bike exist and Lines of Code LOC(Car) = LOC(Grass) is valid. In both classes the functionality of Bike has to be implemented. It is very likely that LOC(Grass+Bike) will be higher than LOC(Car+Bike) because there will be some similar functionality in the classes Car and Bike which can be merged
- **Interaction Increases Complexity**: The complexity of a class combining the functionality of two other class can be higher than the sum of the complexity of the two basic classes

This means each metric is able to generate significant values, which makes it possible to compare different source codes to each other. Basili et al. [5] analyzed Chidamber and Kemerer's metrics [13] according to its distribution and correlation. Distribution means the value range of a metric. The distribution of a metric is important to interpret a given metric value. For example, the metric Number of Children gives for nearly all evaluated classes the value 0 and the maximum value is 4. This means that this metric has a very low distribution. The result of the correlation analysis can be seen in Figure 2.8.



	R^2 Values					
	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0.02	0.24	0	0.38	0.13
DIT		1	0	0	0.01	0
RFC			1	0	0.09	0.31
NOC				1	0	0
LCOM					1	0.01

Figure 2.8.: Correlation matrix for the Chidamber and Kemerer [13] metrics suite [5]

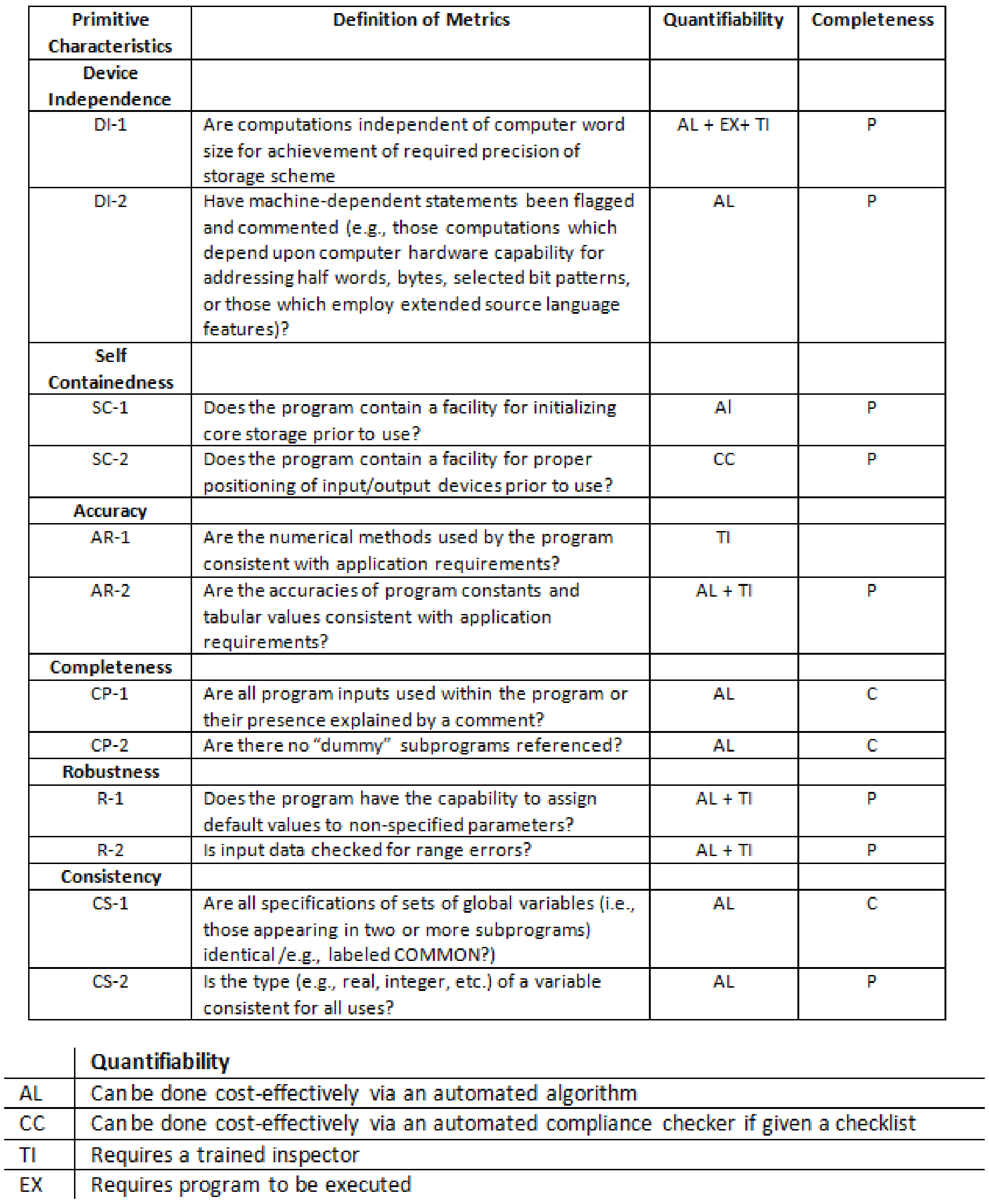
The values given in this figure are the squared correlation between two metrics. Overall, no or very low correlation has been detected. A low correlation is good for combining metrics (for more information about correlation see Section 2.5). Besides correlation and distribution, the relationship between fault probability and the metrics was analyzed. Basili et al. [5] states *"five out of the six Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during* *the high- and low-level design phases of the life-cycle"* (Lack of Cohesion is the metric Basili et al. excluded from this statement).

Another good validation of metrics is provided by Gyimothy et al. [29]. This validation uses object-oriented metrics on a Mozilla open source software project. The object-oriented metrics used are those introduced by Chidamber et al. [11] (the definition of these metrics can be seen in Section 2.2.) with slight modifications made by Basili et al. [5]. Gyimothy et al. [29] evaluated the source code of the Mozilla software project and combined this information with data from the corresponding "BugZilla" bug tracking system. The authors associated each reported bug with a class within the source code. Because of this combination of information Gyimothy et al. were able to evaluate the metrics. According to the results of the evaluation, the following two metrics achieved the best results and are therefore highly recommended: *Coupling Between Objects* and *Lines of Code*. The metrics *Weighted Methods per Class*, *Response for a class*, and *Lack of Cohesion on Methods* also performed very well.

2.4. Automatic Metrics Calculation

After defining different metrics they have to be calculated. Rubey and Hartwick [49] described their metrics as very specific and stated that they may require *"a detailed analysis of the program being evaluated"*. Because this would be a very time consuming task and therefore very expensive they suggested using computer programs to mechanize this analysis. Boehm et al. [7] even worked out which of their defined characteristics could be calculated automatically. In Figure 2.9, all of their metrics combined with "Quantifiability" and "Completeness" can be seen. Quantifiability provides information if and how this metric can be quantified automatically. The "Completeness" shows how completely the source code can be evaluated automatically. The letter "P" in completeness means partial completeness possible and "C" means completeness possible. Metrics for automatic calculation can be chosen with this evaluation.

Dandashi [14] conducted a study, the subject of which was to identify the relationship between automatic calculated metrics and software quality attributes collected via a survey. Part of the automatic calculated metrics were the metrics of Chidamber et al. [11] presented above. Correlations were found between the metrics "Number of Children", "Depth of Inheritance Tree", "Response For a Class", and "Coupling Between Objects" (The definitions of these metrics can be seen in Section 2.2.) and the quality attributes "Adaptivity", "Completeness", "Maintainability", and "Understandability" (The definition of this quality attributes can be seen in Section 2.1) of the class. Dandashi [14] observed that "**as DIT and NOC increases, (higher level of inheritance), the adaptability, completeness, maintainability, and understandability [...] decrease**". Dandashi provided the fol-



Primitive Characteristics	Definition of Metrics	Quantifiability	Completeness
Device Independence			
DI-1	Are computations independent of computer word size for achievement of required precision of storage scheme	AL + EX+ TI	P
DI-2	Have machine-dependent statements been flagged and commented (e.g., those computations which depend upon computer hardware capability for addressing half words, bytes, selected bit patterns, or those which employ extended source language features)?	AL	P
Self Containedness			
SC-1	Does the program contain a facility for initializing core storage prior to use?	Al	P
SC-2	Does the program contain a facility for proper positioning of input/output devices prior to use?	CC	P
Accuracy			
AR-1	Are the numerical methods used by the program consistent with application requirements?	TI	
AR-2	Are the accuracies of program constants and tabular values consistent with application requirements?	AL + TI	P
Completeness			
CP-1	Are all program inputs used within the program or their presence explained by a comment?	AL	C
CP-2	Are there no "dummy" subprograms referenced?	AL	C
Robustness			
R-1	Does the program have the capability to assign default values to non-specified parameters?	AL + TI	P
R-2	Is input data checked for range errors?	AL + TI	P
Consistency			
CS-1	Are all specifications of sets of global variables (i.e., those appearing in two or more subprograms) identical /e.g., labeled COMMON?)	AL	C
CS-2	Is the type (e.g., real, integer, etc.) of a variable consistent for all uses?	AL	P

	Quantifiability
AL	Can be done cost-effectively via an automated algorithm
CC	Can be done cost-effectively via an automated compliance checker if given a checklist
TI	Requires a trained inspector
EX	Requires program to be executed

Figure 2.9.: Evaluation of completeness and possibility of metrics automation from Boehm et al. [7]

lowing four statements which are supported by the previous observation: Increasing [14] "*high levels of inheritance [...] make the classes*

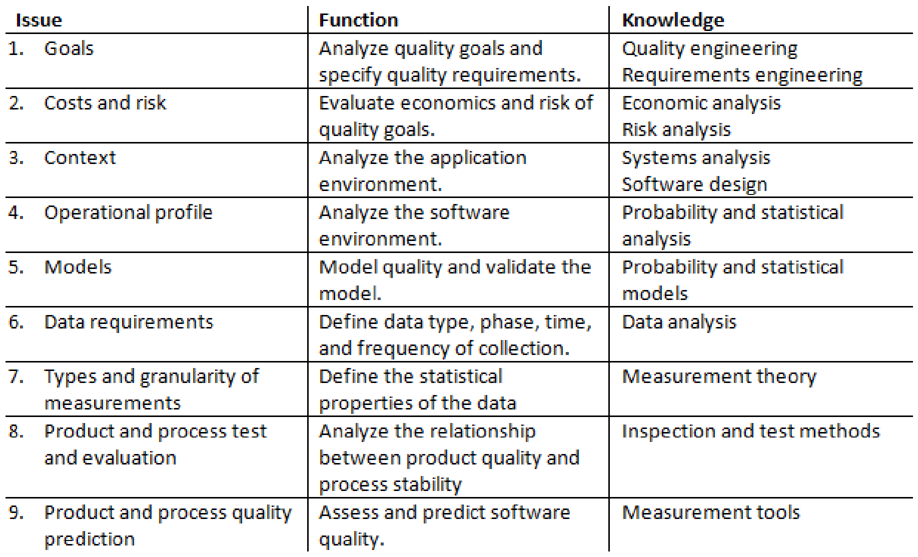
- *Increasingly difficult to adapt (and subsequently difficult to reuse), as these high inheritance levels imply an increasing degree of specialization in terms of the objects they represent*
- *More dependent on other inherited classes (more difficult to reuse than stand-alone or self-contained components)*
- *Less maintainable as stand-alone software artifacts in that their proper maintenance (perfective, adaptive or corrective) may require the modification of increasing numbers of inherited classes*
- *Less understandable as stand-alone software artifacts, in that their understandability may require the analysis of increasing numbers of inherited classes"*

Dandashi also observed a weak trend which shows that low values for RFC and CBO lead to increased quality attributes mentioned above.

2.5. Metrics Suites, Correlations and Conflicts

The major problem for Boehm et al. [7] was the fact that metrics can be in conflict with each other. One example they gave is *"added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability"*. Not only conflicting metrics improve the difficulty in choosing metric suites. An overall result which is calculated with metrics correlating will mislead. For example, if the metric suite used contains four metrics and two of them measure a very similar quality attribute, this quality attribute becomes more important than the others. This effect needs to be considered during the decision process of the metrics suite.

A software measurement suite in a broader sense is provided by Schneidewind [51]. Broader sense means a software measurement suite that no mathematical metrics are presented in this work. Instead, Schneidewind presents a concept for software quality measurement for the entire software development life cycle. This concept is based on two standards for software quality: AIAA and ANSI [1] and IEEE [30]. This so called "Issue-oriented approach" is split up into nine issues (see Figure 2.10).



Issue	Function	Knowledge
1. Goals	Analyze quality goals and specify quality requirements.	Quality engineering Requirements engineering
2. Costs and risk	Evaluate economics and risk of quality goals.	Economic analysis Risk analysis
3. Context	Analyze the application environment.	Systems analysis Software design
4. Operational profile	Analyze the software environment.	Probability and statistical analysis
5. Models	Model quality and validate the model.	Probability and statistical models
6. Data requirements	Define data type, phase, time, and frequency of collection.	Data analysis
7. Types and granularity of measurements	Define the statistical properties of the data	Measurement theory
8. Product and process test and evaluation	Analyze the relationship between product quality and process stability	Inspection and test methods
9. Product and process quality prediction	Assess and predict software quality.	Measurement tools

Figure 2.10.: The nine issues of the Issue-oriented approach for software quality measurement [51]

The "Functions" in this image define actions needed to provide all information for the issue. The "Knowledge" means the knowledge an employee must have to perform the according functions. This can help a project manager to define which personal resources are needed. Because the business view on metrics is not the topic of this thesis, no further discussion of these issues is given here. The interested reader should take a look at Schneidewind [51].

An Evaluation of Chidamber et al. [11]'s metrics suite is presented by Succi et al. [55]. They present an empirical exploration of the distributions and correlations of the Chidamber and Kemerer object-oriented metrics suite. The resulting values for the correlations indicate a high correlation between following metrics:

- "*Number of Methods*" and "*Lack of Cohesion between Methods*"
- "*Number of Methods*" and "*Response for a Class*"
- "*Response for a Class*" and "*Lack of Cohesion between Methods*"
- "*Response for a Class*" and "*Coupling Between Objects*"

(for definition of this metrics see 2.2). This means these metrics should not be used together.

2.6. The Downside of Metrics

There are not only advantages when it comes to metrics. Boehm et al. [7] even stated *"Finally, we concluded that calculating and understanding the value of a single, overall metric for software quality may be more trouble than it is worth"*. This means a single metric which leads to high quality software for every software project does not exist. In the chapters above many quality attributes and corresponding metrics have been introduced. There may be metrics which are important for every project (e.g. Consistency, Completeness) but even then it has to be defined how important each metric is for a certain project. As stated in Briand et al. [9]'s work, metrics are a very efficient instrument for quality evaluation but seem to be very context-sensitive. Briand et al. [9] evaluated several cohesion and coupling metrics concerning their ability to predict the probability that a fault occurs in a class during operation of the system. Beside the difficulty of choosing the right metrics there are also coherent or even contradictory metrics (e.g. Time Usage vs. Space Usage). Stockman et al. [54] stated that because of the difficulties in justifying a particular weighting scheme, the concept of metrics presented has very limited value. All these decisions have to be made to use metrics. This means also: If a set of metrics has been chosen for a project the quality target of the project has been defined - explicitly or implicitly.

2.7. Different Interpretations of Metrics

In this thesis metrics are all about measuring software quality attributes. This is not the only way to interpret metrics. Stockman et al. [54] do not agree with this interpretation. They take the following definition of measurement: *"Measurement is the process of empirical objective assignment of numbers to the properties of objects and events in the real world in such a way as to describe them"* [21]. This definition was mapped to the software engineering domain by the following definition of key words and phrases [54]:

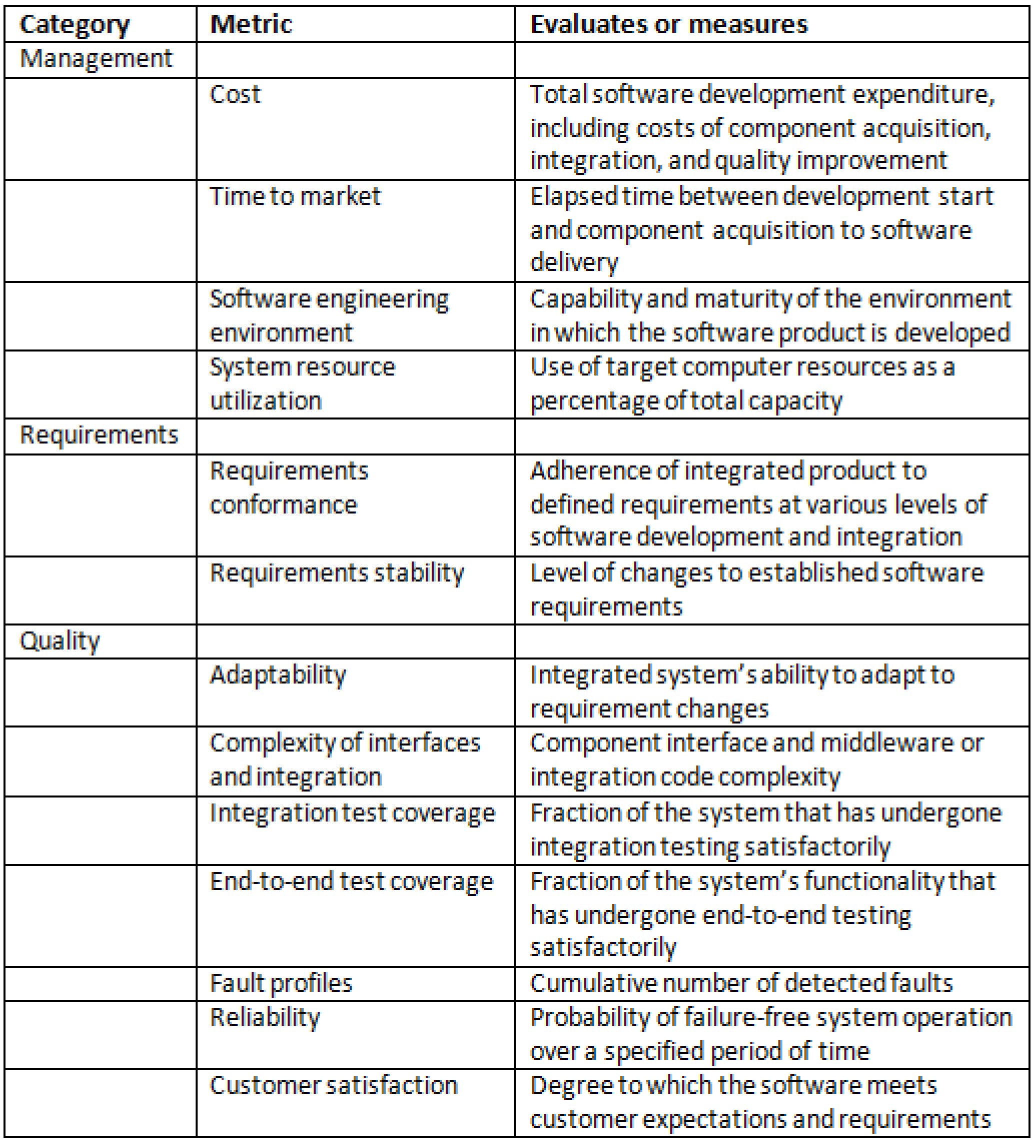
- "Empirical" - this highlights the role of experimental validation of software measures
- "Objective" - this highlights the need for independence from subject judgments of specific individuals. Measurements should be within an acceptable error bound, irrespective of who collects them
- "Assignment of numbers to the properties of objects" - this highlights the need for ordering relationships between the target objects. Such scales should accurately reflect gradations of real world properties [*]

[*]Annotation: This means: measuring an object implies to assign a value (number) to each part (properties) of the object.

- "Objects and events" - in software terms, this can be interpreted as the products and processes that characterize the software production life cycle
- "In the real world" - measurement must relate to real world issues and problems. A mathematically sound measure is of no use unless it can fit into a realistic model or theory of real world behavior

(This mapping is cited from Stockman et al. [54].)
Stockman et al. [54] conclude from the given definition of measurement and the interpretation of it given above, that "Complexity Metrics" cannot be seen as measurements. They state that *"complexity is a psychological attribute on which it is difficult to define an ordered relationship which can be objectively assessed"*. Nevertheless, Chidamber et al. [11] give an example, how to use the previously introduced Chidamber and Kemerer [13]'s metrics suite, which contains only complexity metrics, as a management tool. In this work they want to provide explanations of the metrics suite in a management context. Besides the metrics suite, they only use additional information about the size of the source code, the effort as developer time in hours, and the productivity. The result of their exploratory analysis is that high coupling and lack of cohesion can be associated with lower productivity, greater rework, and greater design effort.

Another work which focuses on management aspects of metrics is from Sedigh-Ali et al. [52]. Their mapping between management aspects and metrics can be seen in Figure 2.11. This table gives examples on how metrics can be used in management. An important chapter in this work concerns the cost of quality. The quintessence of this chapter can be summarized as follows: *"'Perfect' quality may not be achievable at finite costs [...]. Although we may be able to determine the overall CoQ (costs of quality) with reasonable accuracy, determining the amount dedicated to improving a particular quality factor is difficult because all factors interrelate."*[52]



Category	**Metric**	**Evaluates or measures**
Management		
	Cost	Total software development expenditure, including costs of component acquisition, integration, and quality improvement
	Time to market	Elapsed time between development start and component acquisition to software delivery
	Software engineering environment	Capability and maturity of the environment in which the software product is developed
	System resource utilization	Use of target computer resources as a percentage of total capacity
Requirements		
	Requirements conformance	Adherence of integrated product to defined requirements at various levels of software development and integration
	Requirements stability	Level of changes to established software requirements
Quality		
	Adaptability	Integrated system's ability to adapt to requirement changes
	Complexity of interfaces and integration	Component interface and middleware or integration code complexity
	Integration test coverage	Fraction of the system that has undergone integration testing satisfactorily
	End-to-end test coverage	Fraction of the system's functionality that has undergone end-to-end testing satisfactorily
	Fault profiles	Cumulative number of detected faults
	Reliability	Probability of failure-free system operation over a specified period of time
	Customer satisfaction	Degree to which the software meets customer expectations and requirements

Figure 2.11.: Table of metrics to evaluate existing components [52]

2.8. Additional Literature

In addition to the information presented above, Purao and Vaishnavi [48] can be highly recommended if further literature on object oriented metrics is needed. In 2002 they summarized the current complete field of research concerning object oriented software measures. Bandi et al. [4] observed the relation between maintenance performance and object oriented complexity metrics. They proposed three hypotheses:

- *H1: There is a difference in the maintenance time required to make changes to systems, depending on whether they have a low- or high-complexity designs:* $\mu_1 = \mu_2$
- *H2: There is a nonzero correlation between the complexity of a system's design and the maintenance time required to make changes to that system:* $\rho \neq 0$
- *H3: There is a nonzero linear regression relationship between the complexity of a system's design and the maintenance time required to make changes to that system:* $\beta \neq 0$

These hypotheses were proven with a case study. In this case study, two versions of the same program were given. One version with high- and one with low complexity, according to predefined metrics. Each program was then given to different students. The students had to maintain the program and report their maintenance-times. H1, H2, and H3 could be approved from the results of this study. El Emam et al. [17] focused on the confounding effect "class size" in the field of metrics. The authors discussed the correlation of coupling metrics and class size and the resulting impact on fault-proneness prediction. The research proved the existence of a confounding effect of class size. This means: "*After controlling for the size confounder, all of these associations disappear, and the differences in the product metrics odds ratios indicates a strong and classic confounding effect* " [17]. Sato et al. [50] analyzed the evolution of metrics in agile projects based on seven projects with different agile method approaches. The result showed that less agile practices lead to higher size, complexity, and coupling measures.

Another domain where metrics are very important are so-called commercial off-the-shelf (COTS) components. The quality of COTS components needs to be determined before a buying decision can be made. Sedigh-Ali et al. [52] present the advantages of metrics in this domain.

Besides the papers mentioned above, which evaluate the Chidamber et al. [11] metrics suite, there are also Tang et al. [56] and Alshayeb and Li [2]. There are also evaluations of other metrics and metrics suites: Olague et al. [44] and Alshayeb and Li [2]. Ortega et al. [46] compare different quality metric suites and introduce their own quality suite, which underlines the differences of effectiveness and efficiency in product or process quality measurements. The authors started with the quality attributes of "Functionality", "Reliability", "Usability", "Efficiency", "Maintainability", and "Portability". Each of these attributes were divided in four groups:

- Product Effectiveness
- Product Efficiency
- Process Effectiveness
- Process Efficiency

Each group of each quality attribute has its own sub-characteristics, which are based on the ISO 9126. ("*ISO 9126 defines product quality as a set of product characteristics*" Ortega et al. [46]). For example, the characteristics detected for product effectiveness of functionality are: suitability, accuracy, interoperability, and security and for product efficiency they are: correctness, structured, encapsulated, and specified. The conclusion of Ortega et al. [46] was that their quality suite "*is an effective tool for analyzing product quality, that can be used to compare different software products*".

Chapter 3

Persuasive Technology

Persuasive Technology is a complex topic which, on the one hand, depends on the technological possibilities which are expanding every day and on the understanding of human behavior on the other hand. The first section of this chapter is dedicated to human behavior change. The following sections deal with existing frameworks, a brief review of current persuasive technologies and, last but not least, a reminder about ethics.

3.1. Persuasion and Human Behavior Change

The definition of the word "persuasion" depends on to whom you are talking to, and in which context the term is used. Fogg [25] defines persuasion as *"a noncoercive attempt to change attitudes or behaviors"*. The noncoerciveness and the attempt to change attitudes or behaviors are very important in this definition. It cannot be called "persuasion" if you use force in order to change attitudes or behaviors. If the user changes his attitudes "by accident" it is not persuasion either. Miller [41] states that the phrase of "being persuaded" *"applies to situations where behavior has been modified by symbolic transactions (messages) that are sometimes, but not always, linked with coercive force (indirectly coercive) and that appeal to the reason and emotion of the person(s) being persuaded."* This definition is meant in a more global way and Miller even gives the example of nuclear power as a means of persuasion. In the context of this thesis clearly the first definition is the more appropriate one.
The definition of human behavior change is based on Fogg [24]. Fogg presents his so-called Fogg Behavior Model* "BFM", which should help the understanding of what is important in order to make humans change their behavior. This model is based on three factors, which are called "*Motivation*",

*See also the website `www.behaviourmodel.org` for further details

"*Ability*", and "*Triggers*". To achieve a certain behavior, all three factors have to be present. A person needs motivation, ability, and a trigger to behave in the desired way. Assuming that a good trigger is given, a person still needs motivation and ability.
The best conditions are, of course, that the user is highly motivated and has the ability to behave in a certain way. Even if one of the aforementioned factors is not present, Fogg states that it's possible to compensate it with another factor. For example, Tom is a programmer and has low motivation to improve the quality of his written code. However, since the tool makes it very easy to do so, he improves anyway. The list[†] of motivators and mechanisms to increase simplicity and hence the user's ability, published by Fogg is the following:

- Elements of Motivation
  - Pleasure / Pain
  - Hope / Fear
  - Social Acceptance / Rejection
- Elements of Simplicity
  - Time
  - Money
  - Physical Effort
  - Brain Cycles
  - Social Deviance (going against the social norm)
  - Non-Routine

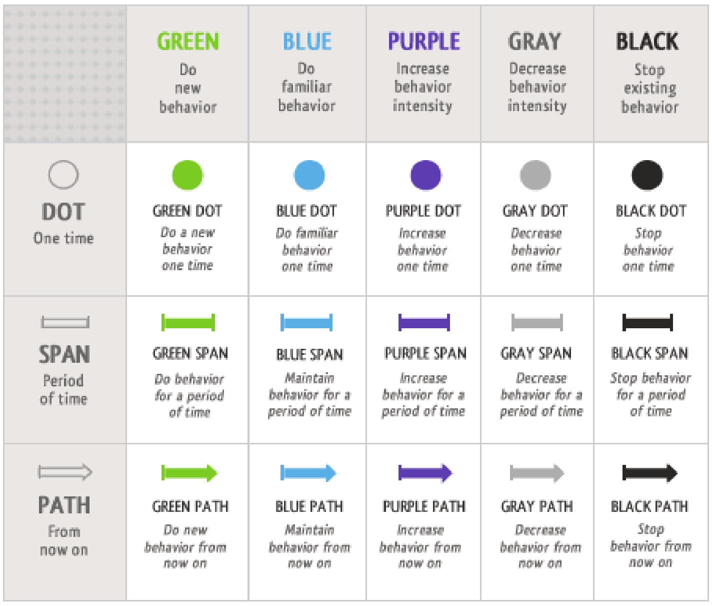
Now that we know how to increase motivation and simplicity, the last missing part is the trigger. The computer uses a trigger to tell the user that he or she should perform a specific action. When it comes to triggers, time is very important. A user won't be attentive if he or she is interrupted at the wrong time. In [31], the importance of triggering is stressed. Intille [31] calls it "Just-In-Time Messaging" and he sees the advantages of mobile devices and real-time context-aware computing in catching the perfect moment for a trigger.
A good insight into persuasion and user behavior is given by Oinas-Kukkonen and Harjumaa [43] in Section 2: "Fundamental Issues Behind Persuasive Systems". In this section they state seven postulates and reason why they are important.

1. *Information technology is never neutral*
2. *People like their views about the world to be organized and consistent*

[†]For further information please see Fogg [24]

3. *Direct and indirect routes are key persuasion strategies*
4. *Persuasion is often incremental*
5. *Persuasion through persuasive systems should always be open*
6. *Persuasive systems should aim at unobtrusiveness*
7. *Persuasive systems should aim at being both useful and easy to use*

A solid basis regarding persuasion and human behavior change is given through the knowledge of these seven postulates and the Behavior Fogg Model. Another work concerning behavior was published by Fogg and Hreha [28]. This work classifies 15 different types of behavior change within a so-called "Behavior Grid" (see Figure 3.1). This classification can help to understand behaviors and



	GREEN Do new behavior	BLUE Do familiar behavior	PURPLE Increase behavior intensity	GRAY Decrease behavior intensity	BLACK Stop existing behavior
DOT One time	GREEN DOT *Do a new behavior one time*	BLUE DOT *Do familiar behavior one time*	PURPLE DOT *Increase behavior one time*	GRAY DOT *Decrease behavior one time*	BLACK DOT *Stop behavior one time*
SPAN Period of time	GREEN SPAN *Do behavior for a period of time*	BLUE SPAN *Maintain behavior for a period of time*	PURPLE SPAN *Increase behavior for a period of time*	GRAY SPAN *Decrease behavior for a period of time*	BLACK SPAN *Stop behavior for a period of time*
PATH From now on	GREEN PATH *Do new behavior from now on*	BLUE PATH *Maintain behavior from now on*	PURPLE PATH *Increase behavior from now on*	GRAY PATH *Decrease behavior from now on*	BLACK PATH *Stop behavior from now on*

Figure 3.1.: The "Behavior Grid" describes different behavior change categories. [28] †

their change. The grid can help to identify different types of behavior change. Based on this information, different persuasive concepts can be chosen. There will be different persuasive steps necessary depending on whether the user should have "a new behavior from now on" (e.g.: "Go for a run every Saturday morning" for a person who did not go running before.) or do "a familiar behavior one time" (e.g.: "Go and wash your hands now").

†This illustration is taken from `http://captology.stanford.edu/projects/behavior-wizard-2.html`

3.2. Models, Frameworks and other Concepts

Fogg [25] states that *"We have no universal theory or framework for persuasion"*. This section will show concepts which may be helpful to design a persuasive system depending on the given situation and context.

Fogg introduced the so called "Functional Triad" [26],[25]. This triad is thought of as a framework for analysis and design. The triad reflects the three roles a computer can play concerning the interaction with humans. Depending on the role the computer is given, persuasion strategies will differ and different possibilities are given. Fogg gives each role additional subcategories to simplify the choice of possible strategies. To aid understanding, an example is provided for each subcategory.

1. Computers **as tools**:
   - **Increasing self-efficacy**: Provides a metrics tool which helps to find and improve quality shortcomings. This tool will help a user who has to improve quality but doesn't know how. He or she will feel more confident to achieve the specified goal and will act according to the suggestions of the tool. For example, the user needs to reduce the complexity of a function but does not know where to start from. The tool will show a current complexity value and possible solutions to decrease this value. The user can try the solutions and immediately gets feedback through the new complexity value
   - **Providing tailored information**: Fairchild et al. [19] published the Foot-LITE system. This system delivers feedback to the driver of a car according his or her driving style. This project's goal is to achieve eco-friendly, safe and efficient driver behavior [‡]
   - **Triggering decision making**: Provides a list of files with software quality flaws on Eclipse startup. The programmer will likely work on one of them
   - **Simplifying or guiding people through a process**: The spelling correction system of Microsoft Word
2. Computers **as media**
   - Computers that **simulate cause and effect**: An annuity calculator: Let the user input an annuity and years (cause) and show what he or she will receive (effect). Customers will be likely to start saving earlier
   - Computers that **simulate environments**: Change of music and light is often used in movies to manipulate feelings (e.g., fear, joy, or sadness)

[‡]A website about the current Foot-Lite project can be seen here [http://www.foot-lite.net/]

- Computers that **simulate objects**: A very famous example was the Tamagotchi in the 1990's. It simulated a pet which needed the care of its owner

3. Computers **as social actors**

    - Computers that **provide social support**: This is often used in computer games. At the beginning of a game a character helps the user to learn the interaction with the game. This character can be invoked every time the user needs further help. Persuasion is possible during this aiding process
    - Computers that **model attitudes and behaviors**: Fogg [25] says *"Computer-based characters, whether artistically rendered or video images, are increasingly likely to serve as models for attitudes and behaviors"*
    - Computers that **leverage social rules and dynamics**: The social agent iCat (see Figure 3.2), presented by [40], gives feedback about a chosen washing program. The feedback is described as social, because it is represented with facial gestures. This feedback led to lower energy consumption results rather than information based feedback

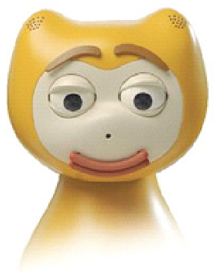


Figure 3.2.: The social feedback agent iCat from Midden and Ham [40] gives feedback through facial gesture

The examples in the computer as tools, computer as media, and computer as social actors sections show that a specific function of a persuasive program can be categorized in more than one of the above introduced categories. "Providing tailored information" can lead to "model attitudes and behaviors" or "leverage social rules and dynamics" can lead to "increased self-efficacy". Some subcategories describe what a program actually does (e.g. "Providing tailored information" or "Simulate cause and effect") and others describe the desired result (e.g. "Increasing self-efficacy" or "Model attitudes and behaviors"). This seems to be a flaw in Fogg's concept, but nevertheless it helps to understand and implement persuasive functionality.

Additionally, Fogg provides persuasive technology tools, depending on a given role. He also underlines the importance of computer credibility. The persuasive tools were advanced by Oinas-Kukkonen and Harjumaa [43]. They criticize that there is no explanation of how to transform a given role into software requirements or concrete implementation out of the triad concept. Oinas-Kukkonen and Harjumaa [43] analyzed the software design process and where persuasive elements should come into play. They see the requirement specification as one of the most important phases in software development. In this phase, in addition to the functional requirements, non-functional requirements (system qualities) are defined. These system qualities should give information about the system's persuasiveness. System qualities are related directly to system features. Oinas-Kukkonen and Harjumaa [43] give a concept, composed of principles, about how to design or even implement persuasiveness depending on the category of the system features. The complete list consists of 28 principles. Below, a part of this list and each category is given and mapped to a popular website.

1. Primary task support (mapped to Facebook.com)
   a) **Reduction**: Make available functions as simple as possible in the specific context: Commenting and sharing links are possible with one click
   b) **Tunnelling**: Guide the user through a process and persuade him or her on the way: When new features are implemented, the user can take a "tour" through them and sees motivating messages why he or she should use them
   c) **Tailoring**: Information and media have to be tailored to the user: Comments, photos, and common interests are provided on each part of Facebook.com
   d) **Personalization**: The user can personalize the software: The user can provide various media he or she likes. Additionally there are Facebook extensions available which allow the user to change for example, the main color of the website
   e) **Self-monitoring**: Let the user keep track of his or her performance: The user can always see what he or she has posted and how other users respond to it
2. Dialogue support (mapped to Facebook.com)
   a) **Praise**: Give the user positive feedback: Third party games on Facebook.com praise the user when he or she perform certain actions
   b) **Rewards**: Give the user virtual rewards: Third party games on Facebook.com award the user with rewards, and all of the user's friends can see it
   c) **Reminders**: Remind the user of desired behavior: If a long time has passed where no communication has taken place between the user and a user's friend, Facebook.com reminds the user to stay in contact
   d) **Suggestion**: Suggest desired behavior during the use of the software: New friends or media are suggested to the user

3. System credibility (mapped to Wikipedia.org)

    a) **Trustworthiness**: Information has to be truthful and unbiased: Every user can edit articles and has to provide external literature to prove it. If insufficient literature is provided, a warning gets added to the article

    b) **Verifiability**: The information must be easy to verify: Every article is based on external literature. The provided literature gives a high level of verifiability

4. Social support (mapped to Wikipedia.org)

    a) **Normative Influence**: Use normative influence or peer pressure to achieve a target behavior: Articles have to be written in a very specific way. If a user does not stick to this, the article gets deleted very quickly by another user

    b) **Social learning**: Give the possibility to observe other user's behaviors: It is possible to see the evolution of articles

Oinas-Kukkonen and Harjumaa [43] give concrete examples for each of these principles.

Besides design patterns which address the structure of source code there are visual design patterns. Tung and Chou [57] proposed three visual design principles for shaping conceptual designs. These principles are explained on the basis of the Figure 3.3. This figure shows the "Simple Joy" concept, which should help people to calm down and take their time while eating. This concept demonstrates how visual design effects human behavior.

- **Implicit interaction**: The user performs an action which is not aimed at interacting with a computer but can be grabbed as input. The system recognizes that people are eating
- **Aesthetic feedback**: Feedback in the surroundings of the user. The system responds to the eating family with decorative designs
- **Emotional engagement**: The feedback should evoke emotional response to support behavior change. The system shows relaxing forms on the table to calm down the users

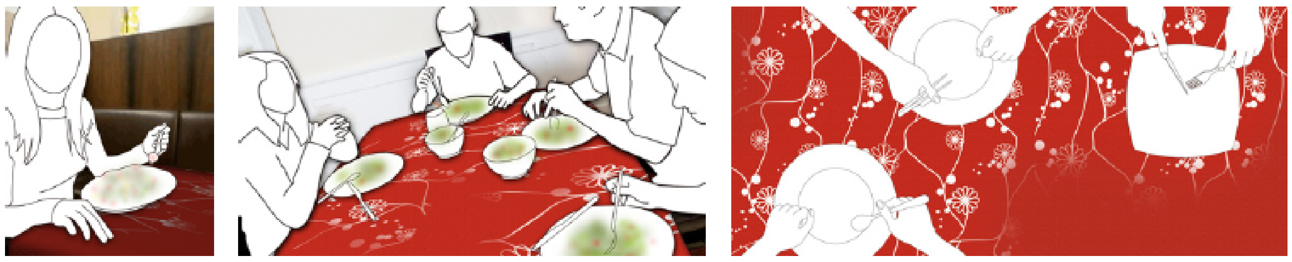


Figure 3.3.: "Simple Joy" persuasive tablecloth to encourage stress-free eating [57]

3.3. Persuasive Technology Today

Fogg [22] invented the term “Captology”, which means Computers as Persuasive Technology. Figure 3.4 visualizes which parts of the topics of computers and persuasion overlap. Fogg [26] sees several

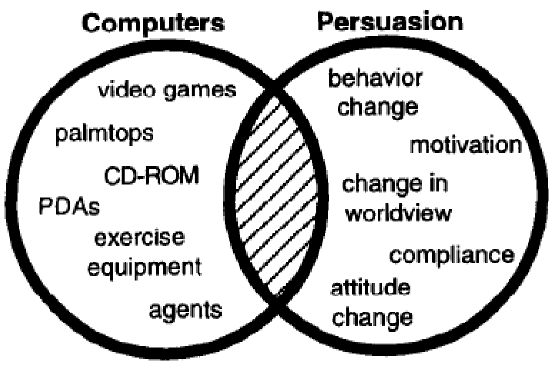


Figure 3.4.: The overlapping of Persuasion and Computers shows the area for “Captology” [22]

advantages of computers as persuaders in contrast to human persuaders:

- More persistent than human beings
- Offer greater anonymity
- Manage huge volumes of data
- Use many modalities to influence
- Scale easily
- Go where humans cannot go or may not be welcome

An interesting example of a persuasive system is given by Orino and Kitamura [45]. They developed a shop-system with the goal of persuading a customer to buy a specific camera. The customer starts with picking a camera he or she likes. Then the persuading process starts and the system tries to change the customer’s decision to pick another camera. The persuasion process can be seen in Figure 3.5. During the designed process disadvantages of the user chosen camera and advantages of the promoted camera are shown to the user. Although the persuasion process was well-designed, only two out of twenty experimental subjects were persuaded. A possible reason is that many subjects didn’t recognize the persuasive functions or the presented advantages of the promoted alternative. The result of this experiment is an indicator that design is very important in the area of persuasion.

Mahmud et al. [36] created an agent to decrease energy consumption. They tested whether a friendly agent is more likely to persuade users than an unfriendly one. They got an interesting result: a significant correlation between friendliness and trustworthiness. Khaled et al. [32] explored

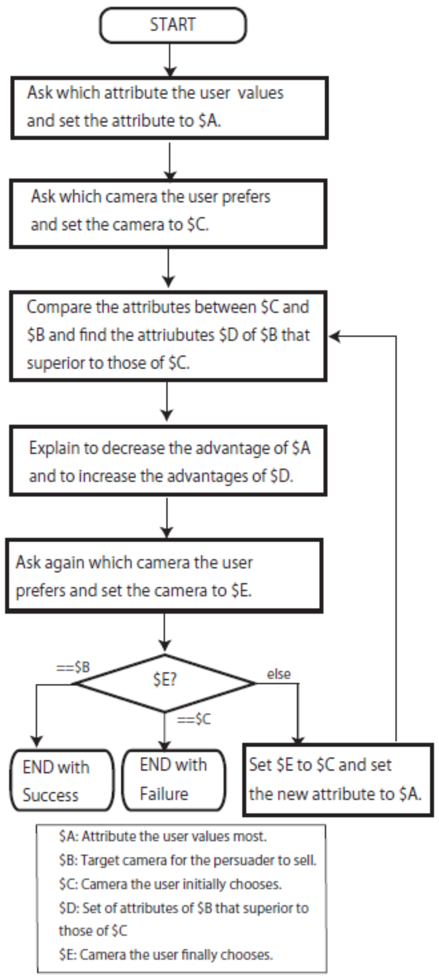
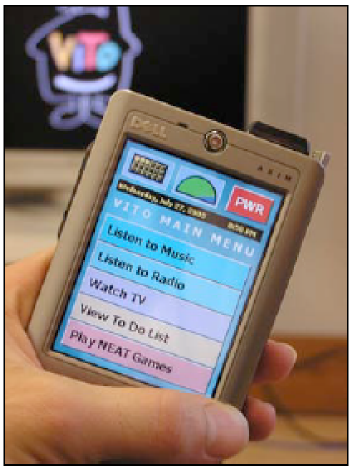

START
Ask which attribute the user values and set the attribute to $A.
Ask which camera the user prefers and set the camera to $C.
Compare the attributes between $C and $B and find the attriubutes $D of $B that superior to those of $C.
Explain to decrease the advantage of $A and to increase the advantages of $D.
Ask again which camera the user prefers and set the camera to $E.
$E?
==$B
else
==$C
END with Success
END with Failure
Set $E to $C and set the new attribute to $A.
$A: Attribute the user values most.
$B: Target camera for the persuader to sell.
$C: Camera the user initially chooses.
$D: Set of attributes of $B that superior to those of $C
$E: Camera the user finally chooses.

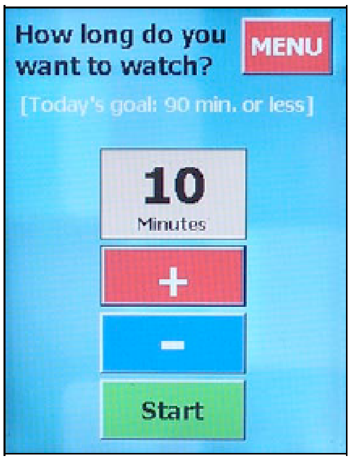
Figure 3.5.: Orino and Kitamura [45]'s process to persuade the user to buy a product different to the user's first choice.

social software regarding persuasiveness. They concluded that social software has a high persuasive potential. This conclusion is based on the detection of the following three persuasive elements in social software: *"Affiliation looks at how community identification increases motivation, while access focuses on how the beliefs and opinions of other members serve in a persuasive role even to non-contributors. Finally, participation relates to how contributing members are persuaded to act in particular ways to uphold group identity and maintain group harmony"*

King and Tester [33] see the main domains of persuasive technology in marketing, health-care, and safety. Indeed, the topic of health-care has many followers in the field of persuasive technology. Mazzotta et al. [37] developed "Portia", a persuasion system for more healthy eating. Tung and Chou [57] show concrete implementations of three persuasive gadgets, together with their principles. All of them are designed to increase health. For example, “Simple Joy” is designed to encourage people to take their time when eating and to slow down for a moment. While people are eating, their table cloths reacts on the movement of their cutlery and gives beautiful visual feedback (see Figure 3.3). Nawyn [42] developed a persuasive television remote control for the promotion of health and well-being (see Figures 3.6(a), 3.6(b), and3.6(d)). This television remote control gives the user feedback about his or her television usage and helps to reduce the time in front of the TV. Fairchild et al. [19] worked on a Feedback System to encourage safe, ecological and efficient driving called “Foot-LITE”. Foot-LITE delivers feedback whether he or she is driving in an eco-friendly manner or not. Fairchild et al. [18] explored the field of unconscious persuasion.



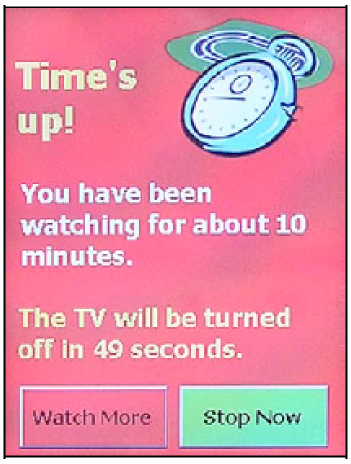
(a) A prototype for a health and wellbeing improving television control called ViTo. Nawyn [42]


How long do you want to watch?

[Today's goal: 90 min. or less]

10 Minutes

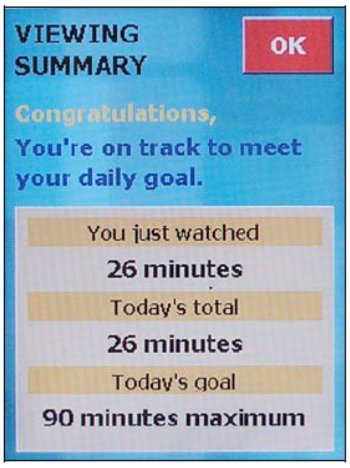
(b) Television timer from the ViTo television control. [42]


Time's up!

You have been watching for about 10 minutes.

The TV will be turned off in 49 seconds.

(c) Feedback and warning from ViTo, the persuasive television control [42]


VIEWING SUMMARY

Congratulations, You're on track to meet your daily goal.

You just watched
26 minutes
Today's total
26 minutes
Today's goal
90 minutes maximum

(d) Feedback and summery from ViTo, the persuasive television control [42]

Figure 3.6.: Examples of the use of ViTo, the persuasive television remote control [42]

3.4. Ethics of Persuasive Technology

The last part of this chapter is about ethics. The act of persuading other people can raise ethical issues. In connection with a system that is thought of as objective, even more issues arise. Berdichevsky and

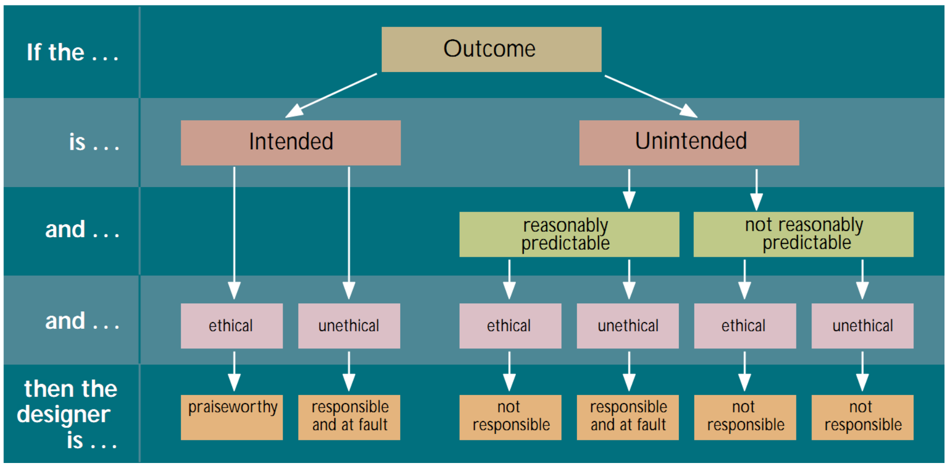


Figure 3.7.: Persuasive designers have to take responsibility for the outcome of their design under specific circumstances. This flow-chart diagram helps to decide when the designer has to be responsible. [6]

Neuenschwander [6] state: “*Persuasion apparently distributes responsibility between the persuader and the persuaded*”. This means with designing a persuasive software the developer has to take responsibility not only for the software but also for the actions users take because of using it. Figure 3.7 sketches when persuasive designers should take responsibility based on an intent. If the persuasive designer did intend an outcome or did not intend but could have reasonably predicted it, he or she has to take responsibility for this outcome.

In [26], a complete chapter is dedicated to this part of Persuasive Technology. The knowledge of Persuasive Technology contains great power. One critically regarded point of persuasion is that the user might not even notice that he or she is subject to persuasion. Some good examples of the impact persuasive technology could have are given in [26]. One of them is given here:
“Julie has been growing her retirement fund for almost 20 years. To optimize her investment strategy, she signs up for a Web-based service that reportedly can give her individualized expert advice. Using dramatic visual simulations and citing expert opinion, the system persuades Julie to invest more in the stock market and strongly recommends a particular stock. Two months later, the stock drops dramatically and Julie loses much of her hard-earned retirement money. Although the system has information about risk, the information isn’t prominently displayed. Nor is the fact that the site is operated by a company with major financial investment in the company issuing the stock.”

Not only computers can be persuasive. That is why Fogg [26] defined six concrete ethical issues which are only applicable to persuasive technology. These ethical issues are:

- *The Novelty of the Technology can mask its persuasive intent*
- *Persuasive Technology can exploit the positive reputation of computers*
- *Computers can be proactively persistent*
- *Computers control the interactive possibilities*
- *Computers can affect emotions but can't be affected by them*
- *Computers cannot shoulder responsibility*

The ethics of persuasive technology was a highlighted topic at the First International Conference on Persuasive Technology for Human Well-Being, Persuasive 2006. Atkinson [3] represented this topic. Atkinson states: *"My own conclusion is that captology requires an immediate ethical safeguard and this could be fulfilled if the purpose of the persuasion, the macrosuasion's intent, was exposed at the beginning of one's engagement with a program. It would then be possible for the user to determine the program's relevance and exercise their right to accept or reject its offering."* Another work on ethics in persuasive technology is provided by Davis [15]. He states the question: "Ethical issues in persuasive computing: unique or common?". In our everyday life, persuasion is everywhere. From salesmen up to politicians there are many people or companies which would like to persuade us of something. Although this seems very dangerous, persuasion does not have to be something bad. Persuasion can, for example, be used to eat healthier, do more sports and drive more careful. Davis [15] even states that it is difficult for designers not to change others people's behaviors, because most information technologies require complex interaction and therefor influence behavior.

Chapter 4

Combining The Aspects

4.1. Basic Idea

The first step in creating an Eclipse plug-in which can assist a programmer in writing proper code was to define its functionality. This first step started with Felfernig, Pribik, Steinparz, and Leitner [20]. The persuasive elements that should be used have been chosen in this paper. In Section 3, different persuasive concepts were presented, but not all of them were included in the implementation. The following section of this chapter is about a pre-study, which was conducted in order to gain a little insight onto whether people think it is possible to be persuaded to improve their code quality.

4.2. Acceptance Pre-Study

Ten persons where asked about their opinion concerning the use of a motivational tool for code quality improvement. The questionnaire contained only open questions to get as much information as possible. Below, the questionnaire and a summary of the answers are provided.

1. Do you think it is possible that visualization of quality improvement potentially motivates code improvement?
    - "Yes, because flaws get obvious"
    - "Maybe an additional reward-system would even more motivate"
    - "At least it will increase the awareness of quality flaws and helps to see them"
    - "I would love to get aid for quality-improvement"
2. Would you use such a plug-in or even use it in your company?
    - "Yes"
    - "Depending on the used metrics"
3. Which advantages would you expect from the use of the plug-in?
    - "A consistent code standard"
    - "Learn how to develop high quality code"
    - "Improved maintainability"
    - "Less software errors and more awareness to them"
4. Which additional functionality would you like?
    - "Automatic improvement"
    - "The possibility to configure the used metrics"
    - "Blank out unwanted code-improvement markings"
    - "Concrete improvement steps and tips"
    - "Self-monitoring about my quality improvements"
5. Which functionality do you not want?
    - "Monitoring given to supervisor"
    - "Automatic improvement"
    - "Prevent compiling or submitting to version control systems"
6. How would you measure if there actually was a quality improvement?
    - "Less maintenance costs"
    - "Less orientation time in foreign code"
    - "Higher chance of code reuse"
    - "Learning effect of the programmer"

This positive feedback supported the idea of creating a prototype. The development of this prototype is described in the sections below.

4.3. Design Study

Besides defining the functionality, an additional step was needed before starting to implement the tool. Relating to the theory of Fogg concerning motivation and ability (simplicity), the interface has to be very easy to understand and handle. This is why a study was conducted concerning only the icons which should be used in the plug-in. A programmer should immediately know what the different icons mean. For this study, ten students answered a questionnaire, which consisted of different icons and questions concerning what they think the icon could mean. They were asked if specific icons were good for representing specific situations. This questionnaire was divided into four sections. The first section is about an image to show the user that he or she has improved the code. The second section is about icons which should give the user feedback about the current quality of a specific piece of code (e.g., package, class or method). See Figures 4.1, 4.2 and 4.3 for some examples. In the third section, icons are presented which may be used to get the user's attention to read further information about his or her source code. In the fourth and last section, an icon for an additional quality check tool is presented. Based on this study, the icons for the plug-in were chosen. This study gave the

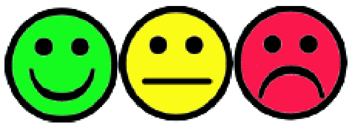


Figure 4.1.: Quality feedback icons variant 1

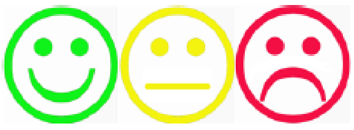


Figure 4.2.: Quality feedback icons variant 2

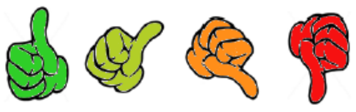
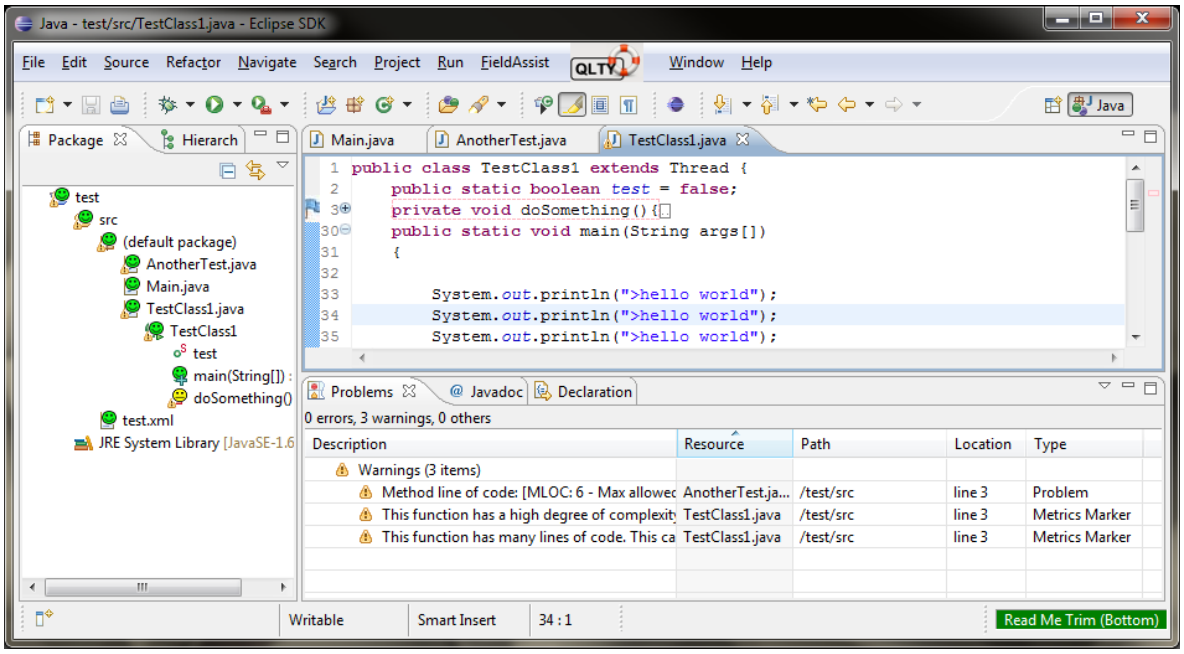


Figure 4.3.: Quality feedback icons variant 3

information needed to achieve a high acceptance of the plug-in later on. The icons in Figure 4.1 were chosen for the feedback on the project tree in Eclipse.

4.4. Tool Description “Persuasive Quality Improver”

The goal of the “Persuasive Quality Improver” is to motivate the software developer to write high quality source code according to a given metrics suite. This leads to two basic functional parts. The first part is to determine: “Is this a proper source code?” The second part is about: “How can I show evaluation results to the developer in a way that he or she cares?” These two parts need the knowledge of software metrics and persuasive technology. In both topics there are many acknowledged techniques, as mentioned in Sections 2 and 3. The first part was to choose some of those techniques and think about how to implement them into a development environment. The Eclipse IDE was chosen as the development environment. Eclipse has the advantages of a widely used open source project and the common programming language Java. This means that it is possible to write a plug-in which can be easily added in most Eclipse environments. Furthermore, many people can use it because they already use the normal Eclipse IDE. The Eclipse IDE with the quality plug-in is shown in Figure 4.4.


Java - test/src/TestClass1.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run FieldAssist QLTY Window Help
Java
Package Hierarch
test
src
(default package)
AnotherTest.java
Main.java
TestClass1.java
TestClass1
test
main(String[]) :
doSomething()
test.xml
JRE System Library [JavaSE-1.6
Main.java AnotherTest.java TestClass1.java
Problems Javadoc Declaration
Writable
Smart Insert
34 : 1
Read Me Trim (Bottom)

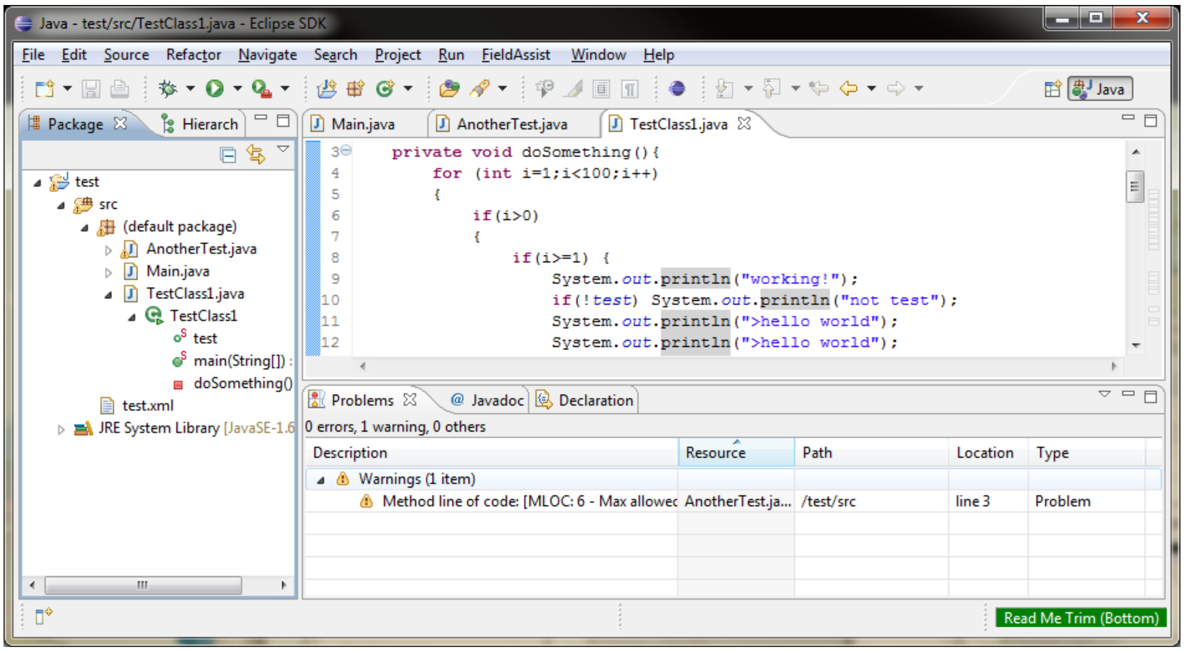
```java
public class TestClass1 extends Thread {
    public static boolean test = false;
    private void doSomething(){
    public static void main(String args[])
    {

        System.out.println(">hello world");
        System.out.println(">hello world");
        System.out.println(">hello world");
```

0 errors, 3 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (3 items)				
Method line of code: [MLOC: 6 - Max allowec	AnotherTest.ja...	/test/src	line 3	Problem
This function has a high degree of complexity	TestClass1.java	/test/src	line 3	Metrics Marker
This function has many lines of code. This ca	TestClass1.java	/test/src	line 3	Metrics Marker

Figure 4.4.: Eclipse IDE with the developed plug-in

This is how each Eclipse user will see his or her code. There is not much difference to a normal Eclipse environment, as can be seen in Figure 4.5. An important point was to keep the familiar look of the Eclipse IDE. This will keep the training period as short as possible.



```
    private void doSomething(){
        for (int i=1;i<100;i++)
        {
            if(i>0)
            {
                if(i>=1) {
                    System.out.println("working!");
                    if(!test) System.out.println("not test");
                    System.out.println(">hello world");
                    System.out.println(">hello world");
```

0 errors, 1 warning, 0 others

Description	Resource	Path	Location	Type
Warnings (1 item)				
Method line of code: [MLOC: 6 - Max allowec	AnotherTest.ja...	/test/src	line 3	Problem

Figure 4.5.: Eclipse IDE without the developed plug-in

A process oriented approach to create persuasive technologies was published by Fogg [27]. This approach consists out of the following eight steps, which the development of the “Persuasive Quality Improver” also was based on:

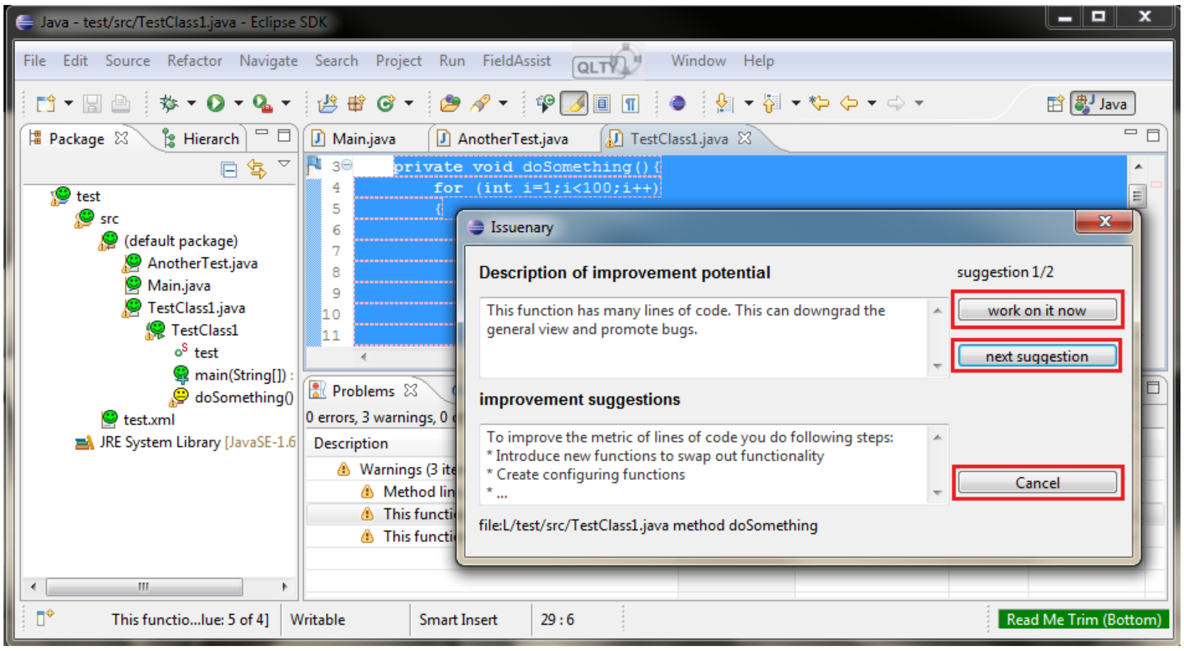
1. **Choose a simple behavior target**: The programmer should do more refactoring and write high quality code.
2. **Choose a receptive audience**: Programmers during developing phase.
3. **Find what prevents the target behavior**: Lack of time and knowledge.
4. **Choose a familiar technology channel**: Eclipse developing environment.
5. **Find relevant examples of persuasive technology**: Warnings of the compiler which are shown to the user inside the Eclipse IDE.
6. **Imitate successful examples**: The default warning view of the Eclipse IDE is known by the Eclipse users. This view is also used by the plug-in.
7. **Test and iterate quickly**: This was done during the design studies.
8. **Expand on success**: This part needs future work. Ideas on how to expand the current system can be found in Section 4.8

These eight steps represent a very detailed and easy-to-follow concept to design a persuasive system.

4.5. Chosen Persuasive Concepts

As seen in Section 3, there are many persuasive techniques. In [20], some concepts were chosen to be implemented in this tool. These concepts are *tunneling, suggestion, conditioning and self-monitoring*. The following descriptions of the techniques tunneling, conditioning and self-monitoring were presented in Felfernig, Pribik, Steinparz, and Leitner [20]:

Tunneling Technology or *Guided Persuasion* is a persuasion type where users are guided *through a predefined sequence of actions or events, step by step.* [23]. This part of persuasion performs the so-called *quality guide* or *Issuenary*. The name Issuenary was chosen, because it guides through different quality issues. This quality guide accompanies the user through a set of problematic software artifacts. During this process the quality guide provides explanations as to why these artifacts are problematic and which actions should be taken into account in order to improve the quality of these artifacts. The quality guide can be seen in Figure 4.6. As can be seen the user has only three buttons to choose from.



```
private void doSomething(){
    for (int i=1;i<100;i++)
    {
```

Description of improvement potential suggestion 1/2

This function has many lines of code. This can downgrad the general view and promote bugs.

improvement suggestions

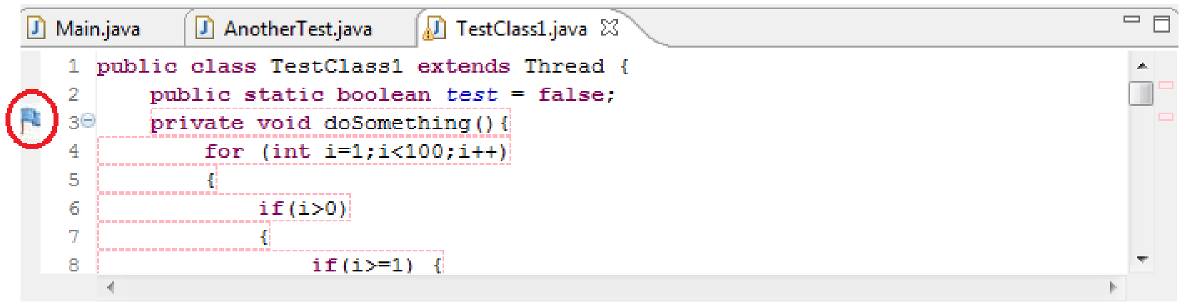
To improve the metric of lines of code you do following steps:
* Introduce new functions to swap out functionality
* Create configuring functions
* ...

file:L/test/src/TestClass1.java method doSomething

Figure 4.6.: Eclipse with the developed plug-in and the activated quality guide (Issuenary).

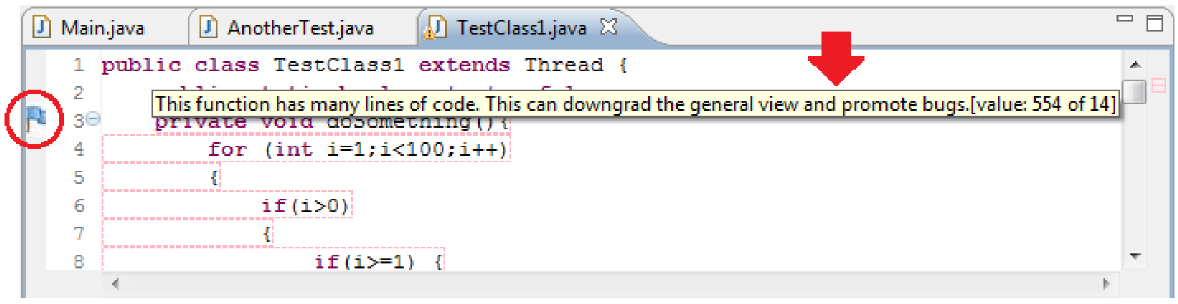
He or she can choose to work on this piece of code, see the next suggestion or cancel the process. If the user chooses to work on it, the dialog will close and the piece of code will be presented directly to the user. This means that the corresponding source file will be opened at the line of source code where the piece of code the user wants to improve starts. If the "next suggestion" button is pressed the user gets information about another piece of code which could need improvement. To make it easier for the user to understand where this specific piece of code is located within the project, in addition to just writing the file name and the method name at the bottom of the dialog, the file opens automatically in the background and the user can see the code behind the dialog.

Suggestion is a persuasive technology which means "*intervening at the right time*" [23]. For example, the system should provide advice regarding software artefact quality improvements at the right time. In this implementation the suggestion is placed in the marker and the quality guide. The user can see that there is a marker directly in the section where he or she is currently working. Every marker is placed on the left side of the line of code it refers to. Additionally, each marker of a source file is visible on the right side of the source code (see Figure 4.8 and 4.7). In these figures it is also shown



```
public class TestClass1 extends Thread {
    public static boolean test = false;
    private void doSomething(){
        for (int i=1;i<100;i++)
        {
            if(i>0)
            {
                if(i>=1) {
```

Figure 4.7.: Suggestion: The Eclipse marker shows the user which code has to be improved.



This function has many lines of code. This can downgrad the general view and promote bugs.[value: 554 of 14]

```
public class TestClass1 extends Thread {
    private void doSomething(){
        for (int i=1;i<100;i++)
        {
            if(i>0)
            {
                if(i>=1) {
```

Figure 4.8.: Suggestion: The Eclipse marker shows the user how to improve the code (textual recommendation)

that a marker can highlight specific code parts for the user so he or she knows the corresponding code lines. This means developers see that there are suggestions for improvements directly when they are working on that code. Each marker has some text to motivate the programmer to change the code, which is shown when the user brings the mouse courser over the marked source code. When the users want to improve the code, they can use the quality guide. A detailed list is shown on how exactly the code can be altered (see Figure 4.6).

Conditioning technology is *"a computerized system that uses principles of operant conditioning to change behaviors"* [23]. Operant conditioning is a method which uses positive reinforcement to achieve a certain behavior (change) [23]. Different types of feedback icons were identified which can achieve such positive reinforcements. For example, every time the quality of a certain software artefact has been improved from one build to the next, the system will show a motivating image (see Figure 4.9). In the package explorer view every artefact that has been improved gets an extra feedback

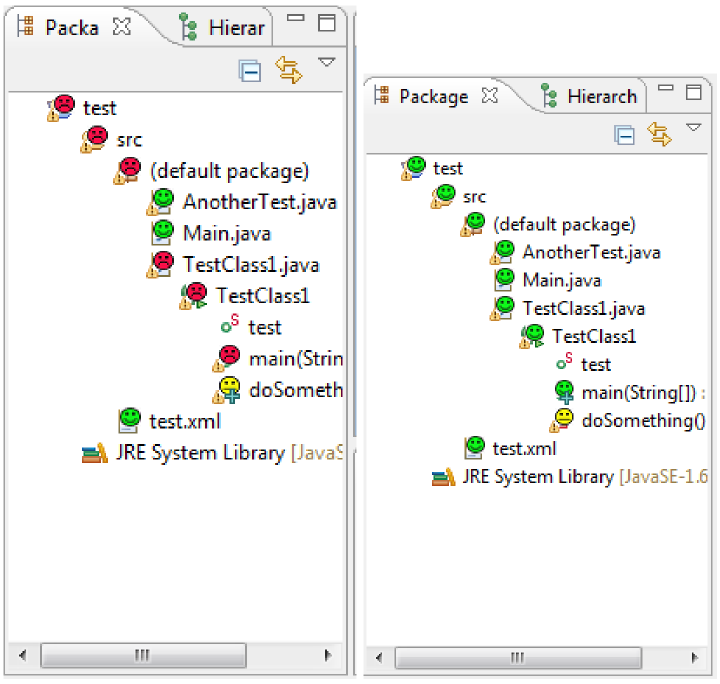
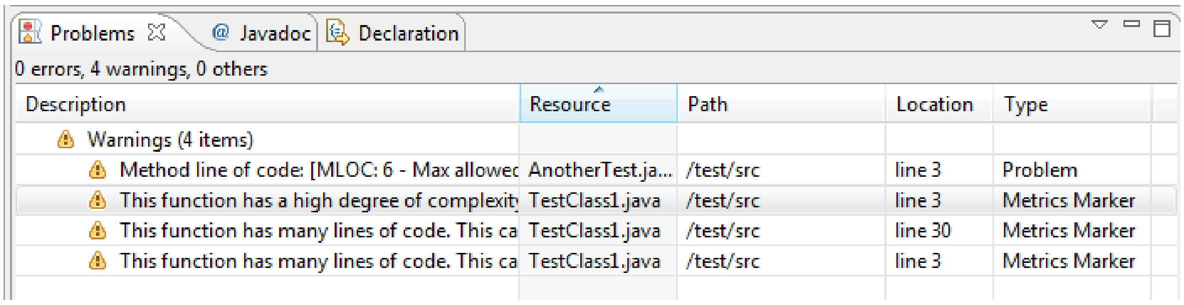


Figure 4.9.: The plug-in's package explorer close-up with very good software quality in the right window (nearly all icons green [bright]) and worse software quality in the left window (many red icons [dark])

icon. There is no negative feedback on quality losses. One of the reasons for this decision ist that at the beginning of the programming each source code is, from the metrics point of view, perfect because it is empty. Every time the user compiles, the evaluation process will determine quality losses. Another reason is that conditioning should give positive feelings primarily in this case. Every other technique marks the negative parts of the software, so there has to be some balancing.

Self-Monitoring is a persuasive technology which "*allows people to monitor themselves to modify their attitudes or behaviors to achieve a predetermined goal or outcome*" [23]. In our context we show several feedback icons that are constantly shown to the user in different areas of the Eclipse environment. The package explorer includes an overview of the quality of the software artifacts (see Figure 4.9). With the "Persuasive Quality Improver" the user gets a quick overview of which parts are in most need of improvement. More detailed information regarding low-quality software artifacts is provided in the Eclipse marker view (see Figure 4.10). In addition, the complete evaluation of the



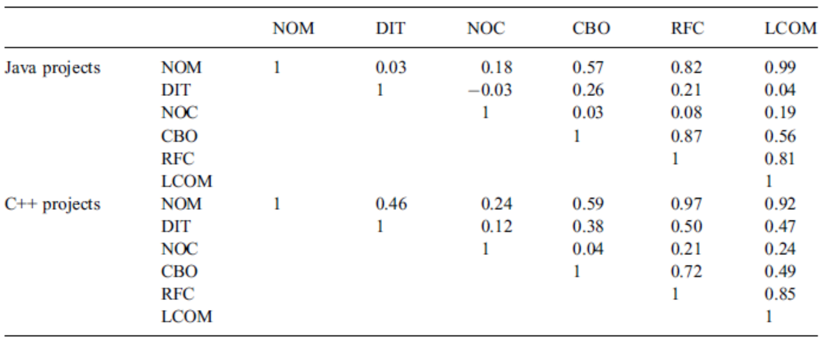
0 errors, 4 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (4 items)				
Method line of code: [MLOC: 6 - Max allowec	AnotherTest.ja...	/test/src	line 3	Problem
This function has a high degree of complexity	TestClass1.java	/test/src	line 3	Metrics Marker
This function has many lines of code. This ca	TestClass1.java	/test/src	line 30	Metrics Marker
This function has many lines of code. This ca	TestClass1.java	/test/src	line 3	Metrics Marker

Figure 4.10.: The Eclipse Marker View close-up including some improvement suggestions for the user.

quality is logged in the background. This has only been used for the user study so far, but can also be used for additional self-monitoring tools.

4.6. Implemented Metrics



		NOM	DIT	NOC	CBO	RFC	LCOM
Java projects	NOM	1	0.03	0.18	0.57	0.82	0.99
	DIT		1	−0.03	0.26	0.21	0.04
	NOC			1	0.03	0.08	0.19
	CBO				1	0.87	0.56
	RFC					1	0.81
	LCOM						1
C++ projects	NOM	1	0.46	0.24	0.59	0.97	0.92
	DIT		1	0.12	0.38	0.50	0.47
	NOC			1	0.04	0.21	0.24
	CBO				1	0.72	0.49
	RFC					1	0.85
	LCOM						1

Figure 4.11.: Correlation matrix of Chidamber and Kemerer [13]'s metrics suite by Succi et al. [55]

The complete evaluation part of the “Persuasive Quality Improver” is based on software metrics. As mentioned in Chapter 2, there are many used and approved metrics defined. This means that the most difficult process was to decide on which metrics give meaningful values and how they should be combined. Fortunately, there are papers that provide answers to both questions. A metrics suite is provided with regard to Chidamber and Kemerer [13] which evaluates a couple of metrics by different properties. This evaluation gives information about the quality of these metrics. Evaluation of the metrics suite was made by Succi et al. [55] and Basili et al. [5]. They took the metrics of the metric suite from Chidamber and Kemerer [13] and evaluated their correlation. If there is a high correlation between two metrics they should not be used together because there is no further information when using both, but rather a big disadvantage. The use of highly correlated metrics can distort the overall rating of the source code because the aspect the metrics are looking at is counted twice. In Figure 4.11 and 2.8, the correlation between two metrics can be seen. Here is a short explanation of the Figure 4.11: The evaluated metrics are Number Of Methods (NOM), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object classes (CBO), Response For a Class (RFC) and Lack of Cohesion in Methods (LCOM) (definition of these metrics can be seen in Section 2.2). The value means the weighted correlations between two metrics. The lower the value, the better both of the metrics can be used together.

$$\boxed{\begin{array}{l} DIT = MAX(d_1, d_2, ..., d_i, ..., d_n) \\ d_i = depth\ of\ one\ possible\ hierarchy\ path\ from\ the\ class\ of\ interest\ to\ its\ root\ class. \end{array}}$$

(a) Implemented metric 1: Depth of Inheritance Tree as defined in Chidamber and Kemerer [13]

$$\boxed{\begin{array}{ll} LCOM = |P| - |Q| & if\ |P| > |Q| \\ \quad = 0 & otherwise \end{array}}$$

(b) Implemented metric 2: Lack of Cohesion in Methods as defined in Chidamber and Kemerer [13]

$$\boxed{McCabe\ cyclomatic\ complexity\ v(G) = number\ of\ decision\ statements + 1}$$

(c) Implemented metric 3: McCabe cyclomatic complexity as defined in McCabe and Butler [39] For example, decision statements can be "if", "switch", or "for"-statements.

$$\boxed{MLOC_i = \sum lines_{method_i} - \sum comment\ lines_{method_i} - \sum empty\ lines_{method_i}}$$

(d) Implemented metric 4: Method Lines of Code as defined in Gyimothy et al. [29]

Figure 4.12.: Definition of the implemented metrics

The "Persuasive Quality Improver" implements the DIT (see Figure 4.12(a)), Lack Of Cohesion between Methods (LOC alias LCOM) (see Figure 4.12(b)), and McCabe (see figure 4.12(c)) metrics. These metrics got a good evaluation in both mentioned works. Additionally, the metric Method Lines Of Code(MLOC) (see Figure 4.12(d)) was used, which is not mentioned in [55] or [13] but in [29].

4.7. "Persuasive Quality Improver" Implementation

The tool was built on the basis of Felfernig, Pribik, Steinparz, and Leitner [20] and an existing framework which provided the functionality of metrics calculation as an Eclipse plug-in. The framework realized basic plug-in functionality and an example implementation of metric evaluation. Further additional functions have been integrated in the Eclipse plug-in developed within the scope of this thesis:

- **Automated** quality check of all projects
- **Decorating** project tree entries depending on their metric's quality values
- Implementing the **metrics** defined in Section 4.6
- Providing **marker view** to the user, so he or she can easily access all the information provided
- Logging **evolution** of code quality to make it possible to evaluate the process of the quality evolution and the response of the user to the given feedback of the software metrics
- Creating a so called "Quality Guide" **tunneling dialog** which is needed to guide the programmer through all the improvement possibilities provided and give information as well as additional motivation

The plug-in structure and its elements are explained in detail in the following sections.

4.7.1. Eclipse Plug-in Basics

Every Eclipse plug-in is built on a basic structure. This structure contains a main class which is extended from AbstractUIplugin and a plugin.xml config file. The plugin.xml file contains so-called "entry points" for the plug-in. These entry points define all parts of the plug-in. For example, if the plug-in has to add a menu item to the Eclipse Environment, this menu item has to be declared in the config file as an entry point (see Listing 4.1). This is why the plugin.xml file is mentioned very often in this plug-in documentation. The second main part of each Eclipse Plug-in is the class which is extended from AbstractUIPlugin. This class usually has the name "Activator" and this is also the name it has in this project. In this class an id should be defined which identifies your plug-in. This id is built up like Java Package names (e.g. at.tugraz.masterthesis.Metricsplugin). To get your plug-in to start and stop working, two functions called *start()* and *stop()* are called by the Eclipse Environment. There is no other functionality needed here.
Because the plug-in needs to parse the source code, so called Nature and Builder classes are needed. A Nature can be connected to projects. In this plug-in it is needed to connect to all projects automatically. It was a little bit tricky to integrate the nature automatically into all user projects. The Eclipse Plug-in Environment doesn't have any ready-to-use functionality to do that. So this had to be done manually, as shown in Listing 4.2.

A Nature has an id and has to be defined in the plugin.xml where it becomes associated with the Builder needed. Now the defined Builder is called every time the developer builds his or her source code and we can evaluate the code while it gets built by Eclipse. The structure of a Builder class provides the possibility to react to different types of source code changes. Source code can either be "changed", "added" or "removed". In this plug-in only the cases "changed" and "added" are

Listing 4.1: Extract from the plugin.xml configuration file. This shows how the “Issuenary” menu item is added to the eclipse environment.

```
<extension
      point="org.eclipse.ui.commands">
   <command
         defaultHandler="at.tugraz.ist.myplugin.issuenary.DialogAction
            "
         id="at.tugraz.ist.myplugin.openIssuenaryDialog"
         name="openIssuenaryDialog">
   </command>
</extension>
<extension
      point="org.eclipse.ui.menus">
   <menuContribution
         locationURI="menu:org.eclipse.ui.main.menu">
      <menu
            id="qltyMenu"
            label="Quality">
         <command
               commandId="at.tugraz.ist.myplugin.openIssuenaryDialog"
               label="Issuenary"
               style="push"
               tooltip="open␣Issuenary␣Dialog">
         </command>
      </menu>
   </menuContribution>
</extension>
```

Listing 4.2: Code for adding the Nature class to all projects in the workspace

```
public static void addNatureToAllProjects()
 {
  try {
   IWorkspace workspace = ResourcesPlugin.getWorkspace();
   IWorkspaceRoot root = workspace.getRoot();
   // Get all projects in the workspace
   IProject[] projects = root.getProjects();
   for (IProject project : projects) {
    IProjectDescription description = project.getDescription();
    String[] natures = description.getNatureIds();
    boolean found = false;
    for (int i = 0; i < natures.length; ++i) {
     if (MypluginNature.NATURE_ID.equals(natures[i])) {
      //already there :)
      found = true;
     }
    }

    // Add the nature
    if(!found){
     String[] newNatures = new String[natures.length + 1];
     System.arraycopy(natures, 0, newNatures, 0, natures.length);
     newNatures[natures.length] = MypluginNature.NATURE_ID;
     description.setNatureIds(newNatures);
     project.setDescription(description, null);
    }
   }
  } catch (CoreException e) {
   e.printStackTrace();
  }
 }
```

needed and lead to a call of the metrics calculation (see Section 4.7.4). Markers (Section 4.7.3) and decorations are placed depending on the calculated metric values.

4.7.2. Decorator

The Decorator is an important part of this plug-in. A Decorator has the assignment to decorate items of the project tree. Decorating means that, in addition to the defined items, a small icon will be displayed. Each item (i.e., projects, packages, classes, methods) can be decorated . Eclipse uses decorating to show the user if the item contains warnings or errors. Another well-known use of the decorator is used by the SVN plug-in Subclipse. This plug-in shows if the item is up to date with SVN or contains changes. In the plug-in developed within the scope of this thesis, these icon decorations are one part of the provided feedback about code quality. The code quality of each item can be seen with these decorations. This makes it easier for the user to see where the most work for refactoring will be needed. An appropriate icon is added to the specific item after each build and metrics evaluation. This additional functionality causes the need of a Decorator class with the override function *decorate()*. This Decorator again has also been defined in the plugin.xml file.

4.7.3. Markers

Markers besides decorating can be used to give feedback to the user. Every developer knows the small icons on the left and right side of the source code window. They give information about syntax errors and various kinds of warnings. Plug-ins can use this technique too. A plug-in can define its own marker with a specific color, highlighting, and icon. In this plug-in, there are two different kinds of markers specified. One is visible for the developer and one is not. The marker type is chosen depending on the value of the different metrics. If the value is above or beyond its specific threshold, the visible marker is chosen. The second one is not visible to the developer and only used for further evaluations. Eclipse provides default marker types. To create an individual marker type it has to be defined in the plugin.xml file. For one marker, three different entries have to be added to the config file. The first one is the marker itself. It contains only a name and an id. The second entry is about the annotation type (e.g., “Error” or “Warning”) that is needed in the third entry which is called marker annotation specification. In this last entry, all the visible attributes mentioned above can be defined. The visible marker has the following attributes: an icon, the text decoration style “dashed box”, one of the predefined link icons (this icon is shown in the metric view) and a red color for the text decoration. This combination leads to the visualization shown in Figure 4.7. A marker type is defined with the steps described. The next step is to create a marker from the plug-in, which is shown by Listening 4.3

Listing 4.3: Attach a marker to source file

```
IFile _file = currentfile;
IMarker marker = _file.createMarker(MARKER_TYPE);
```

4.7.4. Metrics

The metrics part is more complex than the parts described earlier. There are several differences in the calculation of the metrics. Every metric needs different information for its calculation and even the hierarchy level of the data has to be different. This means there are metrics which have to be calculated on the class level and others that have to be calculated on the method level. Nevertheless, this part has to be very adaptable because it is important to be able to modify or add metrics for the usability of the "Persuasive Quality Improver". This leads to a larger class structure. Here, a short overview of the classes which are involved in the metrics calculation is given:

- The *metrics classes* itself
- The base class for each metric class: *MetricsCalculator*
- *MethodVisitor and FieldVisitor*: These classes are implemented as private classes in the metric classes which need them (e.g., LOC)
- *MetricTypeResult* is a Collection of all MetricResult for one specific metric
- *MetricsJavaElement* which contains the IJavaElement which is provided by the Eclipse Plug-in Framework and the related *MetricTypeResult*
- *SourceEvaluator*: This class is called for each source file and gets the MetricsJavaElement Object to create the necessary Eclipse markers
- *MetricBuilder*: This is the builder of the plug-in. This builder has the assignment to call the SourceEvaluator with each source file which gets built

4.7.5. Dialog

There is only one dialog in this plug-in. Only the quality guide needs to be a dialog to interact properly with the developer. This dialog consists of two parts. The first part is the dialog itself. For this, a new class was extended from the StatusDialog which is provided by the Eclipse Plug-in Framework. The dialog itself is an ordinary Java Composite. This means there is a basic Composite object to which all visible elements have to be added. The second part is a so called Action. This action is needed to show the dialog to the user. Its only responsibility is to open the dialog needed when it's called. The time of activation of the dialog has to be defined in the plugin.xml configuration file.

4.7.6. Database and Logging

One important feature which is not visible to the developer is the logging mechanism. Every time the developer builds his or her source code and the metrics are calculated, the whole evaluation is saved in a SQLite database. This feature was needed for the evaluation of the case study in the next chapter and can be used for further studies or evaluations. With the collected data it is possible to see the evaluation of the quality over time. The SQLite database was chosen because of the given requirements. The complete database has to be delivered with the plug-in itself and it cannot be guaranteed that an internet connection is available. Additionally, the configuration of the plug-in is based on this database.

4.8. Remarks for Functionality Extension

This section is meant as a technical collection of possible extensions of the “Persuasive Quality Improver”. Each idea is described from a technical point of view and presents one part of the open structure of the, so that a developer can start easily with modifications.

The architecture used was designed to allow functionality extension. During the implementation process many additional ideas for persuasive concept implementation arose, which have not been applied. Some of the ideas where just out of scope of this thesis. The following sections present these ideas together with hints on how to implement them in the existing architecture.

4.8.1. Add more Metrics

This is the most obvious and likely functionality extension. To implement a new metric, the following steps are needed:

- Create a new class which extends from *MetricsCalculator*
- Override the function *calculate()* in the new class
- Create a MetricResult which includes all the necessary information for an Eclipse marker in the *calculate()* function
- Add information of the new metric to the IMetricsCalculator
- Add the new metric to the metrics list in MetricsCalculator *getMethodCalculators()*
- Add a new entry in the database. The database initialization can be done with the *DataBase.initializeMetricTable* function

When all these steps have been undertaken, the plug-in takes the new metric into account, will create markers, and decorate the project tree.

4.8.2. Suggest refactoring-tips to the user at the right time

It is difficult to get a good time when it comes to interrupting a programmer. If the programmer is currently deeply involved in the design process of an algorithm, it is highly destructive to interrupt him or her. One of the following could be an appropriate moment:

- The programmer tries to debug the program, but does not find the bug
- The programmer tries to get an overview of foreign code
- The programmer is already in a refactoring process
- The programmer introduces a new function to a class
- The programmer has not started programming yet or has taken a break

This functionality extension mostly concerns detecting what a programmer is doing just in time. With this knowledge the plug-in can decide whether it is a good moment to interrupt the programmer with a good advice or not. If a programmer is interrupted at an inappropriate moment, the tool may be ignored.
The implementation of this functionality could start in the Builder-class. Every time the programmer builds the program, the builder could log the changes. From this information the plug-in could guess the current intention of the user. For example, the user works on the same function or functions which call each other for a long time, but not much of the code is changing. This could be an indicator of debugging. In the *MetricBuilder.visit()* method a logger can be installed. Additionally, an evaluation unit will be needed, which evaluates the log during each build and raises actions when an appropriate moment is detected.

4.8.3. Improved Self-Monitoring

Currently, self-monitoring is limited to real time. The user cannot see a history of the code's quality evaluation. Maybe the user would be more motivated in using the plug-in when it's obvious that the code quality has really improved over the time. Additionally, the programmer can learn about his or her working practices. For example, in the morning a lot of new code is written and software quality goes down, while in the afternoon the software quality is improved again because the new code is refactored. The technical requirements are all satisfied. All the information concerning metrics has already been collected by the implemented Database Logger. The logging has to be adapted only

if additional information is needed. Currently, all the markers are stored in the database within the *SourceEvaluator.createMetricsMarkers()* function. If this is not needed, only the visualization has to be integrated.
Some ideas for possible visualizations are the following:

- Today's quality evolution of the current class or method
- Overall quality evolution of the current class or method. (see Figure 4.13(a))
- Most improved classes or methods of a certain interval (day/week/month). (see Figure 4.13(b))
- Evolution of a quality based class-ranking including the absolute quality value of each class (based on metric-evaluation).
- Overview on which classes the programmer spent the most time over a certain period (day/week/-month). Here, additional information has to be logged. (e.g., active time a user spent with a file) (see Figure 4.14(a))
- Overview which classes the programmer actually worked on today/this week/this month

4.8.4. Introduce a social factor

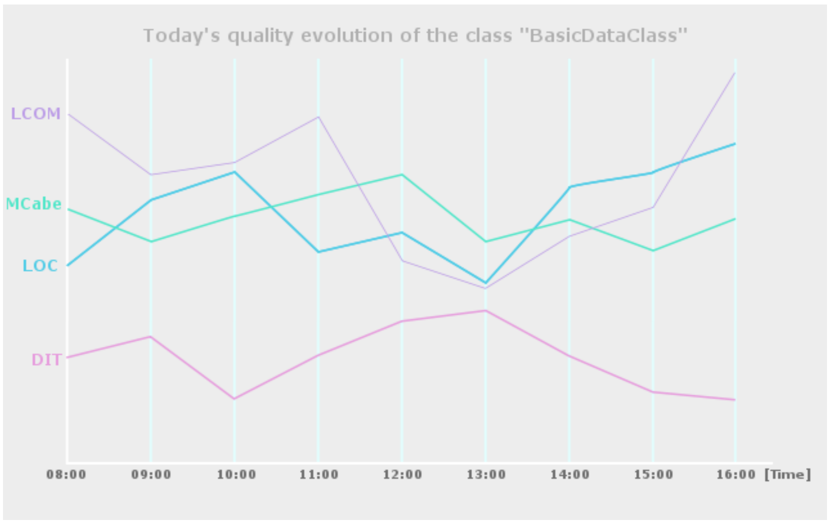
In the current prototype implementation, no social aspects have been taken into account, although these are one of the most persuasive elements (This aspect is also an issue of future work. See Chapter 6. There are many possibilities to take social aspects into account.

Show community analysis

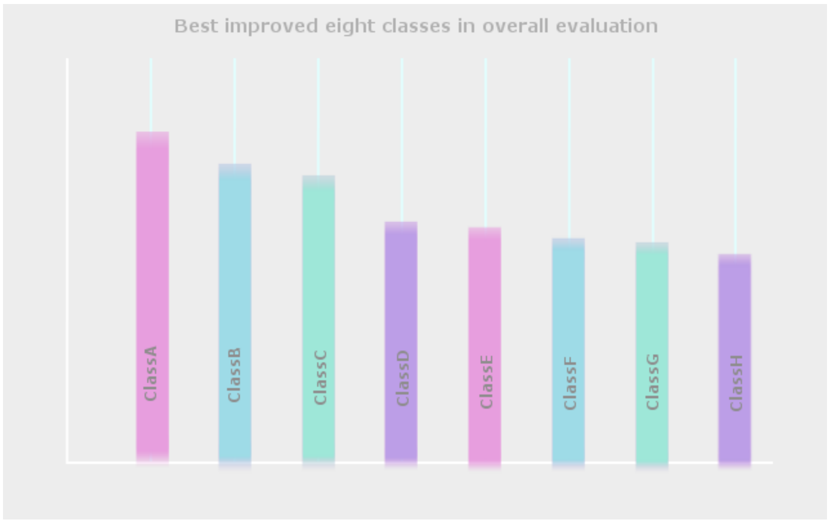
Give the programmer the possibility to compare him- or herself to his or her colleagues. This can lead to higher motivation for two reasons: On the one hand, there is a competition-factor and, on the other hand, it is clear that there are also other programmers who have to struggle with software quality.

- A list of the most alerted metrics of a certain interval (day/week/month)
- A list of the most alerted metrics overall
- A list of the highest and average combined values of all metrics of a certain interval (day/week/month)
- A list of the highest and average combined values of all metrics

The features mentioned above need an adjustment concerning the current database. The database which is currently used is directly located at the computer of the programmer. This means there are

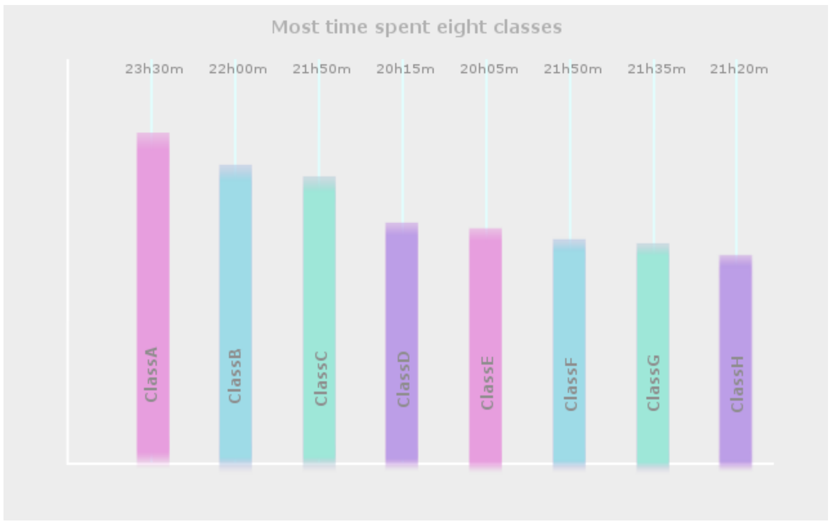


(a) Possible visualization of quality evolution of a class

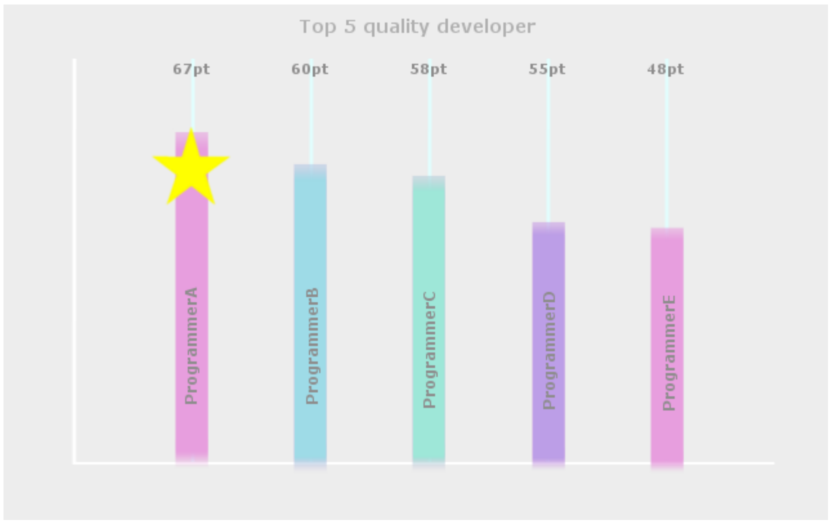


(b) Possible visualization of most improved classes

Figure 4.13.: Part 1: Visualization of further ideas



(a) Possible visualization of most time intensive classes



(b) Possible visualization of praise system

Figure 4.14.: Part 2: Visualization of further ideas

no values available from other programmers. A possibility to get information from all programmers is to publish the locally saved data to a global server. This could happen when the programmer is shutting down the Eclipse Environment to keep the building process short. This can be realized in the Activator class. This class contains the stop()-function which is called at the end of the plug-in life cycle, which normally means that the complete Eclipse Environment is shut down. Therefore, a global server would be needed, as well as a concept how to save data. Because of data privacy, no relation to the author of the code should be saved. A constructed id for classes, their methods and the calculated metrics should be enough information for such an analysis. In addition to the external database, a platform where the analysis can be displayed is needed. This could be a website or web services which are then again included in the Eclipse Environment to be directly shown to the programmer.

Qualitative evaluation

Metrics can only calculate values based on a predefined function. Often, there are circumstances which need code that has a low quality according to metrics. For example, code that has to run on machines with low call stack. Here the code has to be written with as few methods as possible, which will lead to a high LOC value. Or, for example, code which contains very complex algorithms which cannot be changed because of runtime or memory management. This code will lead to a high McCabe value. The metrics are not able to take into account special challenges a program has to cope but another programmer would be able to. For example, the runtime of a program has to be short. This may lead to a complex code and a poor metrics evaluation. The programmer may be able to give a more professional opinion onto the complexity than a metric. Another advantage of the review of another programmer is that metrics cannot give context-adapted tips for possible improvements.

- Let programmers evaluate code fragments from other programmers
- Let programmers give tips on how to improve a specific source code which has bad quality according to metrics

As discussed in the previous section, the first problem concerns the global database. Here again, global data from all the programmers is needed. Additionally, the code is needed for the review. The original programmer also should be able to see the given evaluation and tips such that it is not possible to save all the data completely anonymously. This may lead to data privacy issues which have to be addressed. A solution could be that the programmer can decide which specific source code will be released for evaluation. The process of ”submitting” source code to an evaluation process can also be enriched with further functionality. During the submit process the programmer could be asked to add documentation to the piece of code. Also specific questions for improvements or concerning problems could be added by the programmer. The more information the programmer adds to the

submitted source code, the better another programmer can evaluate and help.
The process of requesting feedback will start with an associated menu-entry. This entry could be in the context menu or in its own menu entry at the Eclipse main menu. These entries have to be defined in the plugin.xml and need an Activator like the current quality dialog button. From such an Activator, the information has to be transmitted to the public database. Here again a platform is needed where the code will be evaluated and commented. A big challenge will be the visualization of the given evaluation for the programmer. Because of the poor visualization possibilities of Eclipse, this should be realized on an external platform. Eclipse provides a possibility to integrate websites. With this functionality it is possible to integrate an external web-based platform into Eclipse.

Competition

Many people like to compete and try to be the best. There is the possibility to compete in code improvement. This kind of social factor can be activated temporarily, maybe only in a refactoring phase where no one should implement new functionality but improve the existing source code. This phase is often disliked by programmers, because there is nothing new in it. The motivation of refactoring can be raised when it is combined with a competition. The possible subjects of a competition could be:

- Improved software quality: The calculated metric values end up in a numeric value of negative points. The higher these points are, the worse the software quality is. When the competition starts, each programmer has to reduce his or her points. The programmers who reduce the most points will be listed in a high score.
- Accepted tips and given feedback: Combined with the qualitative evaluation functionality, programmers can accept given improvement tips and feedback. For every accepted feedback or tip the reviewer get points.
- Overall software quality score: This calculation is very easy but the result will be slightly problematic. Depending on the context in which the programmer is working, it would be more easy or difficult to achieve good metric values.

Depending on the focus of the competition, this could even be calculated without any change in the current plug-in. Only the corresponding analysis has to be implemented.

4.8.5. Praise and Reward

Currently the user doesn't get any rewards from the plug-in besides, perhaps, a higher quality of code. A high quality of code, of course, gives the programmer advantages when it comes to maintenance or functional enrichment. This will not be obvious in the starting phase but in the long-term. To get higher motivation from start up, a reward system could be introduced. The problem here is the subject of the reward. The current plug-in does not have much to offer that can be measured and rewarded. Only the metric values are available from the start. It could be difficult to reward the increase in software quality. At some point valuable improvement is no longer possible and different programmers will have different opinions concerning the complexity of code. Besides the metrics evaluation the usage intensity of the plug-in could be evaluated. There are many possibilities for a reward system. For example, a blackboard where the top quality coders are published (see Figure 4.14(b)) or even a coder of the month meeting.

4.8.6. Increase the System Credibility

The metrics are predefined and the programmers cannot change the metrics or even the thresholds. Programmers may have negative experiences with metrics. If this is the case, the acceptance of a metrics tool will be very low because the programmers won't trust in its expertise. The system credibility has to be take seriously. The only information provided by the plug-in is the short descriptions given in the Eclipse Environment concerning the intent of metrics. The system credibility will benefit from an additional knowledge base concerning metrics and their advantages. Based on [43] and the given principles for system credibility support, such a knowledge base should at least include following details:

- Metric name
- The intentions when using the metric
- References to other sites or research
- Name(s) of person(s) in the organization who decided on the use of this metric
- Simple description including advantages and disadvantages of this metric

Chapter 5

Case Study

The case study conducted and its results will be presented in this chapter. The goal of this study was to determine if programs developed with the plug-in presented have a better quality than programs developed in the standard Eclipse Environment. This case study consisted of twenty informatics students as programmers, an assignment, an automated evaluation, and four code reviewers.

5.1. Participants

The same requirements were defined for the programmers and the code reviewers. Each participant had to have experience in software development, especially in the JAVA programming language, and with the Eclipse IDE. All the participants had education in software development. Three participants were female and 21 were male. The 20 programming participants were divided in two groups. Each group had to implement the same assignment (see Section 5.2). The difference between these groups was the IDE provided. One group had to develop in the standard Eclipse IDE and the other in the modified Eclipse IDE presented in Chapter 4. The reviewers had to review the code developed by the twenty programmers. Each reviewer got five code projects. A reviewer did not know whether the given source code was developed by a programmer out of the group with the standard Eclipse IDE or the modified one.

5.2. Chosen Assignment

The programmer's assignment was to develop a parser. This parser should be able to read CSV and XML files in a predefined format and independently write the data in a predefined format to the console or CSV or XML file. The test data was about the 2010 FIFA World Cup. The predefined format can be seen in Listenings 5.1, 5.2, and 5.3.

Listing 5.1: The assignment's predefined CSV format

```
START;NATION1;GRUPPE1;TORE1;NATION2;GRUPPE2;TORE2
11.06.2010 16:00;Suedafrika;A;1;Mexiko;A;1
11.06.2010 20:30;Uruguay;A;0;Frankreich;A;0
```

Listing 5.2: The assignment's predefined XML format

```
<?xml version="1.0" ?>
<WM>
<match start="11/06␣16:00">
  <nation tore="1" gruppe="A">Suedafrika</nation>
  <nation tore="1" gruppe="A">Mexiko</nation>
</match>
<match start="11/06␣20:30">
  <nation tore="0" gruppe="A">Uruguay</nation>
  <nation tore="0" gruppe="A">Frankreich</nation>
</match>
</WM>
```

Listing 5.3: The assignment's predefined text output format

```
Spiel vom 11/06 16:00: Suedafrika gegen Mexiko spielten 1:1
Spiel vom 11/06 20:30: Uruguay gegen Frankreich spielten 0:0
```

The programmer's assignment's framework was:

- *"Write your program in about four hours"*
- *"Internet access is available"*
- *"Write the program in the provided IDE in the JAVA programming language"*
- *"Use the provided test-files to prove the correctness of your program"*
- *"The written programm will be extended by another programmer. Choose your program structure accordingly"*

This assignment gave many possibilities on how to implement and structure the program. The different solutions provided by the programmers approved the choice of this assignment.

5.3. Automated Evaluation

The first part of the evaluation was done automatically. This evaluation is based on the final evaluation of each of the twenty developed programs calculated by the Eclipse Plugin. This evaluation provided information about the persuasiveness of the system. Have the users been willing to improve the source code the plug-in gave feedback on? This evaluation should give an insight on whether people working with the plug-in, and getting feedback on their source code were actually responding to the feedback and, therefore, had different results to people working without the plug-in. The concrete values can be seen in Tables 5.1, 5.2, 5.3, and 5.4. The overall value represents how often the specific metric has been calculated. The value "above threshold" represents how many times the calculated value was above the defined threshold. The failure rate describes the relation between the number of values and the values above the threshold ($\frac{[\#abovethreshold]}{[\#allvalues]}$). The results of the LCOM and McCabe of source code metrics written with the developed plug-in are significantly different from those of source code written without the plug-in. This means the source code improved significantly regarding these metrics with the use of the plug-in. The metric DIT also has different values depending on the use of the plug-in, but the number of failures is very low in both situations. This is why these values are not considered as significant. The fourth metric LOC seems to be independent from the use of the plug-in. This is a good result, but a question about the cohesion metric is raised: Why didn't the cohesion improve with the plug-in? One possibility why the programmers did not improve their programs concerning cohesion is that it is quite difficult to do so. Improving the cohesion of a given source code is much more difficult than, e.g., reducing the lines of code. Even if the plug-ins alerts immediately when the threshold is reached, the programmer already has a concrete concept of the class-structure in mind. So the alert could have been ignored. Another possibility could be the message from the plug-in. A description of the metric and improvement suggestions are given for each metric. The improvement suggestion given is:
Decide which kind of functionality should be in here and move all other functions to another class. Maybe you have to introduce new classes.
Compared to the suggestions of the other metrics, this one is quite unspecific (e.g., McCabe suggestions: *Try to resolve nested IF-Statements and introduce a CASE-Statement, Try to simplify the algorithm*). A completely different approach to explain why the cohesion was not significantly different is that the algorithm was not granular enough to detect differences between the small programs. Half of the values are between -4 [*] and 2. Overall, these results show that the plug-in did persuade the users. There are significant improvements in the code developed using the plug-in.

[*]Values beyond 0 should be set to 0 according to [13]. Because in this plug-in only one threshold is used it doesn't matter if values below 0 are set to 0 or not.

LOCM	Above threshold	Overall	Failure rate	χ^2
Standard Eclipse IDE	73	314	23.35%	
Extended Eclipse IDE	40	300	11.76%	
				< 0.001

Table 5.1.: Lines of code per method - evaluated results

DIT	Above threshold	Overall	Failure rate	χ^2
Standard Eclipse IDE	3	99	2.94%	
Extended Eclipse IDE	2	102	1.92%	
				0.63

Table 5.2.: Depth of Inheritance Tree - evaluated results

LOC	Above threshold	Overall	Failure rate	χ^2
Standard Eclipse IDE	26	102	25.49%	
Extended Eclipse IDE	27	104	25.96%	
				9.4

Table 5.3.: Lack of Cohesion in Methods - evaluated results

McCabe	Above threshold	Overall	Failure rate	χ^2
Standard Eclipse IDE	38	314	12.10%	
Extended Eclipse IDE	10	340	2.94%	
				< 0.001

Table 5.4.: McCabe - evaluated results

5.4. Code Review

The second evaluation which was accomplished was carried out by the reviewer group of participants. These participants received a predefined evaluation sheet which they could use to give points from one to ten on questions on different topics. This evaluation gives, besides the very well analyzed field of metrics (see chapter 2), an insight into the question of whether a programmer really thinks, that "good" source code, regarding metrics, actually is a "good" source code. Therefore, this evaluation should give information about the subjective interpretation of quality of the code beyond metrics.
To give an impression of the subjective evaluation, the questionnaire is provided in Table 5.5 [†]. The questionnaire is divided into four sections. Each item can be rated from one to ten, where ten means "perfect job" and one "not or very poorly done" .

The first category of the questionnaire was meant to get the reviewer to look through the code and get a first impression before starting to evaluate more specific topics. The next three parts give feedback according to "Complexity", "Customizability", and "Structure". The last question gives the reviewer a possibility to rate the overall quality. A chi-square significance test was made for each of the 14 questions. This evaluation gives the possibility to distinguish between quality factors which are connected to the metrics and are, therefore, supported by the plug-in, and quality factors which are not. Here is a mapping between the metrics and items used in the questionnaire:

- McCabe
  - (E) Complexity of Functions
- Lack Of Cohesion in Methods (LCOM)
  - (K) Class Cohesion
  - (D) Complexity of Classes
- Method Lines of Code (MLOC)
  - (D) Complexity of Classes
  - (E) Complexity of Functions
- Depth of Inheritance Tree
  - (D) Complexity of Classes
  - (F) Locating Errors will be easy
  - (M) Good Inheritance

[†]This questionnaire was conducted in German.

- **Program Overview**
  - (A) Purpose of complete program is obvious: *How long does an external reviewer need until he or she understands what the program's purpose is?*
  - (B) Purpose of classes is obvious: *How long does an external reviewer need on average until he or she understands what a class does?*
  - (C) I know where to look for specific functionality: *Does the program structure follow any logic?*
- **Complexity**
  - (D) Complexity of classes: *Are the classes more complex than needed?*
  - (E) Complexity of functions: *Are the functions more complex than needed?*
  - (F) Locating errors: *Are functions' purposes obvious and will an error causing function be easy to locate?*
- **Customizability**
  - Extend Functionality
    - (G) # of classes which have to be changed: *Does the program structure support adoption?*
    - (H) Expected time exposure: *Are the inter class dependencies very complex?*
  - Use functionality in other programs
    - (I) # of classes which have to be changed: *Does the program structure support reuse?*
    - (J) Expected time exposure: *Are the inter class dependencies hard to lose?*
- **Structure**
  - (K) Class cohesion: *Is a high class cohesion given?*
  - (L) Balanced functionality between classes: *Are the program functionalities divided between different classes fairly?*
  - (M) Good inheritance: *Is inheritance used and is it used in a reasonable way?*
  - (N) Good interface design: *Are any interfaces used and are they used in a reasonable way?*
- **Overall Impression**:
  - (O) Give a subjective overall feeling

Table 5.5.: Code Review Questionnaire

The results of the χ^2 test are shown in Table 5.6. The questions (C), (D), (E), (F), (G), (H), (I), (J), (K), (L), and (M) were answered significantly differently. It is interesting that the class cohesion (question (K)) was significantly better in the plug-in assisted group than in the other one, especially in connection with the cohesion metric evaluation result in Section 5.3. This could mean that the programmer's idea of cohesion does not overlap with the metric's definition. This is an indicator that, contrary to the assumption in Section 5.3, the programmers did try and achieve to improve the program's cohesion. The questions (A), (B), (C), (J), (L), (M), (N), and (O) did not show a significant

Question	χ^2
(A)	0.20
(B)	0.23
(C)	0.21
(D)	**0.03**
(E)	**0.05**
(F)	**0.04**
(G)	**0.09**
(H)	**0.08**
(I)	**0.06**
(J)	0.15
(K)	**0.05**
(L)	0.25
(M)	0.12
(N)	0.12
(O)	0.10

Table 5.6.: χ^2 test results of questionnaire

difference. (A), (B), and (C) are very general first impression questions and therefore do not degrade the positive results of this evaluation. Code quality is not necessarily obvious. Otherwise there would not be so much interest in metrics. A similar argument can be used for question (O). Even if the subjective overall feeling did not achieve a significantly different result, the other results show objective differences. The result of question (N) has to be discussed in more detail. The question (N) relates to the use of interfaces. Concerning to it's results the quality of the use of interfaces does not distinguish significantly between source code written with or without the "Persuasive Quality Improver" plug-in. The implemented metrics did not refer to the use of interfaces. This means the plug-in did not give any feedback according to this code quality. This leads to the conclusion, that quality factors, which are not supported by the plug-in are not improved either. The same argumentation and conclusion is valid for question (L). There is no metric which measures balanced functionality between classes. There are two questions left: (J), and (M). Question (M) is related to the metric DIT. This metric was also not evaluated well in Section 5.3. The questions (G), (H), (I), (J) also have interesting results.

These four questions apply to the group of the customizability quality. This quality factor was not measured by the implemented metrics. Nevertheless, three out of these four questions were evaluated as significant. This is an indicator that customizability is effected by structure and complexity quality factors. This dependency is not surprisingly at all. A well structured program with low complexity is more easy customize, than a highly complicated one with poor structure. Customizability can be seen as a part of maintainability. Further research concerning maintainability is discussed in Chapter 2.

The programmers did respond to all the metrics beside DIT. This is positive feedback for the developed plug-in.

5.5. Feedback

A lot of feedback was collected from the participants in this case study. On the one hand, there is feedback from the programmer who used the plug-in and, on the other hand, there is feedback from the review. Both kinds of feedback can be used to improve the plug-in. The comfort of using the plug-in can be improved by taking into account the programmer's feedback and the metrics the plug-in uses can be adapted concerning the feedback of the reviewers. Here are some examples of the feedback given by programmers (translated from German):

- *"I liked the smiles-overlay-icons in the project-tree view"*
- *"I did not recognize the icon for the additional quality tool as a button and therefore didn't use it"*
- *"I think the coherency metrics help was great. But I really would have liked to define functions as debug-functions to suppress the warning "*
- *"The smiles were too unspecific"*
- *"I liked that the tool is warning the programmer at the beginning of a problem and inspires to start refactoring earlier"*
- *"I didn't need the extra quality tool because the flags in the code were clear enough"*
- *"The metric tips were too annoying"*
- *"The build process took too long because of the calculating metrics"*

Overall, the programmers accepted the plug-in very well, although there is a high potential for improvement. The most criticized issue was the code marking which was seen as too flashy.
The feedback of the reviewers was mostly about what they liked or disliked in the reviewed code

design.
An extract of the given positive feedback:

- *“Correct use of interfaces”*
- *“Many functions are documented”*
- *“Quite good structure. It’s obvious where to look for implementations and possible errors. ”*
- *“Good naming”*

An extract of the negative feedback given:

- *“Unclear formatting and too many lines of code in functions of one class”*
- *“Too many classes - although the purpose of each is obvious, a few of them are unnecessary”*
- *“Bad naming and therefore very difficult to understand the code. No principal in package definition noticeable. A lot of structural misfitting.”*
- *“Only one big class which handles everything. Bad overall design which cannot be used in other projects.”*
- *“Bad naming of methods and no documentation of interfaces.”*
- *“Violates coding standards”*

This feedback gives the possibility to expose what programmers like if they have to maintain a foreign program and should be used for future work. This qualitative feedback was not taken into account in the evaluation in Section 5.4.

Chapter 6

Future Work

There are many possibilities for future work. Additional functionality can be implemented and analyzed in user studies. Some ideas of additional functionality based on the thoughts of persuasive technology are already given in Section 4.8. Further functionality ideas can be found in this chapter. Other possibilities for ongoing work are technical improvements of the implemented prototype. The qualitative feedback stated in Chapter 5.5 contains ideas for further functionality.

6.1. Implementing the Feedback

Additional functionality was stated by various attendees in the feedback collected from the user-study.
Define Debug-Methods or Classes:
A few programmers want the possibility to define methods or classes as debug classes. This means that these classes will not be evaluated. The programmer should have the possibility to define specific methods as "OK", even if the metrics don't agree. To provide this functionality the process of metrics evaluation must be adapted. This process starts in the *SourceEvaluator* class. In this class, the *calcuateMetricsForResource()* method is located. The built resource is passed to this function which then calls the *MetricsJavaElement.calculateMethodMetrics()* function. This function iterates through all metric implementations and starts the metrics calculation.
This function can lead to a completely useless plug-in. If it is too easy for the programmer to define methods or classes for debugging reasons so that they don't get bad quality evaluations, it is possible that no non-debugging methods will be left for evaluation. Here are some ideas to prevent this situation:

- Define mandatory comment attributes

- Introduce a metric which calculates the balance between normal and debugging code
- Introduce a metric which checks if debugging methods are used by non-debugging ones

6.2. Technical Improvements

The main focus of the existing prototype was on the functionality needed and user interaction. The prototype does exactly what it should do, but maybe there is potential for improvement. One part that could need improvement is the build process and metrics calculation speed. In the case study, the feedback was given that the build process takes a very long time. If the plug-in were to be used in a real-life project this would quickly lead to its dismissal. Improving the build process should be considered. The reason why it takes to long can be found in the metrics calculation. During the metrics calculation every changed method has to be reviewed for each metric. This takes too much time.

6.3. Offer More Information

Currently the plug-in consists of only three different parts that provide the user with information: the markers, decorations and the quality dialog. A new visible item could be a self-defined marker view. Such a view could be designed in a way that gives the user a better overview and more information.

6.4. More Detailed Improvement Suggestions

The improvement suggestions provided in the quality guide and the markers received positive feedback and were accepted very well. The way information is provided influences the reader [36]. More concrete and detailed suggestions could lead to better metric results in less time. The programmer would know exactly how to improve the quality.

Chapter 7

Conclusion

In this thesis, metrics and persuasive technology were discussed and combined. The comprehensive field of metrics has been discussed and summarized in Chapter 2. These results show us how to use metrics. The collected knowledge was then used to integrate persuasive technologies into software development processes in Chapter 4. This is a new field of application for persuasive technology. An eclipse plug-in was developed to investigate whether persuasive approaches are successful. The developed plug-in was used to conduct an empirical study. The results of this study clearly show the applicability of the developed approaches (see Chapter 5).

Bibliography

[1] AIAA AND ANSI. 1993. Recommended practice for software reliability. *R-013-1992*. (Cited on page 22.)

[2] ALSHAYEB, M. AND LI, W. 2003. An empirical validation of object-oriented metrics in two different iterative software processes. *Software Engineering, IEEE Transactions on 29,* 11, 1043 – 1049. (Cited on page 26.)

[3] ATKINSON, B. 2006. Captology: A critical review. In *Persuasive Technology*. Lecture Notes in Computer Science, vol. 3962. Springer Berlin / Heidelberg, 171–182. (Cited on page 40.)

[4] BANDI, R., VAISHNAVI, V., AND TURK, D. 2003. Predicting maintenance performance using object-oriented design complexity metrics. *Software Engineering, IEEE Transactions on 29,* 1, 77 – 87. (Cited on page 26.)

[5] BASILI, V. R., BRIAND, L. C., AND MELO, W. L. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering 22*, 751–761. (Cited on pages xi, 18, 19, and 49.)

[6] BERDICHEVSKY, D. AND NEUENSCHWANDER, E. 1999. Toward an ethics of persuasive technology. *Commun. ACM 42*, 51–58. (Cited on pages xii and 39.)

[7] BOEHM, B. W., BROWN, J. R., AND LIPOW, M. 1976. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 592–605. (Cited on pages xi, 7, 8, 9, 11, 12, 19, 20, 21, and 23.)

[8] BOOCH, G. 1986. Object-oriented development. *IEEE Trans. Software Eng. 12,* 2, 211–221. (Cited on pages xi, 13, and 14.)

[9] BRIAND, L. C., WÜST, J., IKONOMOVSKI, S. V., AND LOUNIS, H. 1999. Investigating quality factors in object-oriented designs: an industrial case study. In *ICSE '99: Proceedings of the 21st*

international conference on Software engineering. ACM, New York, NY, USA, 345–354. (Cited on page 23.)

[10] CHATLEY, R. AND TIMBUL, T. 2005. Kenyaeclipse: learning to program in eclipse. *SIGSOFT Softw. Eng. Notes 30,* 5, 245–248. (Cited on pages xi, 3, and 5.)

[11] CHIDAMBER, S. R., DARCY, D. P., AND KEMERER, C. F. 1998. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw. Eng. 24*, 629–639. (Cited on pages 19, 22, 24, and 26.)

[12] CHIDAMBER, S. R. AND KEMERER, C. F. 1991. Towards a metrics suite for object oriented design. *SIGPLAN Not. 26,* 11, 197–211. (Cited on pages xi, 13, and 14.)

[13] CHIDAMBER, S. R. AND KEMERER, C. F. 1994. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng. 20,* 6, 476–493. (Cited on pages xi, xii, 14, 17, 18, 24, 49, 50, and 67.)

[14] DANDASHI, F. 2002. A method for assessing the reusability of object-oriented code using a validated set of automated measurements. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*. ACM, New York, NY, USA, 997–1003. (Cited on pages 19 and 21.)

[15] DAVIS, J. 2009. Design methods for ethical persuasive computing. In *Proceedings of the 4th International Conference on Persuasive Technology*. Persuasive '09. ACM, New York, NY, USA, 6:1–6:8. (Cited on page 40.)

[16] DEMARCO, T. 1982. *Controlling software projects: management, measurement & estimation*. Yourdon Press, New York, NY :. (Cited on page 7.)

[17] EL EMAM, K., BENLARBI, S., GOEL, N., AND RAI, S. N. 2001. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng. 27*, 630–650. (Cited on page 26.)

[18] FAIRCHILD, R., BRAKE, J., THORPE, N., BIRRELL, S., YOUNG, M., FELSTEAD, T., AND FOWKES, M. 2009a. Unconscious persuasion by ambient persuasive technology: Evidence for the effectivity of subliminal feedback. In *Proceedings of the AISB Convention*. (Cited on page 38.)

[19] FAIRCHILD, R., BRAKE, J., THORPE, N., BIRRELL, S., YOUNG, M., FELSTEAD, T., AND FOWKES, M. 2009b. Using on-board driver feedback systems to encourage safe, ecological and efficient driving: the foot-lite project. In *Proceedings of the Persuasive Technology and Digital Behaviour Intervention Symposium*. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour, 28–31. (Cited on pages 32 and 38.)

[20] FELFERNIG, A., PRIBIK, I., STEINPARZ, S., AND LEITNER, G. Towards persuasive technologies for improved software quality. *UMAP 2011 Workshop on User Models for Motivational Systems*. (Cited on pages 41, 46, and 50.)

[21] FINKELSTEIN, L. AND LEANING, M. 1984. A review of the fundamental concept of measurement. In *Measurement vol. 2*. 25–34. (Cited on page 23.)

[22] FOGG, B. 1997. Captology: the study of computers as persuasive technologies. In *CHI '97 extended abstracts on Human factors in computing systems: looking to the future*. CHI EA '97. ACM, New York, NY, USA, 129–129. (Cited on pages xii and 36.)

[23] FOGG, B. 2002. *Persuasive Technology: Using Computers to Change What We Think and Do (Interactive Technologies)*, 1 ed. Morgan Kaufmann. (Cited on pages 46, 47, and 48.)

[24] FOGG, B. 2009a. A behavior model for persuasive design. In *Proceedings of the 4th International Conference on Persuasive Technology*. Persuasive '09. ACM, New York, NY, USA, 40:1–40:7. (Cited on pages 29 and 30.)

[25] FOGG, B. J. 2003a. Motivating, influencing, and persuading users. In *The human-computer interaction handbook*, J. A. Jacko and A. Sears, Eds. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, Chapter Motivating, Influencing, and Persuading Users, 358–370. (Cited on pages 29, 32, and 33.)

[26] FOGG, B. J. 2003b. *Persuasive Technology: Using Computers to Change What We Think and Do*. Morgan Kaufmann, San Francisco, California. (Cited on pages 32, 36, 39, and 40.)

[27] FOGG, B. J. 2009b. Creating persuasive technologies: an eight-step design process. In *Proceedings of the 4th International Conference on Persuasive Technology*. Persuasive '09. ACM, New York, NY, USA, 1–6. (Cited on page 45.)

[28] FOGG, B. J. AND HREHA, J. 2010. Behavior wizard: A method for matching target behaviors with solutions. *Persuasive Technology 6137*, 117–131. (Cited on pages xi and 31.)

[29] GYIMOTHY, T., FERENC, R., AND SIKET, I. 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng. 31*, 897–910. (Cited on pages 19 and 50.)

[30] IEEE. 1998. Standard for a software quality metrics methodology. *1061-1998*. (Cited on page 22.)

[31] INTILLE, S. 2004. A new research challenge: persuasive technology to motivate healthy aging. *Information Technology in Biomedicine, IEEE Transactions on 8,* 3 (sept.), 235 –237. (Cited on page 30.)

[32] KHALED, R., BARR, P., NOBLE, J., AND BIDDLE, R. 2006. Investigating social software as persuasive technology. In *Persuasive Technology*. Lecture Notes in Computer Science, vol. 3962. Springer Berlin / Heidelberg, 104–107. (Cited on page 36.)

[33] KING, P. AND TESTER, J. 1999. The landscape of persuasive technologies. *Commun. ACM 42*, 31–38. (Cited on page 37.)

[34] LEUNG, H. K. N. 2001. Quality metrics for intranet applications. *Information & Management 38,* 3, 137 – 152. (Cited on pages xi, 15, and 17.)

[35] LI, W. AND HENRY, S. 1993. Object-oriented metrics that predict maintainability. *J. Syst. Softw. 23*, 111–122. (Cited on pages 14 and 15.)

[36] MAHMUD, A. A., DADLANI, P., MUBIN, O., SHAHID, S., MIDDEN, C. J. H., AND MORAN, O. 2007. iparrot: Towards designing a persuasive agent for energy conservation. In *Persuasive Technology* (2008-01-21), Y. de Kort, W. IJsselsteijn, C. J. H. Midden, B. Eggen, and B. J. Fogg, Eds. Lecture Notes in Computer Science, vol. 4744. Springer, 64–67. (Cited on pages 36 and 76.)

[37] MAZZOTTA, I., DE ROSIS, F., AND CAROFIGLIO, V. 2007. Portia: A user-adapted persuasion system in the healthy-eating domain. *Intelligent Systems, IEEE 22,* 6 (nov.-dec.), 42 –51. (Cited on page 37.)

[38] MCCABE, T. 1976. A complexity measure. *IEEE Transactions on Software Engineering 2*, 308–320. (Cited on pages 12 and 13.)

[39] MCCABE, T. J. AND BUTLER, C. W. 1989. Design complexity measurement and testing. *Commun. ACM 32*, 1415–1425. (Cited on pages 13 and 50.)

[40] MIDDEN, C. AND HAM, J. 2009. Using negative and positive social feedback from a robotic agent to save energy. In *Proceedings of the 4th International Conference on Persuasive Technology*. Persuasive '09. ACM, New York, NY, USA, 12:1–12:6. (Cited on pages xi and 33.)

[41] MILLER, G. 1980. On being persuaded: Some basic distinctions. In *Persuasion: New directions in theory and research*, M. E. Roloff and G. R. Miller, Eds. SAGE Publications, 11–28. (Cited on page 29.)

[42] NAWYN, J. 2006. A persuasive television remote control for the promotion of health and well-being. M.S. thesis, Massachusetts Institute of Technology, Massachusetts. (Cited on pages xii and 38.)

[43] OINAS-KUKKONEN, H. AND HARJUMAA, M. 2008. A systematic framework for designing and evaluating persuasive systems. In *Proceedings of the 3rd international conference on Persuasive Technology*. PERSUASIVE '08. Springer-Verlag, Berlin, Heidelberg, 164–176. (Cited on pages 30, 34, 35, and 63.)

[44] OLAGUE, H. M., ETZKORN, L. H., MESSIMER, S. L., AND DELUGACH, H. S. 2008. An empirical validation of object-oriented class complexity metrics and their ability to predict error-

prone classes in highly iterative, or agile, software: a case study. *J. Softw. Maint. Evol. 20*, 171–197. (Cited on page 26.)

[45] ORINO, M. AND KITAMURA, Y. 2010. An approach to create persuasive web sites. *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on 3*, 116–119. (Cited on pages xii, 36, and 37.)

[46] ORTEGA, M., PÉREZ, M., AND ROJAS, T. 2003. Construction of a systemic quality model for evaluating a software product. *Software Quality Control 11,* 3, 219–242. (Cited on pages 26 and 27.)

[47] PRICE, M. W. AND DEMURJIAN, SR., S. A. 1997. Analyzing and measuring reusability in object-oriented design. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, New York, NY, USA, 22–33. (Cited on pages xi, 15, and 16.)

[48] PURAO, S. AND VAISHNAVI, V. 2003. Product metrics for object-oriented systems. *ACM Comput. Surv. 35,* 2, 191–221. (Cited on page 26.)

[49] RUBEY, R. J. AND HARTWICK, R. D. 1968. Quantitative measurement of program quality. In *Proceedings of the 1968 23rd ACM national conference*. ACM '68. ACM, New York, NY, USA, 671–677. (Cited on pages 7, 10, 11, and 19.)

[50] SATO, D., GOLDMAN, A., AND KON, F. 2007. Tracking the evolution of object-oriented quality metrics on agile projects. In *Proceedings of the 8th international conference on Agile processes in software engineering and extreme programming*. XP'07. Springer-Verlag, Berlin, Heidelberg, 84–92. (Cited on page 26.)

[51] SCHNEIDEWIND, N. 2002. Body of knowledge for software quality measurement. *Computer 35,* 2 (Feb.), 77 –83. (Cited on pages xi and 22.)

[52] SEDIGH-ALI, S., GHAFOOR, A., AND PAUL, R. A. 2001. Software engineering metrics for cots-based systems. *Computer 34,* 5, 44–50. (Cited on pages xi, 24, 25, and 26.)

[53] SLINGER, S. 2005. Code smell detection in eclipse. M.S. thesis, Delft University of Technology Faculty of Electrical Engineering, Mathematics and Computer Science Department of Software Technology. (Cited on pages xi, 3, and 4.)

[54] STOCKMAN, S., TODD, A., AND ROBINSON, G. 1990. A framework for software quality measurement. *Selected Areas in Communications, IEEE Journal on 8,* 2 (Feb.), 224 –233. (Cited on pages 23 and 24.)

[55] SUCCI, G., PEDRYCZ, W., DJOKIC, S., ZULIANI, P., AND RUSSO, B. 2005. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Em-*

pirical Software Engineering 10, 81–104. 10.1023/B:EMSE.0000048324.12188.a2. (Cited on pages xii, 22, 49, and 50.)

[56] TANG, M.-H., KAO, M.-H., AND CHEN, M.-H. 1999. An empirical study on object-oriented metrics. In *Proceedings of the 6th International Symposium on Software Metrics*. IEEE Computer Society, Washington, DC, USA, 242–. (Cited on page 26.)

[57] TUNG, F. AND CHOU, Y. 2009. Designing for persuasion in everyday activities. *Design Semantics of Form Movement DesForM09*, 105–113. (Cited on pages xii, 35, and 37.)

[58] VAN EMDEN, E. AND MOONEN, L. 2002. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. IEEE Computer Society, Washington, DC, USA, 97. (Cited on pages xi, 2, and 3.)