Empirical Evaluations of Semantic Aspects in Software Development

This thesis presents empirical research in the field of software development with a focus on handling semantic aspects. There is a general lack of empirical data in the field of software development. This makes it difficult for industry to choose an appropriate method for their particular needs. The lack of empirical data also makes it difficult to convey academic results to the industrial world.

This thesis tries to remedy this problem by presenting a number of empirical evaluations that have been conducted to evaluate some common approaches in the field of semantics handling. The evaluations have produced some interesting results, but their main contribution is the addition to the body of knowledge on how to perform empirical evaluations in software development. The evaluations presented in this thesis include a between-groups controlled experiment, industrial case studies and a full factorial design controlled experiment. The factorial design seems like the most promising approach to use when the number of factors that need to be controlled is high and the number of available test subjects is low.

Another contribution of the thesis is the development of a method for handling semantic aspects in an industrial setting. A background investigation performed concludes that there seems to be a gap between what academia proposes and how industry handles semantics in the development process. The proposed method aims at bridging this gap. It is based on academic results but has reduced formalism to better suit industrial needs. The method is applicable in an industrial setting without interfering too much with the normal way of working, yet providing important benefits. This method is evaluated in the empirical studies along with other methods for handling semantics.
Empirical Evaluations of Semantic Aspects in Software Development
Martin Blom. *Empirical Evaluations of Semantic Aspects in Software Development*

DISSERTATION

Karlstad University Studies 2006:26  
ISSN 1403-8099  
ISBN 91-7063-057-7

© The author

Distribution:  
Karlstad University  
Faculty of Economy, Communication and IT  
Computer Science  
SE-651 88 KARLSTAD  
SWEDEN  
+46 54-700 10 00

www.kau.se

Printed at: Universitetstryckeriet, Karlstad 2006
Abstract

This thesis presents empirical research in the field of software development with a focus on handling semantic aspects. There is a general lack of empirical data in the field of software development. This makes it difficult for industry to choose an appropriate method for their particular needs. The lack of empirical data also makes it difficult to convey academic results to the industrial world.

This thesis tries to remedy this problem by presenting a number of empirical evaluations that have been conducted to evaluate some common approaches in the field of semantics handling. The evaluations have produced some interesting results, but their main contribution is the addition to the body of knowledge on how to perform empirical evaluations in software development. The evaluations presented in this thesis include a between-groups controlled experiment, industrial case studies and a full factorial design controlled experiment. The factorial design seems like the most promising approach to use when the number of factors that need to be controlled is high and the number of available test subjects is low. A factorial design has the power to evaluate more than one factor at a time and hence to gauge the effects from different factors on the output.

Another contribution of the thesis is the development of a method for handling semantic aspects in an industrial setting. A background investigation performed concludes that there seems to be a gap between what academia proposes and how industry handles semantics in the development process. The proposed method aims at bridging this gap. It is based on academic results but has reduced formalism to better suit industrial needs. The method is applicable in an industrial setting without interfering too much with the normal way of working, yet providing important benefits. This method is evaluated in the empirical studies along with other methods for handling semantics. In the area of semantic handling, further contributions of the thesis include a taxonomy for semantic handling methods as well as an improved understanding of the relation between semantic errors and the concept of contracts as a means of avoiding and handling these errors.
# Contents

List of Publications xi

Acknowledgements xiii

1 Introduction 1
   1.1 Problem Definition . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.2 Main Contribution . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.3 Outline of Thesis . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Empirical Methods in Software Development 5
   2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   2.2 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.2.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.2.2 Process . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.2.3 Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
      2.2.4 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   2.3 Case Studies . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
      2.3.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . 12
      2.3.2 Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
      2.3.3 Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
      2.3.4 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
   2.4 Surveys . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      2.4.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      2.4.2 Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
      2.4.3 Validity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
      2.4.4 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
   2.5 Data Gathering Techniques . . . . . . . . . . . . . . . . . . . . 19
   2.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3 Methodology

3.1 Initial Hypothesis .................................................. 23
3.2 Industrial Case Study 1 ............................................. 24
3.3 Theory Building ...................................................... 24
3.4 Controlled Experiment 1 ............................................ 25
3.5 Theory Revision ...................................................... 25
3.6 Industrial Case Study 2 ............................................ 26
3.7 Controlled Experiment 2 ............................................ 26

4 Semantic Aspects in Software Development .................................. 29

4.1 Background .......................................................... 30
4.2 Terms and Definitions .............................................. 30
4.3 A Taxonomy of Semantic Specification Techniques ...................... 32
  4.3.1 General Issues of Semantic Concern .......................... 32
  4.3.2 Levels of Formalism for Semantic Specifications ............... 33
  4.3.3 Phases in a Module’s Life .................................... 40
  4.3.4 The Taxonomy .................................................. 43
  4.3.5 Summary and Conclusions ................................... 44
4.4 An Industrial Case Study on Semantic Aspects .......................... 45
  4.4.1 General Criteria ................................................ 45
  4.4.2 Interface Criteria .............................................. 46
  4.4.3 Internal Criteria .............................................. 47
  4.4.4 External Criteria .............................................. 48
  4.4.5 Case Study Environment .................................... 48
  4.4.6 Case Study Project .......................................... 49
  4.4.7 Examination of the Criteria .................................. 49
  4.4.8 Example Application of Criteria ............................. 51
  4.4.9 Results from the Case Study ................................ 52
  4.4.10 Summary and Conclusions .................................. 53
4.5 A Detailed Example from the Case Study ................................ 53
  4.5.1 Switching Sections ............................................ 54
  4.5.2 Scope ........................................................ 54
  4.5.3 Structure and Documentation ................................ 55
  4.5.4 The Case Study and its Semantic Integrity .................... 57
  4.5.5 A Design Structure and Development Method for Switching Sections ........................................... 58
  4.5.6 Case Study Compared to the Proposed Method ................ 62
  4.5.7 Summary and Conclusions ................................... 64
4.6 Discussion ............................................................ 65
7 Concluding Remarks ........................................ 119
  7.1 Contribution ........................................... 119
     7.1.1 Empirical Evaluations of Software Development .... 119
     7.1.2 Method for Handling Semantic Aspects ............. 120
     7.1.3 Taxonomy for Semantic Handling Methods .......... 121
     7.1.4 Relation Between Contracts and Errors ............ 121
  7.2 Impact of Thesis ....................................... 121
  7.3 Future Work ........................................... 122

A SEMLA .................................................. 131
  A.1 Introduction .......................................... 131
     A.1.1 Applicability of the Method ....................... 131
     A.1.2 Appendix Structure ................................ 132
  A.2 Terminology .......................................... 132
     A.2.1 Module ............................................. 133
     A.2.2 Interface .......................................... 133
     A.2.3 Black Box .......................................... 134
     A.2.4 Semantics and Semantic Integrity ................. 134
  A.3 Guidelines for Design .................................. 135
     A.3.1 General Guidelines ................................ 135
     A.3.2 Guidelines for Interfaces ......................... 135
     A.3.3 External Guidelines ................................ 140
     A.3.4 Internal Guidelines ............................... 143
  A.4 Guidelines for Documentation ........................... 147
     A.4.1 The Purpose and the Parts of the Documentation ... 148
     A.4.2 Documentation of the Overview .................... 148
     A.4.3 Documentation of the Interface .................... 150
     A.4.4 Documentation of the Implementation ............. 153
  A.5 Guidelines for Maintenance and Reuse .................. 156
     A.5.1 Maintenance vs. Reuse ............................. 156
     A.5.2 Maintenance of a Module .......................... 156
     A.5.3 Reuse of a Method ................................ 161
  A.6 Quick Reference Guide .................................. 162
     A.6.1 Quick Reference for Module Design ................. 162
     A.6.2 Quick Reference for Documentation ................. 163
     A.6.3 Quick Reference for Maintenance and Reuse ........ 164
  A.7 List of Terms ........................................... 164
  A.8 General Guidelines .................................... 166
     A.8.1 Use a Modular Design .............................. 166
     A.8.2 Keep Down the Number of Dependencies .......... 166
     A.8.3 Encapsulate Data and Methods ..................... 167
     A.8.4 Hide Data and Methods ............................. 168
     A.8.5 Use Abstractions of Data and Methods ............. 169
A.9 Documentation and Implementation of a List .......................... 170
  A.9.1 Overview ....................................................................... 171
  A.9.2 Interface ......................................................................... 172
  A.9.3 Implementation and Internal Description ......................... 176
List of Publications

Parts of the work presented in this thesis have been published in the following publications:


LIST OF PUBLICATIONS

• "Error Management with Design Contracts", Eivind J. Nordby, Martin Blom, Anna Brunstrom, Proceedings from First Swedish Conference on Software Engineering Research and Practice (SERP’01), Blekinge Institute of Technology, pages 53-59. ISSN 1103-1581, ISRN BTH-RES–01/10–SE.


Acknowledgements

I would first like to send a big thank you to my wife Elin for agreeing to live with me and for keeping me real and opening my eyes, and to my family, both biological and in-law, for supporting me in my endeavors. I would also like to thank all friends for rooting for me and for making my spare time enjoyable.

I would further like to thank my supervisor Anna Brunstrom for not giving up on me, despite us being different and despite my habit of never saying no to anything which has led me astray from the PhD studies on many occasions. Like having Yin supervising Yang or vice versa.

Eivind J. Nordby, my loyal partner in crime, deserves a big thank you for always helping out and for providing many interesting discussions on various topics. Erland Jonsson, my assistant supervisor, has also always pushed me forward by being clever and seeing possibilities rather than problems. Thank you.

My colleagues too deserve a thank you for providing a friendly atmosphere and for putting up with me and my rambling. I see a bright future for the department of computer science. With so many good people at one place, there is no failure.

I would also like to thank our industrial partners for allowing us insight into and access to their projects. Thank you TSAC, Ericsson Infotech AB and particularly project Skywalker for your kind cooperation and support during the studies.

I would further wish to thank NUTEK, Vinnova and the Knowledge Foundation of Sweden (KK-stiftelsen) for funding my studies and helping new universities develop their research.

Last, but not least I would like to dedicate this thesis to all those who are no longer among us and to those who are to come.
Chapter 1

Introduction

Software development is an interesting and ever-evolving area and constitutes the core area of software engineering. The main concern in software development is the production of high-quality software. Since software is a complex product, there are many different ways in which the quality can be improved. Everything from process improvement [62, 82], through inspections [40, 73] down to design tools [36, 33] and specification languages [12, 14, 46] has been proposed as partial solutions to the quality problem. Many of the proposed solutions have been applied successfully and some have been empirically shown to have positive effects. A large number of solutions, however, look promising, but lack empirical evaluations that show what effects the solutions have [7]. This is a problem since empirical data is necessary if the solution is to be widely accepted [26, 71, 78]. There are some indications that local evidence in contrast to general evidence often is needed to persuade professionals that the empirical evidence applies to their particular situation [65]. The work presented in this thesis gives insight into empirical evaluations in software development with a focus on semantic issues and shows both successful and unsuccessful experiments performed in academia and in industry. The thesis thus provides one step in the process of obtaining a more mature empirical environment in the software development area. Hopefully, there will be more empirical evaluations performed in the future, such that software managers can decide on scientific grounds what approach to use for a particular situation. The thesis also presents a novel method for handling semantics in a controlled but not overly formal way. The method can be used in industrial settings without interfering too much with the normal way of working, yet providing important benefits.
CHAPTER 1. INTRODUCTION

1.1 Problem Definition

The main problem this thesis addresses is how to evaluate software development techniques and methods. It is not uncommon that a method or technique is used in large-scale software projects without empirical proof of its benefits. If more empirical investigations in the field were done, the selection of methods and techniques would be made easier [25, 71]. The main question is thus how software development methods can be compared and evaluated in a scientific way [69]. This thesis makes no claims to entail answers to all problems regarding empirical evaluations in software development, but provides some insight into the problem area and presents some lessons learned.

Another problem the thesis addresses is how software developers can achieve a higher level of semantic quality in their software products without having to master a specific language or a complex method or tool.

1.2 Main Contribution

The main contribution of this thesis is an increased understanding of empirical methodology in conjunction with software development. In the thesis, a number of experiments using different experimental designs are presented as well as lessons learned from the experiments. The results from the earlier experiments are not conclusive, but the experiments per se have provided an increased understanding of how to plan and execute experiments in a software development setting. The latest experiment shows that factorial experiment designs are suitable for experiments where the number of test subjects is limited and where a large number of factors need to be controlled and gauged. The experiment also provides some statistically significant results and showed that the methods for handling semantics influenced the development time. Another contribution is the development and evaluation of a software development method that helps developers focus on important semantic aspects in the development process. More details on the main contributions of this thesis can be found in Chapt. 7.1.

1.3 Outline of Thesis

The remainder of the thesis is structured as follows: Chapter 2 presents different empirical techniques and methods usable in software development, to give some background into what methods are available. Chapter 3 presents the research methodology used in the thesis and explains the choice of techniques and methods in some detail. Chapter 4 presents a survey on the handling of semantic aspects in academia and in industry. It proposes a taxonomy that summarizes
1.3. OUTLINE OF THESIS

how semantic aspects are handled in academia as well as examples on how semantic aspects are managed in industry. This chapter provides the background information on semantics. Since there is a gap between how academia wants semantic aspects to be handled and how they are generally treated in industry, a method that tries to bridge this gap was developed and is presented in Chapt. 5. This method is evaluated together with other related methods in Chapt. 6. This chapter presents three empirical evaluations that illustrate different empirical designs in a software development setting. The lessons learned from these evaluations are presented at the end of each section. Finally, Chapt. 7 entails a discussion on the results and their impact as well as some conclusions.
Chapter 2

Empirical Methods in Software Development

In software development, as in all sciences and technologies, empirical evaluations are often needed to assess the qualities of a method, a tool, a change in work process or any other entity. The empirical evaluations can be done using a variety of methods and techniques for obtaining and treating the data. This chapter presents three of the most common methods used in empirical evaluation, *experiments, case studies* and *surveys*. First, a shorter overview of the three methods is given, followed by a more in-depth description and discussion where the benefits and drawbacks of each method are presented as well as a number of examples. The chapter further outlines techniques for data gathering and their areas of application. The definitions and descriptions used in the chapter conform to those found in [84] and [9].

2.1 Overview

An empirical evaluation is concerned with understanding or identifying one or more *variables*. In many cases, the main focus of the evaluation is to understand the variations in a particular variable. As will be shown later in the chapter, different empirical methods are more suitable to certain problems. The ideal outcome for any empirical evaluation is to obtain results that are generally applicable in a certain context. Because this context is usually rather large, i.e. all programmers or all requirement engineering approaches, it is often necessary to take a *sample* from the total population and perform the evaluation on that sample. The results from the evaluation on the sample is then generalized back to the total population. This process is illustrated in Fig 2.1. A common research focus is to identify and quantify relationships between different variables.
and perhaps most commonly the relationship between a number of background variables and a main variable\(^1\).

One of the main differences between the empirical evaluation methods is the level of control. In experiments, it is possible to exercise a high level of control whereas case studies and surveys generally have a lower level of control. In an experiment, it is possible to control not only the gathering of data, but also the background variables. In case studies, it is generally not as easy to control background variables since a case is run in a real environment. Surveys are usually done to evaluate a main variable and can collect information on background variables, but not control them per se. Figure 2.2 tries to summarize the level of control and the effects the level of control has on the generalizability of the results for the different methods. It is worth noting that the figure only tries to summarize the inherent properties of the methods and that the preparations before and the execution of an evaluation are what determines the real levels of control and generalizability.

The following sections start by presenting the respective methods, their application areas, definitions and processes. After that discussions on the validity threats to the methods are given. Four validity areas [84] are addressed:

- Conclusion validity, which is concerned with the strength of the results, i.e. if the results are conclusive or not.

---

\(^1\)In reality, most empirical evaluations are monitoring more than one main variable, but for simplicity, we will use the singular form in the remainder of the thesis.
2.1. OVERVIEW

Figure 2.2: Empirical Taxonomy

- Internal validity, which is concerned with the cause-effect relationship, i.e. if the results are caused by factors taken into account or not.

- Construct validity, which is concerned with the relationship between theory and observation, i.e. if the evaluation setup corresponds to the real world setup.

- External validity, which is concerned with generalization from the evaluation population to the total population, i.e. if the results are generally applicable or only valid in the sample population.

It is worth noting that the four different validity areas affect each other to a certain extent. Construct validity, for instance, influences external validity in that a badly constructed evaluation setting corresponds equally badly to the real world. That in turn makes it difficult to generalize the results, thus lowering the external validity. The same type of discussion applies to other combinations of validity threats. Since the validity areas are not disjunct, it is sometimes also difficult to associate a validity threat with only one validity area. A successful evaluation must nevertheless take the threats to validity seriously and handle the threats as thoroughly as possible.

After the validity of the respective method has been discussed, each method is exemplified by good research articles that highlight some of the aspects presented before regarding the method.
CHAPTER 2. EMPIRICAL METHODS

2.2 Experiments

An experiment is the most well controlled empirical approach. It has the power to isolate the aspect of interest and hence produce more unambiguous results. In an ideal experiment, all variables are to be controlled except the variable under evaluation. The high level of control makes it possible to detect small differences in the main variable, differences that might go unnoticed using any of the other methods presented below. The drawback is that designing experiments in areas like software development, where most experiment designs demand some extra actions from the participants, is difficult. The isolation of certain aspects and the exclusion of other also impose more restrictions on the participants, thus making the situation less natural and the results more difficult to generalize. Although experiments have inherently high generalizability, it is therefore often difficult to generalize the results of experiments in software development.

If the environment in which the experiment is to be implemented cannot allow too many intrusions, one possibility is a quasi-experiment. It might for instance not be possible to randomly assign people to tasks or methods and hence a proper experiment cannot be constructed. A quasi-experiment can still be done, sacrificing some of the statistical strength. A quasi-experiment can for instance be executed in an on-line setting, where the main focus of the participants is regular work rather than experiment-related work. The organization where the quasi-experiment is done does not have to change or behave much differently than normal.

2.2.1 Definitions

In a controlled experiment, certain background variables, referred to as treatments, are varied in a controlled way and the changes in the main variable is observed. If the evaluation for instance aims at determining what method of documenting software is the most time effective, the methods are the treatments and the main variable is the time spent on documentation. Variables that might influence the outcome but that are not of main interest and that have been identified prior to the execution of the evaluation can be controlled using blocking and randomization. Blocking means dividing a variable into blocks such that its effects on the outcome can be identified as stemming from the blocks rather than from the main variable. Randomization means distributing the random variations not handled by blocking evenly such that they do not affect any of the treatments more than the other. As written by Box, Hunter, Hunter [9], "Block what you can and randomize what you cannot". Both blocking and randomization are control mechanisms adding to the high level of control in experiments.

Below is a list of research questions that experiments can provide answers to:
2.2. EXPERIMENTS

- Which of the available methods are most suitable with regard to some criteria in a certain context?
- Which of the techniques perform the fastest in a given situation?
- Is application one safer than application two according to certain criteria?

As can be seen in the list, common questions for experiments are of the same type, trying to compare a number of different candidates to evaluate which one is superior. There are of course experiments that evaluate whether something works or not, but it can be argued that in software development the majority of experiments compare at least two candidate solutions.

2.2.2 Process

Figure 2.3 below entails a schematic description of the experiment process and its phases. Every experiment starts with an idea of what the experiment should evaluate. This initial idea might not be possible to evaluate right away, but needs some definition and reformulation. The problem needs to be in an evaluable format such that an experiment is able to produce results providing input to the problem. After the problem has been defined, the experiment design has to be decided upon. There exist a large number of experimental setups, but this plethora can be categorized into two major categories; comparison of 2 treatments, and comparison of k treatments. Both these categories can be divided into two subcategories; blocked and unblocked arrangements. Unblocked arrangements are primarily used for simplicity or when the object of study is easy to isolate. Blocked designs are more applicable because they can be used
in all experiments, but come at a price due to the increased need to control and monitor other variables than the one of interest. A decision is then made on how many treatments should be used, what type of data analysis is to be made at a later stage, what participants to include (if humans are involved) and how the experiment should be executed in a feasible way. The design phase is crucial and the resulting design should be reviewed and possibly redesigned as indicated in the figure. When the design appears reasonable, the experiment can be executed. In the execution phase, it is important to monitor and control as much as possible. Even if all previous phases are done perfectly, the execution phase can destroy all good ideas and preparations if not done properly. Making sure that threats to the results are controlled and/or monitored is perhaps the most important task in the execution phase. The last phase in the process is to analyze the data to extract statistically sound results.

If the results are statistically valid, they can be strengthened by having another researcher replicate the experiment. If the results are non conclusive, replication of a previously executed experiment can provide the solution using the knowledge gained in the first experiment as input to the second. Replication is also an important method for building the body of knowledge within a field. If two or more independent evaluations reach the same conclusion, that conclusion is stronger than without replication. If replicated evaluations produce diametrically different results, however, there is either some problems with the execution of either experiment or some other unforeseen issue that needs to be addressed. It could for instance be that the underlying cause-effect relationship of the hypothesis is wrong and needs to be reformulated or that there exist a previously unknown variable that is what causes the variations in the output variable.

### 2.2.3 Validity

A properly constructed experiment generally has high conclusion validity since background variables can be controlled, thus making it easier to find a cause-effect relationship between the treatments and the main variable. The real conclusion validity can, however, be lowered by poor execution or poor statistical evaluation of the data. Experiments can due to the high level of control provide high internal validity. If one does not obtain high internal validity in an experiment, it is often due to poor planning or execution of the experiment. Construct validity concerns how well the environment and experiment setup represent the reality in which the problem is found. If reality has been properly understood by the researchers and properly modeled in the experimental design, construct validity is high. It is, however, not always easy to understand and model reality and hence not all experiments achieve high construct validity. One common problem is the usage of students in the roles of software professionals, as discussed in [34, 68]. The students often get to play the part of software
developers, even though they are actually not. The students can be said to be a construct of the actual entity software professionals, and hence fall into the area of construct validity. The usage of students also make any generalization of the results to the outside world difficult, thus lowering the external validity. The problem with using students is most strongly connected to experiments, since surveys and case studies are more often performed on non-constructed artefacts such as real projects or real software professionals. It might, however be argued that students at later stages in their education often possess almost the same properties as professionals do and it depends on the type of evaluation which position to assume. What is important when performing experiments and other empirical evaluations is to be aware of the threats that do exist and to try and remedy these as well as possible. In conclusion, experiments have the potential of achieving high conclusion and internal validity, but for the area of software development, they are at the same time susceptible to low external and construct validity.

2.2.4 Examples

In this section, some good examples of properly executed software development experiments are presented. Both examples are taken from the large array of good experiments produced by Lutz Prechelt and Walter Tichy. They are included to illustrate how empirical research can be done in a software development setting and highlight some of the concepts presented above. They can serve as input for further studies into experimental evaluations by showing some combinations of research problems and experimental solutions to the problems. Both studies can be said to be of type 1 as presented in 2.2.1: Which of the available methods are most suitable with regard to some criteria in a certain context? The first experiment tries to answer the question: Which of the programming languages is most suitable with regard to correctness, time and space? The second experiment answers the question: Are methods based on patterns or non-patterns most suitable with regard to time and correctness?

Prechelt - An empirical comparison of seven programming languages [61]

This article presents an experiment that compared seven different programming languages (C, C++, Java, Perl, Python, Rexx and Tcl) using good experimental methodology. The experimental methodology was a between-groups test where each programmer used one of the seven programming languages. This setup has the obvious drawback of having to handle differences in competence between the groups. The treatments were the programming languages and the main variables were correctness of the resulting programs, time consumption, lines of code and productivity. The study included 80 programs written both by
subjects in the form of university students and volunteers found by advertising in newsgroups. The program implemented by the test persons was a phone code problem, which implies a conversion from telephone numbers into word strings according to certain rules. The implementations were tested with three test files, one large file with valid numbers, one large file with empty entries and one totally empty file. These test files and the assignment specifications are the only explicit constructs used in the experiment. Other implicit constructs include using students and volunteers as constructs of real software developers. The results were analyzed using box plots and Mann-Whitney tests. The results show that the script programs were faster to construct and were half as long as the non-script programs. The script programs consumed twice as much memory as the non-script programs. The execution time overhead of Java was huge compared to C and C++, but execution times were still acceptable for all languages.

Prechelt and Tichy- A controlled experiment in maintenance: comparing design patterns to simpler solutions [63]

This paper entails a controlled experiment using a good experimental design that investigates the merits of using patterns in maintenance situations. The experiment was done using four groups of test subjects working with four different programs in two different versions, one using patterns and one not. The subjects were to implement new functionality into existing programs, constructed by the researchers. Time and correctness was recorded. The experimental setup was roughly a blocked factorial design where all subjects were exposed to all methods and solved all assignments, albeit in different versions and in different order. The results from the experiment show that patterns are mostly beneficial to use, but not always. In some cases patterns made the maintenance tasks harder.

2.3 Case Studies

If no intrusions except for the data collection can be done, an appropriate approach is a case study where data is gathered by observation or enquiries such as questionnaires. Since few intrusions in the regular work are done, the level of control is lower than for experiments, but case studies is still a valuable tool due to its low cost and high level of realism.

2.3.1 Definitions

A case study usually implies a detailed study of one or a few particular cases. A case study is in principle non-intrusive because the main objective is observation
rather than control. A case study is done when information on an existing entity is needed as-is and where the low level of control can be accepted. Evaluation of a new tool in production is a typical example where a case study might be suitable. Below is a list of questions suitable for an approach using case studies:

- How much time is the average software developer at company X using for documentation?
- How do the software developers use the online help?
- Does method Y seem appropriate for solving certain problems?

The types of questions are more inclined towards obtaining an understanding of a single variable in a specific context, rather than relating it to other entities or evaluating its general merits. A case study is perfect for using in a pre study to an experiment due to its low price and ease of use and can provide one small input to the general problem area. Case studies are often performed as a first step towards empirically showing the merits of for instance a new piece of technology, a new process or a new tool. A case study can at least show that the new artefact works in a given context, thus encouraging more empirical research on the artefact.

### 2.3.2 Process

The process of performing a case study is similar to that of experiments, as can be seen in Fig. 2.4 below, but it also has a number of differences. The

![Case Study Process](image)

**Figure 2.4: Case Study Process**

process starts with an idea that is formulated as a research problem. After
CHAPTER 2. EMPIRICAL METHODS

that, definition of necessary properties of the case study follows which provides input to the selection of a suitable case. When the case is run, it is studied by the researchers who gather data which is analyzed to produce results. Since a case study is done live on a real project, it is not possible to replicate exactly the same study as is indicated in the figure by the arrow going back to selection of a suitable case rather than to the beginning of the actual study. To obtain more data, it is necessary to select a new suitable case and then perform a study on that. It is of course possible that a very similar project is done in the same environment, making both case studies similar and thus rendering almost the same effects as a replication would.

2.3.3 Validity

High conclusion validity is generally harder to obtain as compared to experiments, since background variables cannot be controlled as rigidly. The background variables can, however, be monitored without disturbing the normal work flow of the case, thus making it possible to relate the outcome of the main variable to the observed values of the other variables. This in turn makes it possible to draw stronger conclusions and hence increasing the conclusion validity. Due to the lack of control, the internal validity can suffer to a great extent, especially when evaluating cause-effect relationships. The construct validity is very high because normally, no constructions are needed for a case study and hence the correlation between the study and what is being studied is one-to-one. External validity is another matter. It is often hard to generalize the results to settings other than those present in the case study. This problem arises from the low control usually exercised in a case study and the specificity of the case itself. The study can say something about the case itself, but not too much on the more general problem. In conclusion, case studies can have high construct validity, but generally lower conclusion, internal and external validity.

2.3.4 Examples

The examples of case studies are both focused on extreme programming, a relatively new notion in the software industry. The first study is done in an academic environment and the second in an industrial setting. What is typical is that the case studies are done at a relatively early stage in the chosen field and roughly conforms to the third type of question listed in 2.3.1: Does method Y seem appropriate for solving certain problems? Both case studies tries to answer the question: Does extreme programming seem appropriate for solving software development tasks?
2.4. SURVEYS

Müller - Case Study: Extreme Programming in a University Environment [53]

In this article the authors try to evaluate some of the claimed benefits of using Extreme Programming (XP). This was done in a university setting using graduate students as subjects, implementing three assignments. Although the setup so far is similar to one that would be used for a controlled experiment, the difference lies in that no control groups are used. The subjects are trying the new technique and their experiences from it are recorded. The results from the study show that pair programming was easily adopted and perceived as enjoyable, but that designing in small increments was difficult and was perceived as "design with blinders". To write test cases before implementing was also seen as difficult and sometimes impractical. XP did not scale well due to the communication overhead and need coaching until fully adopted.

Hulkko - A multiple case study on the impact of pair programming on product quality [30]

This paper presents two interesting empirical issues, first it presents the state-of-the-art in literature regarding pair programming, and second it presents a multiple case study trying to judge how well pair programming works. The case studies were done using various volunteers from industry that implemented assignments. The results from previous research differ somewhat from the results provided by the multiple case studies presented in the article. Productivity was sometimes higher and sometimes lower when using pair programming as compared to solo programming. Adherence to coding standards was lower using pair programming, which is contrary to what might be expected. Comment ratio was higher for pair programming, whereas defect density showed no definitive pattern. To summarize, some of the claimed benefits of pair programming were confirmed whereas some were not.

2.4 Surveys

A survey is generally done when opinions or trends are the focus, investigating a representative sample of the total population. The results from the survey are then analyzed to find patterns or anomalies, and then generalized back to the total population.

2.4.1 Definitions

For a survey, the means by which data gathering is done is often in forms of questionnaires, observations or literature studies. Surveys are, as mentioned previously, the most non-intrusive approach which makes them cheap to perform
but hard to control. Since a survey usually measures existing opinions and
trends, there is really no control at all being exercised. In the social and political
sciences, surveys are often used to determine how a population relates to an
upcoming event, commonly an election of politicians. In software development
a survey is often done when trends or best-practices are to be investigated.
Below are three examples of research questions that might be answered by using
surveys:

- Which software development tool is most used in a given context today?
- What do researchers in a particular area propose as a solution to a certain
  problem?
- How is a certain aspect reported in the literature?

Problems generally applicable for surveys often focus on obtaining an under-
standing of how and why a specific variable varies. The difference as compared
to experiments is that experiments manipulate background variables (treat-
ments) and examines the main variable whereas surveys only look at the main
variable.

2.4.2 Process

The process of performing a survey naturally has many similarities to the exper-
iment process, but is not identical as can be seen in Fig 2.5. The process begins

Figure 2.5: Survey Process

with an idea, as do all empirical evaluations. This idea needs to be formulated
as a problem such that the survey can provide an answer to the problem. After
that, the survey needs to be designed to provide input to the problem. This activity has two sub-activities, identification of total population and variables of interest, and definition of sample from the total population. Once these steps are done, the actual survey can commence followed by data analysis. Replication can be done in two different ways, either the same sample is used again or a new sample from the population is taken. Using the same sample for the same evaluation can be done if trends or changes in opinions are the focus whereas a new sample is needed if the new survey demands a fresh sample. This might be because the first survey might have biased the sample and the second survey needs an unbiased sample. Replication might also be done when a larger sample is needed to obtain statistically stronger results than were possible using a small sample.

2.4.3 Validity

Conclusion validity can be very high in a survey that focuses on capturing trends within a particular variable and is mainly affected by the size of the sample. If the survey is concerned with cause-effect relationships, the lack of control over background variables makes it difficult to achieve high conclusion validity. Internal validity, i.e. if the survey measures what it is intended to, depends on how the actual data gathering is done and is not necessarily higher or lower than for other approaches. However, since most surveys are concerned with examining a single variable and not identifying relationships between variables, the internal validity is only affected by how the data is gathered and not with complex relationships. Surveys usually employ few or no artefacts and hence construct validity is of little concern, except for the construction of questionnaires and other data gathering techniques. External validity is by definition high if the sample used in the survey is representative of the total population. In conclusion, surveys can have high internal, construct, and external validity. Conclusion validity can also be high, but only in surveys not concerning cause-effect relationships.

2.4.4 Examples

There are a number of articles presenting surveys regarding many different aspects of software development. The two examples chosen are not only thorough and well executed; they are also meta-studies of the entire field of software engineering research. Both surveys hence provide not only examples of surveys, but also an insight into software development research as such. Both surveys aim at understanding a particular area of interest, in these cases, software engineering research. Both articles can be said to answer questions 1 and 3 as presented in 2.4.1, Which SW development tool is most used in a given context today and how is a certain aspect reported in the literature? The instantiation of these two
questions would roughly be: *Which research method is most used in software engineering today and how is research methodology reported in the literature?*

**Glass - Research in software engineering: an analysis of the literature [25]**

This article presents a thorough review of research in software engineering. The authors have examined 369 papers published in leading journals within the field of software engineering research. The article assesses for each paper the topic, research method, reference disciplines and the levels of analysis used among other aspects. As for the research topics, the articles show a good variation, but for the other aspects of the articles, the focus is considerably narrower. Most papers used "conceptual analysis" and "concept implementation" as research method, which means that they aim at analyzing or implementing a certain concept. This finding confirms the need for more empirical evaluations. Most authors do not rely on other disciplines for reference, which might be an indication of the immaturity of software research. Once an area is better understood, it is easier to relate it to other disciplines. As for the levels of analysis used, the predominant level is technical. Very few used "social" levels.

**Sjøberg - A Survey of Controlled Experiments in Software Engineering [71]**

This article reports on controlled experiments, by which is meant both randomized experiments and quasi-experiments. The authors examined over 5400 article titles and abstracts from leading journals and conferences in order to find those who reported on controlled experiments. The terminology used in different papers apparently varies and makes it harder to determine where a true experiment has actually been done. The authors report that the word 'experiment' is sometimes used for evaluations where no treatment is ever applied. In the final analysis 103 papers remained, which is roughly 2 % of the total number of examined papers. This is a further indication of the need for more proper experimentation in the software development area. The papers were analyzed with regard to topics, subjects, tasks and environment and presented quantitatively in tables and graphs. Surprisingly many papers do not report on important aspects that might have affected the outcome of the experiments such as internal and external validity. The article shows that experiments in some areas such as inspections are more popular than other and that students make up 87 % of the subject population used in the experiments.
2.5 Data Gathering Techniques

When performing empirical evaluations, there exists a number of techniques for gathering data, some suitable for most types of evaluations and some suitable for a specific approach. The list below presents some of the most common techniques.

- Direct observation - Direct observation implies that a researcher is observing an activity as it takes place. It is a useful technique when investigating work flows, group dynamics and similar activities where it might be difficult for the participants to remember and express correctly how they perceived the situation afterwards. Observation can also be used for evaluating a new tool or method to gain input for further improvements to the tool.

- Indirect observation - Indirect observation is done live using sensors to record activities or processes taking place. It is a useful technique when many different sources of data are to be observed simultaneously or when it is impossible to observe directly. Recording test subjects time consumption using automatic tools is one example of indirect observation, automatic logging of activities is another.

- Questionnaires - Questionnaires can either be in written or electronic form and are suitable when opinions from human participants are to be investigated. Questionnaires are often used in software development to assess background information and similar personal information, but also to obtain information on problems with the evaluation.

- Interviews - Interviews is an important approach in all sciences where humans and their opinions and feelings are the main focus. In software development, interviews are not that common as a stand alone method, but are often used as a complement to experiments and case studies. Interviews can only capture the subjective views of the participants but in many cases that is enough, for instance in evaluations of ergonomy, work climate, acceptance of new methods and similar.

- Participatory studies - A participatory study is a study where the researcher actively participates in the context being studied. This is sometimes done openly such that the other participants are aware of themselves being studied and sometimes done secretly in order to not disrupt the normal way of working or thinking. This approach is commonly used in the social sciences where people and their behavior are studied, but is also well suited for studying software developers in their normal context.
CHAPTER 2. EMPIRICAL METHODS

• Post-mortem analysis - Post-mortem analysis of data is done when the need for scrutinizing existing historical data arises. It can of course be argued that all data analysis per definition is post-mortem, but what is meant in this context is that no preparations or manipulations are done prior or during the creation of the data being analyzed. Only already existing data is used. Using error reports from finished projects to trace causes of errors is an example of post-mortem analysis.

• Literature studies - Literature studies implies reading existing books, articles and theses to obtain an understanding of a certain area or problem. The benefits of doing a literature study is that it is low-cost and that it does provide a well-prepared path into the area of interest. Literature studies is commonly used for performing surveys on trends within a scientific field, besides serving as a base for any research activity.

A common approach for an empirical evaluation is to combine a number of data gathering techniques, using for instance interviews for background information followed by indirect observation for the main evaluation.

2.6 Summary

Experiments, case studies and surveys are three major methods for performing empirical evaluations. They all have their benefits and drawbacks, both in terms of where they are most suitable, but also in terms of validity threats to the methods. What is important when planning an empirical evaluation is to carefully select the appropriate method for a given situation. It is further important to construct and execute the evaluation as thoroughly as possible, keeping the validity threats in mind at all times.

When executing an empirical evaluation, there exist a number of data gathering techniques that can be used. Table 2.1 below shows the possible combinations of methods and techniques.

As can be seen in the table, most of the techniques can be used for gathering data in all the three empirical methods. What technique to use in a given situation depends on the kind of data to be gathered, if the evaluation involves humans and most importantly what techniques are possible in that situation. A thorough empirical evaluation often employs different techniques for different parts of the evaluation.
<table>
<thead>
<tr>
<th>Data gathering technique</th>
<th>Empirical Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Experiments</td>
</tr>
<tr>
<td>Direct observation</td>
<td>X</td>
</tr>
<tr>
<td>Indirect observation</td>
<td>X</td>
</tr>
<tr>
<td>Questionnaires</td>
<td>X</td>
</tr>
<tr>
<td>Interviews</td>
<td>X</td>
</tr>
<tr>
<td>Participatory studies</td>
<td>X</td>
</tr>
<tr>
<td>Post-mortem analysis</td>
<td>X</td>
</tr>
<tr>
<td>Literature studies</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Methodology

Every scientist follows a scientific methodology when progressing in his/her work. Describing the methodology makes it easier for the readers to assess the quality of the work and to better understand the results, set in their context. A methodology description also provides input for future endeavors in that it shows which strategies have been successful and which have not. This section briefly describes the different methods that have been used in the work leading up to this thesis. More details on methodological issues can be found throughout the thesis in conjunction with the presentations of the evaluations. Although the main focus of this thesis is empirical evaluations in software development, it also contributes to the area of methods for handling semantics. The focus of all evaluations has been semantic aspects and a method for handling structured semantics in industry is also presented. The focus on semantic aspects is reflected in the initial hypothesis, the selection of case studies and the selection of treatments in the evaluations.

3.1 Initial Hypothesis

The starting point for the entire work process reported in this thesis was a common idea among a group of computer science teachers of what good software development was. This idea was built on teaching experience, programming experience and studies of current literature in the field. At the time, we did not know exactly what methods were available or how industry handled their software development. We did have a feeling that the semantic area of software development was rather immature. The first step in the process included gathering of information regarding semantics and their practical applications. A number of promising methods and tools for handling semantic aspects did exist and a few of these had been used in industry. It seemed, however, that
most of the methods and tools had not been empirically tested in a statistically sound manner. They seemed to work nevertheless and were advocated by the constructors of the respective method or tool. *Design by Contract* or as it was known at the time, *Programming by Contract*, coined by Bertrand Meyer [44, 46] in the late 1980’s seemed like a very promising approach. It was, however, perceived by students as rather complex and formal and also relied quite heavily on the programming language Eiffel [47]. Since the students found the method difficult, it was likely that industry people would agree, although perhaps to a lesser extent. Our hypothesis was that an increased focus on semantic aspects in software development in industry would be beneficial in terms of time and quality. More information on how industry worked was however needed to verify this. The process described so far would scientifically qualify as hypothesis and theory building. The informal literature study performed as part of this initial work has continued all through the entire research process and was formalized at a later stage. The results from that formalized literature study is presented in Sect. 4.3.

### 3.2 Industrial Case Study 1

The first step taken to verify our initial hypothesis was an investigation on how semantics were handled in industry. This was done using inspection of software artefacts such as code and documentation combined with interviews of software engineers. Inspection is common in software engineering and is a powerful tool, given a reasonably competent reviewer. The combination of inspections with interviews enables the researchers to better understand why the artefacts are constructed in a particular way, something that a pure inspection might not encompass. Interviews also educate the researchers in the area in which the project is set, which makes for further understanding and proper interpretation of the data. Since an additional aim of this particular investigation was to help the industry partners improve their semantic work, feedback was provided both informally and formally. This was done by reporting verbally about our findings as well as providing them with the interpreted, written results. The results indicated that semantic aspects were not always handled adequately and that hence more support in this field would be helpful. This step qualifies scientifically as a case study.

### 3.3 Theory Building

The information gained by the case study served as input when designing and developing a method that would try and remedy the problems found in the study. The aim of the method was to help software developers incorporate
3.4. CONTROLLED EXPERIMENT 1

a semantic awareness into their development work. This method was called Semla and is further described in Chapt. 5 and can also be found in full in Appendix A. Semla is based on previous methods, especially that proposed in [46], but was tailored specifically for industrial needs and conditions. A scientific classification of the work leading up to this method would best be described as theory building using an abductive approach which means that both previous theoretical knowledge and knowledge gained from empirical investigations are combined to form the new theory.

3.4 Controlled Experiment 1

To verify the merits of any theory or theoretical construct, empirical proof is needed. We had indications of deficiencies in semantics handling in industry and a method we believed would remedy the problem. To verify if the method would solve the deficiencies we needed empirical data. The first attempt at verifying the method was done in a controlled experiment in a university setting, using computer science students as test subjects. The experimental design of this initial experiment was rather simple, a comparison between groups working with the method and groups working with a base-line method. The subjects were to complete a large assignment in groups of six people as quickly or efficiently as possible. Throughout the process of working with the assignment, the test subjects reported on their time consumption for different phases in the process. We managed to find some indications that the method would serve its purpose, but unfortunately no strong conclusions could be drawn. One positive note was that the method seemed at least as good as the baseline method, thus providing enough confidence to move on to industrial testing. If the results had shown that the method would demand more time or be perceived as more difficult than a baseline method, further experiments using this particular method would not have been planned.

The experimental setup used where groups subjected to different treatments are compared also has certain drawbacks, such as the need for a large number of groups and the difficulties in handling differences in test subjects. The experimental setup was subsequently changed in a later experiment. The scientific classification of this step would be a between-groups controlled experiment.

3.5 Theory Revision

Before continuing to industrial testing of the method, it was rewritten and tailored in cooperation with a representative from our industry partner to better suit industrial needs. The method underwent several revisions and program language changes before ending up supporting Java in the version being used in
the industrial experiment presented next. A scientific classification of this work would again be theory building using an abductive approach.

3.6 Industrial Case Study 2

After the controlled experiment, a full-scale industrial case study was performed to evaluate how the method worked in live projects. The initial aim was to compare a number of variables in the project being studied with previous similar projects, but due to a number of factors this turned out to be more difficult than expected. We were not able to secure any quantitative data from neither previous projects, nor the one under scrutiny, because data was only logged for other purposes such as billing. No data on efficiency, time consumption in relation to productivity or other aspects we were interested in, were logged. The case study was further disturbed by contaminating factors such as changes in the working environment, change of programming language and addition of new personnel. The project assigned to us also turned out to be too big to control and hence some participants avoided our method, some attempted to use it, but failed and some used it as intended. This made any comparison difficult since only parts of the project were affected by the method. In spite of all the afore mentioned problems, we were still able to obtain qualitative data from the project on how the project members perceived using the method and how they believed it affected the project. More specifically we measured the project members’ attitudes towards the method and how they believed it would affect the quality of the resulting product. A scientific classification of this step is case study.

3.7 Controlled Experiment 2

The last step in the data acquiring process was a controlled experiment where four different methods for handling semantics were compared using a factorial design. The reason for using four methods rather than only two, as in previous experiments was to widen the scope and thus covering more ground in the experiment. The aim was to see which of the four methods was the most effective in terms of development time in relation to semantic quality. Students were once again used as test subjects and were to complete four roughly equivalent assignments, using the four methods one at a time. Time consumption to complete the assignments was recorded. Since the computer science community has yet to reach consensus on what metrics that properly assess the quality of a piece of software, we decided to set a basic level of quality for the assignments and have the test subjects quit working when this level was reached. The quality level basically indicated whether the software worked or not, the most
important metric of all since all other metrics matter little if the software is non-functional. The factorial experimental setup turned out to be very usable in an experiment like this, when the number of test subjects is low and the number of factors to control and evaluate is high. The scientific classification of this step is a controlled experiment using a full factorial design.
Chapter 4

Semantic Aspects in Software Development

This chapter presents two evaluations of handling of semantic aspects, one literature survey of academic work and a case study of industrial work. The industrial case study is studied from two different perspectives, first an overview of semantics handling in the entire project, and second a more detailed study of a smaller part where some improvement suggestions are also given. The chapter hence provides some background into how semantics are handled in the two different worlds of academia and industry. It provides the reader with an understanding of the difference in how semantics are handled and also helps the reader appreciate the gap between the two worlds. This gap is what lead us to develop a method that would try to raise the level of semantic proficiency in industry and introduce some of the academic ideas into the industrial world.

The remainder of the chapter is structured as follows: First, background information and some terms and definitions are given in Sections 4.1 and 4.2, followed by a section (4.3) on how semantics are reported in the literature. After that, the two studies on the industrial case are presented to give examples on how semantics are handled in industry. The first study, presented in Sect. 4.4, focuses on general semantic aspects whereas the second study, presented in Sect. 4.5 focuses on a particular construct frequently used in the project. This construct is discussed with regards to the semantic implications of the construct as well as how it is possible to improve the semantic quality using some of the techniques outlined in Sect. 4.3.
CHAPTER 4. SEMANTIC ASPECTS IN SOFTWARE...

4.1 Background

Software development can roughly be said to take place on three different levels, organizational/process-oriented, semantic-oriented and syntax-oriented. In order for any software development project to succeed, the work on all three levels must be satisfactory at the least. The lowest level, handling syntax-oriented questions is per definition the most important one. Without code there is no semantics and there is certainly no processes or organizations needed to produce code. The syntactic level only handles language and language-specific issues. As soon as code is constructed, however, semantic aspects emerge\(^1\).

Our definition of semantics includes everything a software developer wants to express in a programming language. It ranges from simple translations from the real world to the computer world to elaborate interpretations of the state of a program. As for the third level in the model, the organizational/process-oriented level, the main focus is to arrange and organize the supporting environment of a software development project. This level handles how to work, how to organize the work and what organization to have to succeed in software development.

As said before, all three levels must be at least satisfactory for any project to be successful. This has lead researchers and practitioners to provide methods and tools to support all three levels. Since the lowest level is the most important and has existed for the longest time, the method and tool support on this level is very good. Compilers, code-checkers, editors and new languages are some examples of tools. The highest level has also received much attention in recent and not-so-recent years. CMM [59], ISO9000 [31], RUP [37] and UP [32] are examples of process and organization oriented methods. The semantic level in between has also had its share of contributions, but to a lesser extent than the two other levels. This might be because both the lowest and the highest level touch upon it, thus implying that this level is handled satisfactorily. Examples of methods and tools that handle semantics are programming by contract [48], UML [32], Javadoc [74] and OCL [57].

4.2 Terms and Definitions

This section presents some terms and definitions used in the remainder of the thesis. As in most scientific fields, terms have different meanings, depending on subfield jargon and tradition and this section tries to formulate the author’s view on some terms. The definitions given below are believed to conform to the terminology most often used in software development literature [11, 80].

**Precondition** - A precondition is a condition that must hold before a call to an operation or part of a system. It is often possible to express this con-\(^1\)

---

\(^1\)Barring the extreme cases in which code is written without purpose. With purpose come semantics.
4.2. A TAXONOMY OF SEMANTIC SPECIFICATION...

dition in a formal language or as program language statements, but some
preconditions have to be expressed in plain text, typically non-functional
conditions. Many techniques for specifying preconditions only allow exe-
cutable conditions [47, 36, 33], whereas the author believes that conditions
not possible to express executably also have to be catered for.

**Postcondition** - A postcondition is a condition that must hold at completion
of an operation or part of a system. Like preconditions, they can be
expressed formally or in plain text.

**Contract** - A contract is the combination of a precondition, a postcondition
and a mutual agreement by the provider and customer to abide by the
conditions. Contract-based programming is one approach for helping the
program developer focus on semantic aspects in software development and
is based on the well-known Hoare-triples of the early seventies [29]. It fo-
cuses on good semantic descriptions in the form of pre- and postconditions
for operations and in the form of invariants for data objects as well as loop
variants and -invariants. The actual term contract was introduced and ad-
vocated by Bertrand Meyer in [45, 46, 48] and has since been adopted to a
certain extent by different description techniques such as Z [72] and UML
[67]. There is also a number of environments supporting working with
contracts such as iTContraction [36], jContractor [33], Jass [2], Handshake [18]
and Biscotti [14].

**Interface** - An interface is the border that provides the access points to a piece
of software. In its most primitive form, an interface is only comprised of
a number of operation signatures. In more elaborate forms, an interface
includes the signatures, contracts for the operations, invariants for the
software piece, documentation and rules for use.

**Semantic Integrity** - The semantic integrity of a software system is the degree
to which its semantic properties are preserved. It can be explained by
stating that each part of the system should respect the intended purpose
of all the other parts. It is essential for any stable system that its semantic
integrity is maintained. This requires that each part is clearly described
and that the description remains valid as the part evolves. If the semantic
integrity is violated, the system will enter an unstable state, which may
eventually lead to some kind of malfunction.
4.3 A Taxonomy of Semantic Specification Techniques

This section presents a literature survey on how semantics are treated in the literature. More specifically, the focus is on module and function descriptions and what methods are available to handle the semantic issues that modules and functions possess. First, some software issues where semantics play an important role are presented. The issues discussed are specification levels, weak and strong contracts, and required and provided interfaces. Following that, five levels of formalism for the specification of semantics are presented and related to current specification techniques. The levels are denoted no semantics, intuitive semantics, structured semantics, executable semantics and formal semantics. After that, three phases in the life of a software module are highlighted and the semantic issues that characterize each phase are presented. The phases are the creation phase, the use phase and the maintenance phase. The levels of formalism and the life cycle phases are then combined into a taxonomy, which can be used to identify different approaches to semantic specifications.

4.3.1 General Issues of Semantic Concern

This subsection discusses three general issues that affect the semantic integrity of a system. The three issues are specification levels, weak and strong contracts, and required and provided interfaces.

Specification Levels

The properties to be specified for a module may be divided into the four levels of syntax, behavior, synchronization and quality of service [3]. Level 1 includes specifications on the programming language level and is not further dealt with. Level 2 relates to semantic aspects [46] and is the main focus of this presentation. Level 3 describes the dependencies between services provided by a module, such as sequence, parallelism or shuffle. Some techniques to manage these issues are discussed in [3, 46, 81] and are not pursued further, although the structural framework presented here supports this level. The same applies for level 4, which deals with quality of service, such as maximum and average response delay, precision of the result and throughput for data streams [64].

Weak and Strong Contracts

The semantics of an operation can be described with a contract, as described earlier. When an operation can only succeed under specific conditions, these conditions are included in the precondition, and the postcondition needs only to specify the outcome in those well-defined situations. This may be called a strong
4.3. A TAXONOMY OF SEMANTIC SPECIFICATION...

contract [55]. However, a module that is directly accessible to human users, for example, should not have preconditions since humans cannot be expected to check conditions at all times. Instead, such a module should take care of filtering away invalid uses in order to meet the preconditions of any internal modules it uses [49]. The postcondition for such a user interface module will then specify the outcome of the invalid uses as well. Such contracts may be called weak. A module with strong contracts may be replaced by one with corresponding weak contracts without affecting its clients [39]. Architectures with front-end modules with weak contracts and back-end modules with strong contracts are common, as for instance in [4]. By making the distinction between strong and weak contracts explicit, one may more easily profit from their properties.

Required and Provided Interfaces

To be composable solely on the basis of its specification, a module must be equipped with explicit declarations of both required and provided properties, such as functionality, synchronization and quality [76]. A module can be used in a given architectural environment only if its required properties are satisfied by the suppliers in the environment and its provided properties satisfy the requirements of the clients in the environment. From a contractual point of view, a supplier interface S satisfies the demands of a client environment C if the contracts required by C are the same as or stronger than those provided by S [54, 55]. This matching must be done for a module’s required interfaces as well as its provided interfaces. The rest of this chapter does not stress the fact that a module is playing a dual role as both client and supplier, as both the required and the provided interfaces are covered by the general considerations. The focus of the presentation will be on provided interfaces, since that is what a module designer can affect most easily.

4.3.2 Levels of Formalism for Semantic Specifications

A module can be described with different levels of formalism. Different environments and developer preferences as well as the maturity of the developers influence the choice of the level of formalism. The semantic awareness of software designers varies as does their ability and willingness to formalize the semantic specification. We have through a literature survey [7] identified five levels of formalism of semantic issues. These are used to structure this section and in the taxonomy presented in 4.3.4. The levels of formalism can be defined in increasing order of formalism by the key words “No semantics”, “Intuitive semantics”, “Structured semantics”, “Executable semantics” and “Formal semantics”. For each level, we select a technique available on that level and use it to describe an example operation. Based on practical experience, the descriptions given are believed to be representative for the different levels of formalism.
CHAPTER 4. SEMANTIC ASPECTS IN SOFTWARE...

An Example

A complete semantic description of a module should include an overall description, showing its general properties, module invariants and a specification for each public operation. In this presentation, however, we will use a running example with a focus on an individual operation rather than on an entire module. The example is a module controlling the access to a random access file of a record type R. This file access module will be called RandomAccess. It may for instance be used by a database management system to access the physical file structure. The physical file controlled by the module contains records of a fixed size, and the access to the file is by record number, counted from zero. It is assumed that the file is continuous, so that all records up to and including a current maximum number, called the high water mark, are in use. The module defines operations for adding and removing records to and from the physical file, for querying the current high water mark and for retrieving and updating a particular record.

In our example, we will assume that the file is already populated and concentrate on the operation getRecord to retrieve a given record. The precondition for this interface contract is that the single input parameter of the operation is the number of the record concerned, which must exist in the file. The postcondition is that the result of the operation is the required data record of type R. If an unrecoverable system error occurs, such as a disk read error or some other file system error, the operation indicates a failure, so this part of the contract is weak. However, no indication will be provided about possibly unsatisfied preconditions, for example a request for a record number that is too high. This strong part of the contract simply assumes that this does not happen [49].

The remainder of this subsection describes this example operation for each of the five levels of formalism. Where relevant, we will use Java as the specification and implementation language, with the obvious assumptions about class, operation and type names.

No Semantics

The focus of the “No semantics” level is exclusively on the syntactic parts of the interfaces, represented by CORBA or COM IDL, Java interfaces or similar interface description languages. They represent the minimum level of specification needed to connect modules. Operations are not explained, except that the names used may be chosen to be as self-explanatory as possible. We have included this level only for the sake of completeness, as it is not concerned with semantics at all. The interpretation is left to the reader’s intuition. The definition of the operation getRecord may for example be defined as in Fig. 4.1. Nothing is said about when number is a legal parameter, the interpretation of number or the resulting R object or the cases in which an IOException is
4.3. A TAXONOMY OF SEMANTIC SPECIFICATION...

```java
public R getRecord (int number)
throws IOException
```

Figure 4.1: A purely syntactic specification of getRecord

thrown. There is plenty of room for free guesses. Most would agree that, with this level of semantic specifications, a commercial software module would not be trustworthy [76].

**Intuitive Semantics**

By “intuitive semantics” we mean plain text, unstructured descriptions and comments about a module and its parts, as illustrated in Fig. 4.2. The description has no particular structure, and no guidelines are provided to determine the kind of information it should contain or how the information should be presented. Lacking such structure for the specification, the designer of the module must rely on his/her intuition and experience when describing its semantic properties. The description in Fig. 4.2 has two serious drawbacks. First, the description does not give the conditions for using the operation correctly. Second, this kind of description is unstructured and the information it contains is scattered and arbitrary. Both these factors make this kind of description difficult to use.

As will be elaborated in Sections 4.4 and 4.5, an investigation of two software engineering projects showed that the semantic aspects were largely based on intuitive reasoning of the same kind as in Fig. 4.2. We believe that a large portion of the noncritical software being developed today falls into this category, i.e. the semantics are mentioned and intuitively described, but without structure and consistency. Compared with what we have seen elsewhere, Fig. 4.2 shows a reasonably good intuitive description. The next level of formalism may be a step forward for designers at this level [76].

```text
The operation getRecord retrieves a record by its number, returning the record requested. If an error occurs, such as a disk read error or a file system error, then an I/O error is returned
```

Figure 4.2: An intuitive specification of getRecord
Structured Semantics

“Structured semantics” are produced by designers and engineers who are highly aware of the semantic implications and requirements of their modules but who are reluctant to express the specifications in too formal a way. Extra cost and competence requirements for present and future users of the specification are examples of valid reasons for such a choice, as is the fact that many requirements are difficult to formalize. The semantics are presented in a structured way but need not be in accordance with any particular syntax or formalism. In Fig. 4.3, the semantic conditions are expressed as plain text contracts [17, 46], which may be inserted as comments in the design documentation, in the interface descriptions or in the code itself. The major focus of structured semantics is on a conscious, structured approach to semantic questions without being too focused on the formalities or executable test mechanisms. One way to structure the information is to begin with a short description of the operation and its parameters, followed by the relevant preconditions and postconditions, as in Fig. 4.3. The postcondition is only specified for cases in which the precondition is satisfied. This structured description is easier to read than the intuitive description in Fig. 4.2, even if it may contain the same information. The fact that the same structure applies to the specification of all the operations also reduces the risk of leaving out important information. It is however necessary for the designer to understand the implications of the different parts of the specification. Case studies (see Sections 4.4 and 4.5) and teaching experience have shown that pre and postconditions are sometimes used in specifications but that their implications have not been understood, so that the conditions set up were meaningless or even wrong.

A structured approach without formal requirements may be a useful improvement over a purely intuitive handling of semantics. It supports two major aspects of the development work. First, formulating the semantics in a structured, textual form stimulates extra thinking and favors a better design. Second, the structured documentation makes reading and understanding the specifi-
tion easier for client programmers as well as for module implementers. This kind of specification can also handle non-executable conditions [35], which set a limit to the applicability of the next level of formalism, the executable semantics.

**Executable Semantics**

With the term “executable semantics” we mean that semantic aspects are expressed in a way that can be executed and controlled by the system during runtime. As such, the executable specification for an operation is included in the implementation of the module. It must therefore be expressed using some executable syntax in the implementation language. The specification of `getRecord` for Java may for instance be expressed as in Fig. 4.4. We assume that `hwm()` returns the current high water mark, `result` is an implicitly declared variable representing the result of the function and `record()` is a primitive function returning a record from the file. The contract in the example says that, provided number is between zero and hwm, the requested record is returned unless an irrecoverable I/O error occurs, which is signaled through an exception. The contract does not state the result if the number is not in the required interval. That situation corresponds to a coding error and a violation of the semantic integrity of the system. It is not covered by the strong contract used. An I/O error, on the other hand, corresponds to a failure outside the control of the software and must be caught in a weak contract. With this formalism, assers-

```
getRecord returns a record identified by its number.
Parameters:
number: the number of the record to retrieve, counted from zero
Precondition: (0 <= number) && (number <= hwm())
Postcondition: throw IOException || (result == record(number))
```

Figure 4.4: An executable specification of getRecord

Tions may be used to express preconditions and postconditions and for testing them during run-time. The most obvious candidate to be tested is the precondition. Bertrand Meyer is a strong advocate of this approach, which is integrated in the Eiffel language [46]. Another example is Biscotti, an extension of Java that enhances Java remote method invocations interfaces with Eiffel-style preconditions, postconditions and invariants [14]. A third example is OCL [58], a specification language developed in the framework of UML where conditions are compiled into executable statements. Other examples include Jass [2], iContract [36], an approach by Guerreiro [27] and jContractor [33]. The execution of the assertions should not add functionality to the module but serve only to detect possible violations of the conditions defined. It should be stressed that
these assertions serve to detect coding errors and should not try to handle or compensate for them [46]. Since these assertions might not be turned on at all times, the behavior of the software system should not be affected by whether or not they are actually executed. In fact, OCL only defines inquiry operations without side effects. OCL statements cannot be used for data processing or for corrective actions. What to do when an assertion does not hold is a robustness issue and not a correctness issue. A broken assertion is a sign of a coding error and a violation of the semantic integrity. It is not discussed in this chapter, where the focus is on how to avoid breaking the semantic integrity, not on how to repair it once it is broken.

For the purpose of the example, we will assume that the `System` class in Java contains an assert method that can be used to execute the assertions. It takes one Boolean argument. If the value of the argument is true, the method returns without any visible effect. If the value of the argument is false, the method terminates the program, possibly leaving some suitable tracing information to help detect the offending code. For debugging purposes, the module itself may then use the executable precondition to trap offending calls, as shown in Fig. 4.5 [49]. It may be worth mentioning again that this assertion is not a mechanism for handling errors, only for detecting them. If the logical expression \((0 \leq number) \&\& (number \leq hwm())\) has the value `false`, then the method `System.assert` does not return. Instead, since this condition identifies a situation in which the outcome of a call to `getRecord` is not defined, the program is typically terminated. This is especially useful for detecting errors early in the development and testing of client systems. More on the handling of different kinds of errors can be found in [56], where the relation between different kinds of contracts and different kinds of errors is discussed.

The client code may also take advantage of the executable assertions by checking the pre-condition before the call, as in Fig. 4.6. We assume that the client has a variable `theFile` of class `RandomAccess` and a variable `record` of class `R`. In addition to the benefits for design and documentation given by the structured semantics approach, the executable specification supports the efforts to maintain semantic integrity by early error detection and removal. However, this benefit is limited by the fact that not all conditions can be expressed in an

```java
public R getRecord(int number) throws IOException
{
    System.assert((0 <= number) && (number <= hwm()));
    // the implementation of the method
}
```

Figure 4.5: Use of executable precondition in a supplier module to trap offending calls
if ((0 <= number) && (number <= theFile.hwm()))
{
    try {
        record = theFile.getRecord(number);
        // record == the record requested
    }
    catch (IOException e)
    {
        unrecoverable IO error
    }
}

Figure 4.6: Use of executable semantics in a client module to ensure a correct call

 executable way. This is illustrated by the postcondition in Fig. 4.4, where the fact that an exception may be thrown is expressed only as a comment.

Formal Semantics

With “Formal semantics”, programs can be proven to have consistent and sound semantics [23, 50, 66]. Formal specification languages such as VDM, Z and \( \lambda \) are well known. We will use Z to express the formal specification of getRecord. The concept for the Z description is taken from [72] but modified to suit our purpose. For further details regarding the meaning of terms and symbols used, the reader is referred to literature on Z, such as [16, 60, 72].

The visible state of the random access module is defined in a state schema called RandomAccess, shown in Fig. 4.7. The term records represents all the records in the file and \( R \) is the record data type. The variable hwm (for ‘high water mark’) shows how much of the file is in use.

\[
\begin{align*}
\text{RandomAccess} \\
\text{records} : \mathbb{N} \rightarrow R \\
hwm : \mathbb{N} \\
\forall i : 0..hwm \bullet \{ \text{records}(i) \} \neq \emptyset
\end{align*}
\]

Figure 4.7: Z state schema for the random access file

The formula \( \forall i : 0..hwm \bullet \{ \text{records}(i) \} \neq \emptyset \) in the lower part of the schema expresses the invariant for the module, which is that all the records from number
zero up to and including the high water mark are in use, that is, the file has no “holes”.

The file operation is defined as a state schema called getRecord, shown in Fig. 4.8. The upper part specifies the parameters of the inquiry and the lower part expresses its pre and postconditions. In agreement with the contract used, no return value is defined if the pre-condition \( \text{number?} \leq \text{hwm} \) is not satisfied. Similarly, also in accordance with the contract, the return value of \( \text{record} \) is undefined if a file system error occurs, so no record value is specified for that case.

A formal approach is valuable for security critical applications. Büchi [13] argues that such an approach will simplify the process of formal reasoning of modules and make it easier to compose modules based on their contracts. However, the rigid formalism restricts the application of their proposed ideas to environments in which the overhead of formal methods can be tolerated. The major part of the market may find the formal approach too costly or too difficult to access.

4.3.3 Phases in a Module’s Life

During its lifetime, a module passes through different phases. This section considers the creation phase, use phase and maintenance phase from a semantic point of view. The creation phase of a module includes its design and implementation, both for the initial creation and for subsequent additions of new functionality. This phase corresponds roughly to the traditional development cycle. The use phase includes all regular use, both initial use and reuse. The maintenance phase includes all changes to the existing functionality after release, including versioning. All these phases may recur without any particular order, except for the initial creation phase. In particular, use and maintenance
of a module are often closely related. As a module is being used, problems and deficiencies are detected and corrected. For the following discussion on semantic considerations, it is assumed that the operation `getRecord` is specified with the structured contract of Fig. 4.3.

The creation phase

The first creation phase begins when the need for a module is identified. Further creation phases frequently occur later in the module’s life, when new functionality is identified and added to the module. During a creation phase, the semantic focus is on design and implementation.

The design of a module defines its required and provided functional and non-functional characteristics. We have already stressed how semantic integrity depends on good semantic specifications and that it is not sufficient to have a syntactic description. The semantic part of the design includes invariants for the module and required and provided contracts with pre and postconditions for each operation. Implementation details should be kept out of this description.

Often, the client programmer will not see any code for the interfaces provided by the module, nor will the module designer see the code for the required interfaces, so the semantic description must express all the information about the module needed by the user and its required and provided interfaces. Even if the code was available to the developer, it can only show what the current implementation happens to be and not the purpose of the operation. This becomes evident in the maintenance phase when the code is modified, as described below. Any client program based on the current implementation instead of on the specification will be endangered by any such modification.

Once designed, the module will be implemented. The contracts and invariants should be implemented in a consistent way. Since the code will not, and should not, be available to the client programmer, no new conditions should be introduced during implementation without also being reflected in the design specification.

The use phase

A module enters the use phase when it is employed as a service provider by some client software. The main semantic issue for the client programmer is to determine the semantic specification of the module and to use it accordingly. If a contract required by the client is the same as or stronger than a contract provided by the module, the operation may be used directly [54, 55]. In other cases, it will be necessary to create an adapting shell around the module, where the interface defined by the shell is better adapted to the client requirements than is the module itself [19].
A module is sometimes developed for use in a particular context and is normally used in this context first. At the same time, one objective is often to produce reusable software and the developer often knows little about the different contexts in which the module will be used. In both cases, the rules for use are the same. To ensure the semantic integrity of the system, the module should only be used as defined in its semantic specification, and the specification must thus be complete and consistent. If the description is unstructured and intuitive, interpretation problems easily arise and trial and error may be the only way of determining how the module behaves.

Testing is sometimes advocated as the major quality assurance mechanism for modules [83]. If the module to be used is not adequately described, testing may be the only practical approach, although it is very resource consuming. However, basing the use of a piece of software on its empirical behavior may cause problems during maintenance, as will be discussed next. Sometimes, as in the case of the French rocket Ariane 5 [38], testing is explicitly excluded because of the high costs. In any case, the use of testing does not contradict the usefulness of thorough planning and description of the semantic aspects of a module. On the contrary, such a description can serve to reduce the number of errors and thereby speed up the testing process.

The maintenance phase

Maintenance is the work performed on a module after its deployment in order to correct errors, improve performance or other properties and adapt it to changes in the environment [77]. The main semantic issue here is to maintain backward compatibility and to detect when it is discontinued. Addition of new functionality is not classified as maintenance but is treated in the same way as the creation phase.

Since the module is identified externally by its semantic descriptions only, any modification to the module that respects these is unproblematic. It is important however to check that a modification does not in fact violate the description.

If the contract of an operation is changed such that demands on the clients are changed, problems may arise. A change in a module is sometimes harmless to client systems and sometimes harmful. Liskov’s substitutability principle [39] can be applied to determine which of the two is the case. It implies that a change is safe if the client cannot observe it. If the new contract is weaker than the original one, so as to demand less from the clients, no problems arise, since all existing clients already conform to the new specification [54, 55]. Otherwise, existing clients conforming to the first contract may suddenly violate the modified contract although the clients themselves have not changed. The modified operation should be recognized as a different one, for instance with a
new interface definition. This versioning issue is a serious source of problems when modules are replaced [75].

4.3.4 The Taxonomy

We now suggest a two-dimensional taxonomy for module semantics. One dimension encompasses the five levels of formalism and the other the three phases of a module’s life. This taxonomy may serve as a reference in discussions and presentations of modules and module semantics. A summary is also given of a survey of the state of the art in semantic management of software modules and is related to the taxonomy. Of the articles examined, we judged 37 publications to be relevant for our purposes. Fig. 4.9 shows how these publications are related to the taxonomy. Each reference in the survey is represented by an asterisk, and some of the references appear more than once. Each reference is related as truthfully as possible to the taxonomy according to its main approach to semantics. Of course, since the survey focuses on semantic management, it did not include publications in the “no semantics” column. The limited number of references found during the survey may be an indication that semantic issues are not actively discussed in these contexts. It was also found that authors have very different approaches to semantic aspects when they discuss modules and components. However, there is a consensus that there is a need for semantic descriptions of modules and that this need is not covered by the interface description languages as they stand.

Figure 4.9: A literature survey related to the proposed semantic taxonomy
Two interesting conclusions may be drawn from the distribution of the publications. The first is that most of the publications dealing with semantics are found in the area of executable semantics. This reflects a general view that exact semantics cannot be efficiently expressed in plain text [58]. However, our experience is that the state of the art among practitioners is chiefly at the intuitive level, as will be elaborated more in Sections 4.4 and 4.5. For practitioners who find themselves at this level, the structured level may be a first step towards a more organized approach to semantics. The structured approach gives good support to the design process itself as well as to the documentation, both of which are crucial factors, without requiring too much formalism. It may be applied even if the conditions in the contract are not executable.

The other interesting observation to be made in the diagram is that there is little discussion of maintenance issues. This lack of discussion is a serious threat to the long-term stability of any reuse-based system. We believe that it would be possible to establish a sound practice for maintenance by applying Liskov’s principle of substitutability [39] to modules with descriptions at the structured level of formalism or above. Using this principle will help to avoid the problem in which a new release of a module compromises the stability of complete systems.

4.3.5 Summary and Conclusions

This section has presented some important semantic issues, how semantic properties of software modules can be described with different levels of formalism, and the importance of semantic integrity in the different phases of a module’s life. Finally, a two-dimensional taxonomy for semantics was presented together with the results of a literature survey that shows how semantic aspects are handled in current research. One dimension in the taxonomy encompasses five levels of formalism in describing semantics, where the level chosen for a particular project depends on the application domain and the maturity of the designers. The other dimension covers three phases in a module’s life, where each phase requires certain semantic considerations. The taxonomy allows a designer to position himself with respect to his semantic awareness and to increase this awareness. The majority of the publications in the survey had a focus on intuitive or executable semantics in the creation and use stages. Few publications discuss the semantic issues involved in maintenance and even fewer discuss structured semantics.

Software developers who find themselves at the level of intuitive semantics can benefit from applying structured semantics for improved design and documentation. Executable semantics adds the potential for dynamic error detection. Formal semantics may be necessary for critical and semi critical applications, but are hardly motivated for the noncritical market. The main point is that, to maintain semantic integrity, the specification of required and provided proper-
ties should be complete and understandable, not only by the module developers but also by the users and by those maintaining it. For the component market in particular, where interfaces play a crucial part, a greater focus on semantics during maintenance should reduce the number of problems experienced in this area today.

4.4 An Industrial Case Study on Semantic Aspects

As presented in the previous section, academia has developed quite a few methods for handling semantics. In this section, we present a case study aimed at determining how semantic aspects are handled in industry. To judge this, a number of semantic quality criteria was established, based on common knowledge in academia in general and the criteria given by Bertrand Meyer [48] in particular. Meyer’s criteria are quite extensive and some of them were deemed as too complex to use for industrial purposes. We thus reduced them such that the remaining criteria were pragmatic, rather basic and possible to use in an industrial project at a reasonable cost. The criteria were divided into four different categories; general, interface, internal and external to cover different aspects of quality. These criteria were then used as a checklist when examining the data from the project under study.

4.4.1 General Criteria

This category includes criteria that are essential when defining modules and abstracting data and functions. The criteria in this category serve as help in the design phase of the program. Most of the criteria are today seen as standard procedure, but are nevertheless not always adhered to.

Modularity

Programs should be divided into modules that should be well defined, self-contained and logically clear. If the modularity criterion is satisfied, the modules can easily be combined. Modularity also increases the understanding of the program by the increased logic and the shorter "chunks" of code that are the results of a modular design.

Weak Coupling

Weak coupling means that the implementation interdependence between modules should be as low as possible. Modules should communicate with as few other modules as possible in order to reduce error propagation. The amount
of information to be exchanged between two modules should be as small as possible. From this point of view, the optimal module is a module with no connections at all. No errors would propagate anywhere and changes in the module would have no effect on other modules. This module would conform to the criterion very well, but would clearly be useless.

**Encapsulation of Data and Functions**

Data should be in the same context as the functions that operate on the data as well as with other related data. The context could either be the linguistic context or the way the data is viewed and used. If data and functions that have something in common are grouped together, it facilitates future addition of other functions or data in that context.

**Data and Function Hiding**

Data should only be accessible through a pre-defined interface. An interface is usually implemented as a set of public functions, i.e. functions exported to other modules. Functions that might put the data in unstable states if misused should be kept private and only be accessible through the interface. This makes it possible to control the access to the data, as only "legal" ways are possible. Hiding the data does not necessarily protect it in a secure manner since it is still possible to reach the data, but it makes the misuse of data harder. It is never possible to achieve total security in a system, but this criterion gives a means of increasing the security.

**Data and Function Abstraction**

The data should be clearly defined and abstracted to a level above the implementation level. If the abstraction is done properly it can serve as a base for the implementation, independently of the implementation language. On the contrary, if the abstraction is insufficient or even missing and the properties of the data elements are expressed with a certain language in mind, this blurs the comprehension of what the data should represent.

Each function in a module should perform a well-defined task, have a limited functionality and no side effects. In this way, they can be used in an intelligible manner without worries about potential errors due to hidden side effects.

### 4.4.2 Interface Criteria

An interface describes how modules can access each other. It is usually composed of a number of interface functions, which operate on data and non-interface functions. However, it is important to be aware that the interface
generally is an abstract construction and that the actual functions which constitute the interface for a certain module are implementations of the interface. More than one module might share the same interface, which stresses the point of defining the interface and the modules separately. An example of this is polymorphism where inherited objects have a common interface with different implementations of the functions in the interface.

**Function Description**

All functions constituting the interface should be so well described concerning their properties, input and output that it should be possible to use them without knowing their implementation. As long as the description holds, changes to the implementation of a function can be done without affecting the user of that function.

**Contracts**

Contracts are an essential part of ensuring proper usage of functions. Setting up a contract enforces that correct output is guaranteed, given that the input is correct. What is correct and what is not is defined in the pre- and postconditions. The difference between programming by contract and simply defining pre- and postconditions is that both the implementer and the user of a module have a mutual agreement to respect the rules specified in the pre- and postconditions (see Sections 4.4.3 and 4.4.4).

**4.4.3 Internal Criteria**

The internal level deals with matters concerning the internal design of a module. Since the module should conform to the interface, it is important to respect all conditions of the interface. It is also imperative to describe data in a clear way.

**Respecting the Contracts agreed upon**

The contracts specified (see Sect. 4.4.2) need to be respected when designing modules as well as when using the modules (see Sect. 4.4.4). If both the designer of the module and the person using it respect the contract, the program will become clean and non-redundant. The programmer designing the functions or modules has to make sure that the contracts specified are not violated. The output must conform to the postcondition of the module. The programmer also has to trust the contract to protect the module from improper usage and not perform any redundant tests on input parameters.
CHAPTER 4. SEMANTIC ASPECTS IN SOFTWARE...

Data Description

All data should be described in such a way that it is clear what each data element represents. The interpretation of the data element should also be described if that is not obvious. Invariants, i.e., conditions that must hold at all (stable) times of the execution should also be used to protect the data from misuse. The invariant might not hold inside a function, but it is the responsibility of the function to re-establish the invariant before completion.

4.4.4 External Criteria

The external level describes how to use a module seen as a black box. The criteria below are to be considered when combining different modules.

Respecting the Contracts agreed upon

When using an already defined module, the contracts specified for that module need to be respected in the same way the designer of the module has to respect them (see Sect. 4.4.3). If the user of a module does not comply with the contract for that module, the results are undefined. The user should also trust the module and not perform any further tests to verify conditions already met by the postcondition of the module.

Using Functions Properly

The user of a module should only use the functions defined in the module in the specified way and not exploit any side effects they might have. If side effects are used, they might disappear if the module is implemented differently and the user would no longer be able to use them.

4.4.5 Case Study Environment

A case study has been carried out with the intention to investigate to what extent the criteria presented earlier were satisfied or even considered. The case study was performed on a project for the public telephone network undertaken at Ericsson Telecom AB (TSAC) in Karlstad. TSAC saw this investigation as a way for them to further improve their program quality. The examination of the case study is organized around the quality criteria presented earlier in this section. The architecture of the examined system is described in the following section, followed by the actual evaluation of the criteria.
4.4.6 Case Study Project

The project examined at TSAC was a service for Ericsson’s Intelligent Networks platform (IN), which takes care of special telephone numbers such as Freephone (phone numbers where the owner of the number pays), Premium Rate (higher rate numbers) and Universal Access Number (e.g. emergency numbers like 112 in Europe). These three different kinds of numbers are all virtual and mapped to one or more actual numbers by the TSAC system. The system is built as a sequence of branches (features) where each branch takes its input from the preceding branch and passes its output to the next branch. The correct branch in each column is chosen by a switch (selector). A schematic model of the system is given in Fig. 4.10.

![Figure 4.10: The Ericsson Telecom model](image)

Due to the block structure of the system, a modular organization was expected. All modules on the same vertical orientation share both forward and backward interfaces. This implies that the interfaces must be well defined and conform to the criteria on interfaces as defined in Sect. 4.4.2.

The material used when examining the TSAC project can be found in [10, 41, 42, 43].

4.4.7 Examination of the Criteria

General

Modularity was used both on an overall level and on the implementation level. On the overall level the system is built by combining different modules as can be seen in Fig. 4.10, where the flow of control goes sequentially through the modules. Each such module performs a well-specified piece of work.

As far as we have seen, the data is not connected to the functions at all although an implicit mechanism for associating the data together with the operations is obviously used. The data is not hidden, but seems to be global, which is a violation of the criterion on Data and Function Hiding. There was no description of the data structure. Nevertheless, it was used in many of the
modules. It seemed that the programming environment handled the data and that the programmers could assume that it existed without knowing where and how.

Most modules have only two interfaces, one from the previous module and one to the next, which indicates that the Weak Coupling criterion is satisfied. However, there are exceptions to this rule which are not explicitly declared as interfaces, but which still constitute such a one. In an error situation all modules have the possibility to raise an exception, in which case the execution continues in a special exception handler. This extra interface is not described in the documentation.

**Interface**

The Function Description criterion is satisfied. The modules are well described and easy to understand. TSAC does not seem to have any problems when specifying the functionality of the modules. Since many modules share the same interfaces, the interfaces should have been clearly defined, separate from the modules, but this was not the case. The only indication that the modules actually share interfaces was given in the block diagrams of the system, which does not give much insight into the semantic aspects of the interfaces. The main weakness in this category is that the interfaces are not abstracted away from the different modules implementing them.

**Internal**

Since contracts are not used to any large extent, it is impossible to judge if they are respected and it is hard to find the connection to other modules in the description. The description of the data structures used is insufficient. This makes it hard to estimate the amount of information flowing between modules.

The only data described in the documentation are the temporary variables, which in fact are not temporary but rather seem to be parts of the main data structure. References and changes to parts of the data structure are often made without explaining the properties of the structure or how it is transported through the system. The structure is neither described on entrance to a module, nor on exit which makes it harder to make changes to modules without violating the semantic integrity of the data structure. The description of the data structure might be found in other documents in which case a reference to these other documents might be useful.

**External**

The interfaces are insufficiently described and they are not abstracted away from the implementation of the different modules that share interfaces. This
makes the usage of other modules harder and information on how to communicate must be obtained in other ways than through an interface description. Contracts are not used, even if some attempts have been made with describing certain in and out parameters. The description of input and output parameters does, however, only mention the basic properties of the parameters and not the semantic meaning. As no contracts are used, it is impossible to judge if they are respected. A person using a module must obtain the information on the communication interfaces in other ways than by reading the documentation. If the answers only reside in other programmers’ heads, this is a serious problem since personnel might come and go, whereas the documentation usually persists.

4.4.8 Example Application of Criteria

To show more clearly what changes would be needed to conform to the criteria, a detailed study of some aspects of one specific module was made. The module examined is the OCT (Ordinary Call Transfer) module [43], whose task it is to find a corresponding actual number (C-number) for a certain called special number (B-number) called from the actual number of the A subscriber (A-number). All special numbers such as a freephone number (020 in Sweden) are mapped to an actual number. The module utilizes a database to find the corresponding C-number for a given combination of A-number and B-number. Some numbers can only be dialed from within a certain area, which explains the need for the A-number as an input parameter. Before the OCT module is called, the A-number has been checked and is allowed to call the B-number. The description of input and output parameters and preconditions currently in the documentation looks as follows:

\[
\text{Input parameters : } - \\
\text{Output parameters : } C - \text{number} + \text{charging information}
\]

This means that there are no input parameters to the module, which implies that the output should be the same for all calls to the module. Since the module according to the specification should generate a C-number for a corresponding A- and B-number combination, this cannot be true. The above erroneous declaration expressed in standard algebraic terms is equal to:

\[
\text{precondition : true}\\
OCT : \emptyset \rightarrow C_{\text{number}}\\
\text{postcondition : } C_{\text{number}} \text{ loaded}
\]
A definition satisfying the criteria for contracts should look as follows:

\[
\text{precondition} : A_{\text{number}} \text{ is allowed to call the } B_{\text{number}}
\]

\[
OCT : A_{\text{number}} \times B_{\text{number}} \rightarrow C_{\text{number}}
\]

\[
\text{postcondition} : C_{\text{number}} \text{ loaded}
\]

This is a simple example, but the style is similar throughout the entire documentation. Since most semantic information about the module is missing, there must be some other way for programmers to understand what needs to be done, e.g. asking other programmers or looking in other documents. Obviously, this is not as effective as having the information in the documentation. It would be surprising if no errors could be traced back to these kinds of incomplete specifications. There are some end-user issues as well as low-level syntax described in the documentation. Even if these are correct, they are of little help to a programmer trying to implement or change a module. The attempts to describe the input and output parameters of modules are a step in the right direction, but still sub optimal since no semantic information is given.

4.4.9 Results from the Case Study

Mostly, the documentation is on an end-user level describing the basic functionality in plain English. Some parts are on the other hand unnecessarily technical, which gives the impression that the person documenting wanted to add a technical level, but did not know what to add. The way we have perceived the documentation is illustrated in fig 4.11.

![Diagram](image)

Figure 4.11: Documentation level

The parts currently in the documentation are obviously rather well understood by the programmer, and will most likely be documented correctly in
upcoming projects. There is a gap between the highest level of documentation and the lowest level. This gap should be filled in with the semantic meaning, i.e. what the module does and why it does it. Since this part is missing, it is hard to understand what the different modules do.

As interfaces are not clearly defined, it is difficult to know what to expect on entrance and exit from each module. In order to program properly, it is essential to include information on pre- and postconditions, i.e. information on the program state at entrance and exit of a module. If, for some reason, the specification of pre- and postconditions could not be done, at least input and output parameters should be specified. This provides some information on the properties of the module even if the semantic aspects are not fully covered this way. It is worth noting that pre and postconditions were sometimes used, but only sporadically and not always correctly.

4.4.10 Summary and Conclusions

Our conclusion is that semantic aspects of programming are not adequately addressed during program development in industry. Although it is impossible to draw any general conclusions from the evaluation of a single project, it might still give an indication of the current state in programming industry. A parallel study of a project in a different environment [5] showed the same lack of semantic documentation. That other project was more technically oriented whereas the project presented above was business oriented. Together, they cover many aspects of industrial programming. If they are representative of the entire industry, the conclusion is clear: companies do not properly address semantic integrity and semantic documentation. Two case studies do not alone, however, validate strong conclusions and the results must hence be seen only as indications rather than as strong proof. We still have reason to believe that the overall quality of the systems would improve if semantic aspects were better understood and considered during program development.

4.5 A Detailed Example from the Case Study

This section presents a detailed study performed on the same project used in Sect. 4.4, but with a focus on the use of contracts in conjunction with so called switching sections, that are described in more detail below. We discuss the switching sections to show how the module integration could be done and documented to achieve a higher level of software quality. In particular, we look at how the concept of contracts can be used to manage the semantics of a switching section and then generalize these principles.

The rest of this section is organized as follows. First, the concept of switching sections is presented, essential to the rest of the case study. Then, the scope
of the case study is outlined, followed by a short description of the structure and documentation of the module. After that, a discussion on how the design documentation supports the semantic integrity of the module is given. A design methodology is then presented that uses contracts and helps integrate the implementation of the switching sections to the overall module design. Finally, a discussion on the merits of the methodology is given followed by some conclusions.

4.5.1 Switching Sections

The following definitions introduce the particular focus of this investigation. We will use the term switching section for the special kind of code section examined. The first part of a switching section is called a switch. It performs a test and, according to the result of the test, one of the following parts, called branches, is executed. All the branches converge to a common continuation point, the convergence point. A switching section is commonly implemented as a possibly nested if-then-else statement, a switch or a case statement. As presented in Sect. 4.4, the switches were called selectors and the branches contained features in the corporate lingo. A switching section is illustrated in Fig. 4.12.

![Diagram of a switching section]

Figure 4.12: The structure of a switching section

The focus on switching sections was chosen because the system being studies consists of a sequence of such sections. We found a lack of semantic specification, called semantic gaps, both between and inside these sections. These semantic gaps represent a potential source of semantic errors as the system continues to evolve.

4.5.2 Scope

The material presented in this section is based on the same industrial case as presented in Sect. 4.4. The objective of the case study was to investigate the description of a module containing switching sections. We wanted to see whether
the description contained the appropriate information needed for proper usage. To use the module one should not need to go to the implementation to see what was actually done - the module specification should tell the whole story. Similarly, to implement the module, one should not need to go to its client modules to investigate what assumptions are made about it. Again, the module specification should give the complete description against which the implementation should be checked. We did not expect the interface to be described as a contract but wanted to see whether the ideas from the contracts concept were used. As it turned out, in addition to the design documentation of the main module, we also received a description of the module’s implementation. This includes the specifications of the next lower level of modules used to implement the main module. We thus had two levels of module description, one used by the implementation of the other. Actually, this allowed us to draw more conclusions than we had planned.

4.5.3 Structure and Documentation

This subsection presents the structure and the documentation of the studied modules. Some details have already been presented in Sect. 4.4. There are two levels of abstraction in the system. The study involved one software module which was implemented using several other, smaller modules. We will use the general attributes outer and inner to distinguish between them. Figure 4.13 shows the relationship between the outer and inner modules.

![Figure 4.13: The relationship between outer and inner modules](image)

The implementation structure of the outer module is illustrated in Fig. 4.14. It consists of a number of chained switching sections and a data structure. The convergence point of one switching section is the switch of the following one. Each of the branches is a separately developed module and represents a particular service. In a certain sense, the solution is dynamic. It is possible for the customer to add new branches for a switch, after the system is set in operation. This allows new services to be added dynamically. The outer module
also contains a local data structure. This data structure is directly accessible to all the inner modules and acts as a common global data structure for them.

![Diagram of the outer module and its overall implementation structure](image)

Figure 4.14: The outer module and its overall implementation structure

The outer module as a whole and its functionality are thoroughly described in a system requirements document. The input to the module and the output from it are well specified, as are the conditions governing when it can be called from its environment. These requirements are symbolized in Fig. 4.14 by the dashed lines, respectively labeled precondition and postcondition, crossing the arrows going into and out of the outer module. Figure 4.15 shows the chain of switching sections, which constitute the inner parts of the outer module. The input requirements for the outer module also serve as input requirements for the first switching section. Similarly, the output requirements for the outer module serve as output requirements for the last switching section. In the case study, the individual inner modules were documented independently of each other. Their functionality was described in terms of their preconditions and postconditions, even though this information to a large extent was incomplete as mentioned in Sect. 4.4. The pre and postconditions are symbolized in Fig. 4.15 by dashed lines crossing the arrows entering and leaving the inner modules, where the modules labeled B1 and Bn serve as examples. However, when it

![Diagram of the inner modules and their interrelationship](image)

Figure 4.15: The inner modules and their interrelationship
comes to the interrelationship between the inner modules, we found a gap in the documentation. All the switching sections, seen as a whole, take First Input as input and produce Last Output as output. Each of the individual inner modules participate in this overall operation. However, there was no apparent connection between the tasks performed by the individual inner modules and this overall role. This aspect will be further discussed in the next section.

4.5.4 The Case Study and its Semantic Integrity

Assume that the individual modules, such as B1 through Bn in Fig. 4.15, are themselves well specified. That is not enough to show how they are related to the First Input condition or how they contribute to meeting the Last Output condition. This is because there are several semantic aspects involved. The outer module has both a specification aspect and an implementation aspect. Similarly, each of the inner modules also has a specification aspect and an implementation aspect. The relationship between these four aspects is illustrated in Fig. 4.16. The specification of the outer module was well documented in the case study, but we could not find any documentation, such as for example A, B and C in Fig. 4.16, for its implementation. This is an example of a semantic gap mentioned previously. For the inner modules, the specification was documented using preconditions and postconditions for each individual module, such as X and Y, but these were not explicitly related to the outer module. The dependency between the inner modules and the surroundings was only implicitly present in the head of the designer and will be lost when a new person takes charge of developing the system further. In this context, it is not important whether the inner modules are implemented as inline code sections or as separate modules. In the case study, they were implemented as separate modules. We did not study the fourth aspect, the implementation of the inner modules.

![Figure 4.16: The specification and implementation aspects of the modules](image-url)
There are two kinds of dependency between the inner modules and their environment. On the one hand, the switching sections to which the inner modules belong are chained together, each one depending on the result of the previous one in the chain. This defines the pre and postconditions, such as A and C of Fig. 4.16, for each switching section and introduces a progress dependency on the inner modules. On the other hand, each branch of a switching section also depends on the outcome of the switch condition. This introduces a branch dependency on the inner modules, such as B and C in Fig. 4.16. Obviously, the pre and postconditions, such as X and Y, of the inner modules must satisfy requirements B and C from the surrounding branch. In many cases X will be the same as B and Y the same as C, but, especially when modules are reused, that does not need to be the case. Actually, the pre and postconditions for the A module, which is reused, are already defined. A sufficient condition then, is that the reused, inner module is "good enough", as described in Sect. 4.5.5 below. With reference to Fig. 4.16, pre and postconditions X and Y must at least satisfy the requirements defined by B and C. These, in turn, are derived from the position of the switching section in the outer module. One reason for the semantic gap that we identified between the specifications of the outer and inner modules may be that the two-layer architecture of the outer module does not appear clearly. The individual inner modules are one semantic level away from the outer module, and this may not be obvious to the designer. The rest of this section will focus on how to identify and document the semantic requirements for the inner modules and how to fill the semantic gap. The objective is to achieve semantic integrity for the outer module.

4.5.5 A Design Structure and Development Method for Switching Sections

Next we try to generalize the lessons learned from the study presented above. We look at a general situation similar to the one studied, e.g. an outer module whose implementation contains at least one switching section. We propose a small, pragmatic set of documentation rules to support the semantic integrity of the outer module, the switching section and the branches for the switch. We also propose a three-step method to help identify the necessary semantic information. The scope of the proposed method is limited to the overall view and one of the switching sections. The questions to which we propose an answer are the following:

- What documentation is necessary in the case of a switching section in order to assure the semantic integrity of both the switching section and of the surrounding module?

- When a switching section appears as part of a module’s implementation, how should one proceed to maintain a semantic chain from the module being
implemented down to the individual branches of the switching section in order to maintain the semantic integrity of the whole module?

The steps to a design and implementation, which we propose, shall promote the semantic integrity of the outer module and its constituents. In this case we propose a top down procedure, going through the following three steps:

1. Section contracts. Identify the major implementation sections of the outer module and express the assertions between the sections. Each switching section in particular will be identified as a separate implementation section. The assertions immediately surrounding the switching section will define the contract it has to satisfy, corresponding to the conditions at A and C in Fig. 4.16.

2. Branch contracts. For each branch of the switching section, identify the branch dependency. It is based on the precondition for the switching section and the switch condition for that branch. Then extract the contract for this branch, corresponding to the conditions at B and C in Fig. 4.16.

3. Branch specification and implementation. Specify and implement each branch so that it satisfies the corresponding contract. The pre and post-conditions for the branch must be "good enough" in relation to the branch contract. This corresponds to the conditions at X and Y in Fig. 4.16.

Step 1 relates to the implementation of the outer module and accounts for the progress dependency mentioned in Sect. 4.5.4. Step 2 relates to the implementation of the switching section and accounts for the branch dependency, also mentioned in Sect. 4.5.4. Step 3 relates to the specification and implementation of the branch sections as shown in Fig. 4.16. In this way, the implementations of the individual inner modules are tied to the requirements stemming from the implementation of the outer module. More often than not, the specification of the branch in Step 3 will be the same as the branch contract from Step 2, but they have different semantic bearings. They may also differ, for instance if the branch implementation is based on reusable components whose pre and post-conditions are already defined. The semantic gaps mentioned earlier are filled by steps 1 and 2. The point here is that we are deducing the requirements of what each one of the branches of a switching section should do (Step 2) from the switching section's position in the implementation of the outer module (Step 1).

This is a way to bridge the implementation of the outer module to the specification of the inner one. This sets the smaller building sections, which are the inner modules, in the context of the surrounding module and supplies a tool for verifying the consistency in the requirements. It therefore helps to achieve the semantic integrity of the outer module by assuring that it implements what it has promised in its own external contract. This method allows for a top down - as well as a bottom up - approach. The implementation may be done top down,
using the outer contract as its requirement specification. Steps 1 and 2 define
the requirements on each branch in the switching section. Step 3 implements
these branches. The implementation of the branches may also be bottom up by
the reuse of a module that already meets the branch contract. The following
subsections will develop these three steps further.

**Identify major implementation sections of outer module and corre-
sponding contracts**

In this section, we study only implementation sections that correspond to a
complete switching section. Once the sections are identified, the assertions be-
tween each section should be described. The descriptions do not need to be very
formal but should be as complete as possible. The assertions before and after
one section will constitute the contract for that section. This is illustrated in
Fig. 4.17. When the implementation of the outer module is seen as a sequence
of sections, each section drives the state of the module towards meeting the
outer module’s postcondition. Assertions Ai, which separate the implementa-
tion sections, should evolve, starting with the outer module’s precondition and
ending with its postcondition. Contracts Ci are defined by matching these as-
sertions pair wise in such a way that the postcondition of one contract will be
the precondition for the following one. Provided each section meets its contract,
this will assure the semantic integrity of the outer module, since the sequence
of implementation sections will implement the outer module correctly. The im-
portant point is that the correctness of the implementation can be based on the
section contracts alone and not on the actual implementation of the sections.

![Diagram of major sections and their contracts](image-url)

Figure 4.17: The major sections and their contracts
Identify the switch conditions and each branch contract

This is the step that connects the requirements specification of the individual branches to the overall implementation of the outer module. It is the central step in assuring the semantic integrity of each switching section. This is illustrated in Fig. 4.18. We have to make a simple assumption about switch S itself. It should give control to one of the branches only and not have any side effects observable to the rest of the switching section. If this is not the case, the switch will have to be split into smaller parts to meet these requirements.

<table>
<thead>
<tr>
<th>S</th>
<th>Pre</th>
<th>Branch i</th>
<th>Post</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pre</td>
<td></td>
<td>post</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i</td>
<td></td>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.18: The branch contracts

The branch contract is deduced from the contract for the whole switching section in the following manner. Assume that the condition for the switch to select branch \( i \) is characterized by assertion \( s_i \). The precondition of contract \( pre_i \) for branch \( i \) is then precondition \( Pre \) for the whole switching section strengthened with \( s_i \). Mathematically this is expressed as

\[
pre_i = Pre \land s_i
\]

The corresponding relation for the postconditions is simpler. Since there is no data manipulation in convergence point \( C \), postconditions \( post_i \) for the branches and \( Post \) for the switching section are the same, and thus

\[
post_i = Post
\]

It follows that each branch will have a separate contract, depending on the corresponding switch condition. If every branch is specified in this way, the semantic integrity of the branching section as a whole will be assured. This in turn then assures the semantic integrity of the outer module as a whole.
Specify and implement each branch so that it satisfies its contract

Now that the contract for each individual branch is known, all that remains is to implement that branch. This can be done in a top down or bottom up fashion. The top down approach is the most common one. It applies to the production of some code that transforms the outer module’s state from the state specified by the precondition to the one specified by the postcondition. The bottom up approach corresponds to reusing an already existing module. How that can be done without the risk of violating the semantic integrity of the branch is discussed below. To be a correct implementation of a branch, the reusable module must satisfy the contract for that particular branch. This is determined by comparing the branch contract with that module’s pre and postconditions. The module’s precondition should be satisfied by the branch precondition. Conversely, the branch postcondition should be satisfied by the module’s postcondition. Informally one may say that the module should be "better" than - or at least "as good" as - what is required by the contract, e.g. produce more with less input. This can be stated more formally by saying that the module’s precondition must be weaker than or equal to that of the contract and the module’s postcondition stronger than or equal to that of the contract. Weak and strong contracts are discussed further in 4.3.1. Expressed mathematically, using index \( m \) for the module and index \( c \) for the contract:

\[
\text{pre}_c \Rightarrow \text{pre}_m \land \text{post}_m \Rightarrow \text{post}_c
\]

It is interesting to note that this is the same condition as the one governing the semantic restrictions of a subclass routine, as discussed in [46]. In that case, \( c \) and \( m \) would correspond to the superclass and subclass, respectively. This analogy can be exploited in further studies. In the case study, one of the requirements was that it should be possible to add more branches in the future. To do that in a safe manner, Steps 2 and 3 should be repeated for each new branch. This, of course, requires that Step 1 has already been done.

4.5.6 Case Study Compared to the Proposed Method

In this section, we compare the design of the product studied with our proposed method for the case of switching sections. The questions we ask are these:

1. Were all or some of the three steps done, implicitly or explicitly, during the product development?

2. Were the results of these steps documented in the specification, in the design documentation or in the code itself?

3. What consequences can we predict from the answers to these questions for the present and future quality of the product implemented?
4.5. A DETAILED EXAMPLE FROM THE CASE STUDY

In the solution studied, there was certainly an understanding of the requirements for the individual inner modules, corresponding to step 1, but it was not documented. According to what we could see in the documentation, only Step 3 was done explicitly. The design leader followed a method, which we did not study in detail. The objective of our study was to see whether contract principles were applied or not, not the process itself. Thus we studied only the resulting documentation and not the development method applied to define this documentation. We may assume that it was a top down method, however, since the inner modules were actually tailored to the specification of the outer module, and thus the output from Steps 2 and 3 is identical. We also learned from the design leader that the method specifies that Step 3 be done, and we actually found that the branch modules were specified using pre and postconditions, although not completely.

However, both Step 1 and Step 2 were missing in the documentation of the design of the inner blocks. Therefore, we could not verify whether the specification and implementation of the branches, as defined by their pre and postconditions, satisfied the requirements of the outer module. On the basis of the conversations we had with the designers of the system, however, we can assume that even Step 1 was done, but informally and intuitively. The design leader, who also worked with the implementation, informally understood the conditions at each step and designed the branch blocks according to her understanding. It is thus reasonable to believe that the implemented solution was correct, although this knowledge was only in her mind. However, we saw no trace of Step 2, which is a conscious analysis of the consequences of the switching operation, in the documentation. In addition, the actual designer does not work with this product any more and, even if she did, she would have forgotten many of the details by now. Hence a problem may arise when the branch modules need to be changed or new branches be added.

In the current solution, part of the knowledge required for maintenance and further development will need to be extracted from the documentation of the pre and postconditions for the existing branch modules. This approach has two problems. First, as it turned out, even these conditions were not complete and well understood, but were included because the method used said so, so it may even be necessary to go to the implementation of the branch modules for information. This of course raises a serious question for any method maker. What is worse, a good method with a poor understanding or a good understanding with a poor method? This is a question for further study.

Second, the branch modules’ pre and postconditions express only the conditions actually met by the individual components according to Step 3. They do not express the needs set by the outer module as by Step 1 or the requirements stemming from the switching logic as by Step 2. Equipped with individual pre and postconditions only, the branch modules are like isolated islands. Some
reverse requirements engineering is necessary in order to extract the conditions they need to meet. In the design structure and method steps proposed earlier, the needs for each part of the outer module are clearly documented in the form of a contract. Any new inner module that follows the contract is a valid module. The outer module does not rely on the knowledge of the implementation of a specific inner module for its correctness, only on the contract.

4.5.7 Summary and Conclusions

In this section, the design and implementation of a software module containing a sequence of switching sections has been analyzed. We discovered "semantic gaps" in the detailed case study, meaning that there was no documented, continuous semantic line from the external requirements of the module down to the specification and implementation of the individual parts. The gaps appeared both in the specification of each switching section and in the documentation of the requirements for each switch branch. Therefore, the specification of the modules, which implemented the individual branches of the switching section, could not be backed up semantically. Rather, these specifications appeared in isolation and did not sufficiently support the continuous development and maintenance process. Nevertheless, at the time of our study, the project was successful.

We have shown how contracts can be set up to specify the requirements for the switching section as a whole. Contracts can also be used to specify the pre and postconditions for the branches of the switching section so that their correctness requirements can be established. When these requirements are documented it is also safe to modify the branches or extend the number of branches at a later date. We have also proposed three steps to identify and document this semantic information and to fill in the semantic gaps. The steps are simple and pragmatic enough to be applied in practical software development work.

It is interesting to note from the detailed case study how the branches of a selection follow similar semantic rules as subclasses in object oriented software construction. The contract defined for each branch corresponds to the contract of an abstract superclass or of an interface definition. The implementation of a branch corresponds to a concrete subclass. The contract for the branch settles the requirements for the branch, but the implementation does not need to follow the contract exactly. It can do better than what is required by the contract without violating it. Actually, this is a general observation regarding contracts [39, 46]. The implementation of the service that shall satisfy the contract may do better without harm but may not do worse. This observation suggests that each branch could be defined as an abstract class, with possibly different implementations defined as subclasses.
4.6 Discussion

This chapter has presented some background information on how semantic aspects are handled in both academia (Sect. 4.3) and industry (Sections 4.4 and 4.5). Although neither the academic survey, nor the industrial case studies provide conclusive evidence of how semantics are handled in general, they both nevertheless provide indications on the current state of affairs. The academic survey concluded that work is being done in the area of semantics handling and documentation and that most articles focus on executable contracts in the development phase. The industrial case studies concluded that industry produces working code in spite of not using the academic ideas to any larger extent. Academia has been busy producing methods and techniques, whereas industry has been busy producing software, to summarize. The question arises what benefits could be made by incorporating more of the academic ideas into industrial practice. The answer is probably not straightforward. Some of the academic ideas would probably cost more than they could bring whereas some would bring great benefits at low costs. What industry needs is more concrete evidence on how the different academic approaches perform in industrial settings in order to make the correct judgements and selection of appropriate methods. This evidence can either come from previous industrial experience or from empirical evaluations. Industrial experience is naturally left to industry and what remains for academia to do is empirical evaluations. Chapter 6 presents three empirical evaluations aimed at comparing different academic approaches for handling semantics.

In order to make industry use a more mature way of handling semantics, a method is needed that can be introduced in industry without diverting the focus from producing software. The method would have to be perceived as "natural" or "intuitive" and also provide measurable benefits in terms of time and hence money in order to gain success. Much of the experience gained from the case study presented in Sections 4.4 and 4.5 has served as input when designing the Semla method, which will be presented in the next chapter.
Chapter 5

SEMLA - A Method for Handling Structured Semantics

As has been shown in Chapt. 4, there exists a number of academic methods for handling semantic aspects but industry does not seem to use these methods. We believe that a partial reason for this is that industry perceives the methods as too complex and too far from their usual way of working. There are of course a number of industrial companies that have successfully adopted contracts like Eiffel Software [20] and Enea [21], but the majority have not. We believe that there are two main obstacles that might limit industrial usage. Firstly, most of the methods available for introducing contract-based programming can be perceived as rather complex or require a specific development environment [14, 24, 28, 33, 36, 47]. Secondly, there is little empirical evidence that such programming is actually beneficial in terms of time and quality. In this thesis we try to address both of these issues and our method can be introduced with little effort in any development environment.

The method, denoted Semla\(^1\), was developed as a means of increasing the semantic awareness of industrial software developers in their daily work. The general underlying hypothesis is that if more work is spent on properly defining the semantics of a piece of software, less semantic errors will occur, thus lowering the total work load needed for producing a correct piece of software. Since syntax is generally well handled by interpreters and compilers, most if not all syntactic errors are removed before the software is released. Semantic errors,

\(^1\)Semla is a Swedish acronym which read out roughly translates to A method and quick reference for handling semantics.
however, can not as easily be detected by automatic means due to a number of reasons.

The first and perhaps most prominent one is that software often communicates with the surrounding environment whose behavior might not be known at release time. An end user cannot be expected to be as thorough as to check every input for semantic consistency. Hence all user input poses a threat to the software.

Another reason that it is difficult to handle semantics automatically is that the possible permutations of the paths through a large software system make it impossible or at least very difficult to cover all combinations to ensure that not a single path results in a semantic incorrectness.

A third and final reason is that every program developer has a perception of what his or her software parts do and what the parts should handle in terms of input from other parts. This information must be conveyed to other developers in order for the internal communication to function properly. There is at present no way for an automatic tool to investigate the minds of the developers and the only way to communicate this is by way of documenting what semantic limitations a certain piece of software has. The method tries to remedy all these problems by imposing an increased awareness of semantic aspects on the developers and providing guidelines on how to handle and document semantically important parts of the software.

5.1 Overview

The rest of the chapter contains a short version of the Semla method description which can be found in full in Appendix A. This chapter only contains the parts of Semla on how to practically handle semantics in the design phase. The choice of including only this part is twofold; first, it is the first part that would be used by industry when first introduced to the method and second, it is the part most used in the evaluations presented in Chapt. 6. The full method description also includes parts on terminology, how to properly document semantic aspects, how maintenance should be done to preserve the semantic integrity, as well as a full example of a linked list constructed and documented according to Semla.

The part presented below is a slightly edited version of the design part of Semla. Semla is based on guidelines and in the particular excerpt included here the guidelines are divided into three categories: interface guidelines, external guidelines and internal guidelines. For each guideline, a short description and motivation is first given, which is followed by a rule of thumb, a working procedure and finally an example of some program code designed according to the guideline. The examples are taken from the List module found in Appendix A.
5.2 Interface Guidelines

An interface describes how a module presents its services to other modules. It is normally built using exported methods. They operate on the data in the module and may use other supporting internal methods. It is important to note that an interface is normally an abstract construction and that the exported methods implement the interface. More than one module may share the same interface, which is another reason to keep the interface definition separate from the modules. As an example, an interface may define some methods and different classes implement this interface in different ways for different hardware, operating systems or versions. This section describes how an interface can be designed.

5.2.1 Describe all Methods

*Description and motivation:* Every method should be described in such a way that the user knows the intended and legal uses of the method. As long as the description is valid, the implementation of the method may be changed without the user being affected by the change. If a method name that reflects the purpose of the method is used, the description can normally be kept short. However, one should always provide a short explanation of what the method does.

*Rule:* Provide an overall description of the intentions and usage of each method.

*Procedure:* If the method is well defined and simple, there is normally no problem in describing it. Write one sentence to explain its functionality and possibly a few more sentences to describe the extra information needed to use it correctly.

*Example 1: The description of a method*

```java
/******************************************
* add: adds anElement to the beginning of the list
******************************************/
public abstract List add(Object anElement);

/******************************************
* remove: Removes the element at position 'position' from
* the list (counted from 0)
******************************************/
public abstract List remove(int position);
```
5.2.2 Define Contracts

Description and motivation: A contract is exactly what it sounds like: the definition of what a user and a supplier can expect from each other. Contracts play an important role in assuring that methods are used correctly. A contract implies that the user of a module is guaranteed a correct result if he supplies the module (or normally a method) with correct input. The pre- and postconditions, which constitute the contract, define what is meant by correct. The following subsections contain more details about preconditions and postconditions.

In addition to the pre- and postconditions, there must be an agreement between the module programmer and the module user that the contract should be respected from both sides. In situations where contracts are not used, a default contract is often assumed, saying “I don’t trust anybody and nobody should trust me”. This normally leads to redundant testing and extra code. Use and trust contracts in order to make the code shorter, simpler and easier to understand.

Rule: Define and express contracts for all methods.

Procedure: Determine the acceptable values for the input parameters to your method. In addition, look for other conditions that the client code must guarantee, for instance that a communications channel is open. Express the precondition in plain English or through program code. Then, assume that all values are correct upon designing the method code. Express in the postcondition what the method has achieved. In some cases the postcondition simply restates the method description, but in many cases it will contain further details.

Example 2: Contract

```java
/**********************************************************
* add: adds anElement to the beginning of the list
* @param anElement the data element to be added
* @param Pre anElement != null
* @param Post anElement is the first element in the list,
* the list size is increased by one,
* the modified list is returned
**********************************************************/
public abstract List add(Object anElement);
```

We use the comment conventions for Javadoc. A Javadoc comment states that the comment should start with /** and end with */. The Javadoc utility will extract the text in the comment to a documentation file. A single * in the beginning of the list is not included in the documentation. The keywords @param is recognized as a parameter description. The first word following the keyword param will stand out in the documentation. Since Javadoc does not specify pre- and postconditions, we use the parameter mechanism to document them.
5.2. INTERFACE GUIDELINES

5.2.3 Specify Preconditions for Methods

**Description and motivation:** The precondition to a method is an expression for a condition that must be true before the method can be invoked. It may be directly measurable, for example that the value of one of the parameters to the method must have a value restricted to a certain interval. It may also be subtler. For instance, it may require that some other method be called prior to the method in question or that certain data elements must have certain values.

**Rule:** Express the precondition for all methods.

**Procedure:** To find the precondition, identify the parameter values that can cause problems for your method or that violate the invariants. The precondition can be thought of as a virtual filter. It lets the correct values pass through and filters the illegal values. The module designer should ask himself the following questions to find the relevant precondition. When and how may the method be called? Are there any values that will cause problems in the method? A method should always have a precondition. In some cases, it will simply be the condition true, meaning that the method may always be called. Examples of methods having this precondition include most so called *predicates*. Predicates are methods that do not affect the object in any way. They only return information about the state of an object.

**Example 3: Precondition**
The method `getElement` from the class `List` shows a simple example of a precondition. The method returns the data element at the given position in the list. It is not possible to return data from a position outside the current list, so the user must assure that the position is within the list limits before calling the method. The precondition for `getElement` can be expressed as

```java
// Pre: position is from 0 through the number of
// elements in the list minus one
```
The precondition can also be formalized using program code, in order to avoid misunderstandings. In some cases, the text volume may also be reduced.

```java
// Pre: 0 <= position && position < size()
```

### 5.2.4 Specify Postconditions for Methods

**Description and motivation:** A postcondition describes the result of invoking a method or the effect produced by the method. In the simplest cases, a variable in the object executing the method may have changed or a value may have been computed and returned. In more complicated situations, several variables or even the state of the whole system may have changed.

**Rule:** Express the postcondition for all methods.

**Procedure:** Determine the result produced by the method. This is normally the postcondition of the method. The postcondition should not be related to specific code but be kept at a more abstract level. The most important aspect is that the postcondition expresses what the method has achieved and not how it is done. Here are some questions to ask to help identify the postcondition:

- What has the method done? In what way is the state of the object different after a call as compared to before? Has any variable changed its value? Which value did the method return? Where convenient, the pseudo-variable `result` may be used to express the result returned from a method.

**Example 4: Postcondition after a change in an object**

The postcondition for `add` in Example 2 was:

```java
// Post: anElement is the first element in the list,
// the list size is increased by one
// the modified list is returned
```

The postcondition can also be more formalized in order to avoid misunderstandings.

```java
// Post: anElement == getElement(0)
// size() == old size() + 1
// result == the modified list
```

The term `old` should be interpreted as the value, at the start of the execution of the method.

**Example 5: Postcondition for accessing the state of an object**

It may be useful to define separate predicate methods, for instance `isPositionOk`, for the kind of preconditions that are needed by `remove` in Example 2 above. We will call this a `guard method`. Such a method improves the understanding of the program. It also makes it easier to avoid hard to spot errors produced by negligence, for instance that a programmer writes `<=` instead of `<` in a test.

```java
// Post: a boolean value telling if position is legal
// for accessing a list is returned.
```
5.2. INTERFACE GUIDELINES

// true= position ok, false=position not ok
public abstract boolean isPositionOk(int position);

This postcondition may seem trivial at first, but gives enough information to make it unnecessary to see the code of the method. Thanks to the postcondition, it is not necessary for the user to know the exact details needed to access the list.

5.2.5 Identify Exported Invariants

Description and motivation: An exported invariant expresses relations and conditions that a module guarantees and that the user can always trust. Since the exported invariants express requirements that the methods of the module must fulfill, they constitute an important part of the definition of the module interface. The exported invariants should not depend on any particular implementation but rather describe the concept that the module is implementing.

Rule: Identify and document any exported invariants for the module.

Procedure: Make notes of any conditions or properties for the modules that are known to the user of the module and that are always true. They are the exported invariants. Often, they seem trivial and are therefore overlooked. Sometimes they may be expressed through one of the methods of the module, sometimes through several methods and sometimes in plain English. The invariants should be expressed using only exported properties and should not depend on any particular implementation.

Example 6: Exported invariants, informally

// Exported invariants:
// the size of a list can never be negative
// a list is empty if its size is zero
// positions are counted from zero through list size minus one
// no position in a list contains null data

Example 7: Exported invariants, formalized

// Exported invariants:
// size() >= 0
// isEmpty() == (size() == 0)
// positions are counted from zero through size() - 1
// for all legal values of position:
// getElement(position)! = null
5.3 External Guidelines

This section contains guidelines on how to use modules as black boxes. The guidelines should be used when several modules are combined with each other or an existing module is used by one that is under development.

5.3.1 Satisfy the Preconditions

Description and motivation: A precondition expresses what is required to use a method in the way intended. It is therefore necessary to make sure that the precondition is satisfied before the method is called.

Rule: Satisfy the precondition for all methods called.

Procedure: Every time a method is called, the user must make sure that the preconditions for that method are satisfied. Either the program flow should be such that it guarantees the precondition or there should be a test for the precondition immediately before the method invocation.

Example 8: Implicitly satisfied precondition
The precondition for the method `getElement` is that the input parameter `position` must have a legal value. If the programmer knows this to be true, for instance because there is only one possible flow of control to the call, the precondition does not need to be tested explicitly.

```java
... 
myList = myList.add(data);  // the list has at least one element at position 0 now 
// and it is therefore safe to call getElement for position 0 
anElement = myList.getElement(0);  
... 
```

The above code is correct.

Example 9: Precondition satisfied through testing
If the program has more than one possible flow of control to `getElement` and not all of them guarantee the precondition, then the user must perform a test to assure that the position is legal.

```java
... 
if (x == y) 
    myList = myList.add(anElement); 
else 
    doSomethingElse(); 
if (myList.isPositionOk(pos)) // test 
anElement = myList.getElement(pos); // method invocation 
else 
    ... 
```
System.out.println
("" + pos + " out of bounds (0-" + myList.size() + ")"); } ...

The program above either adds an element or does something else, depending on the values of \(x\) and \(y\). The validity of the call to \texttt{getElement} must therefore be tested explicitly.

In many cases, the program flow and input values are given by an end user, and end users may never be trusted. In all cases where an end user is involved, the preconditions must be tested.

### 5.3.2 Trust the Postconditions

\textit{Description and motivation:} To trust a postcondition simply implies that the user can assume that the method has done what it has promised through the postcondition. There is no need to test what is already guaranteed to be true according to the postcondition. This of course assumes that the user has satisfied the precondition when the call was made. If the precondition was not satisfied, nothing can be said about the situation when the method returns (if it returns at all).

\textit{Rule:} If the precondition was satisfied, then trust the postcondition.

\textit{Procedure:} Make sure that the conditions contained in the precondition are satisfied, either implicitly through the program flow or explicitly through a test. Then, trust that the method does what it has promised through the postcondition and do not test for conditions that are already stated to be true.

\textit{Example 10: Trust the postcondition}
This example shows how the precondition is tested before the call to \texttt{remove}, but the postcondition is not tested after the return from the method.

```java
List myList = new List(); ...
myList = myList.add(aValue); ...
if (myList.isPositionOk(pos)) /*checking pre for remove */
{
    myList = myList.remove(pos);
    // no need to check outcome,
    // success guaranteed by the postcondition
    System.out.println("'Element at pos ' +pos+' was removed'");
}
```
5.3.3 Use Methods Correctly

_Description and motivation_: The user of a module should only use the methods of the module in the intended way and not utilize any side effects, which may exist. If the module is implemented in a different way in the future, side effects that are exploited can disappear. The client software would then be based on effects that no longer exist. Correct use of methods is normally not a large problem, but it is still worth mentioning.

_Rule_: Only use a method according to the documentation.

_Procedure_: Normally, using methods correctly does not present problems. It is still worth mentioning that one should be aware of the issue.

_Example 11: Use methods correctly_
Assume that one wants a certain delay in a program. Assume further that experience has shown the addition of a certain number of elements to the list and the subsequent removal of the same elements to consume the necessary amount of execution time.

```java
... 
{ 
    List delayList = new LinkedList().add(new Object());
    for (i = 0; i < delay; i++) // consume some time:
        delayList = delayList.add(delayList.size()-1);
    delayList = delayList.erase();
}
... 
```

The List implementation may be replaced by a more efficient one in a future release, or the program may be run on a faster machine. It would therefore be a mistake to use the list in order to create the delay in the client program. Often it is not even possible to use a method for other effects than the intended ones, but it may still be worth keeping this point in mind.

5.4 Internal Guidelines

The internal guidelines contain criteria for the internal design and the implementation of a module. Since the module is restricted by the interface, it is important to respect all the conditions defined by the interface definition. It is also important to describe data in a clear way.
5.4. INTERNAL GUIDELINES

5.4.1 Respect the Contracts

Description and motivation: To respect the contracts implies to trust them and not to test for conditions that are already true according to the contract. If both the programmer and the user of a module respect the contracts, then the software will be clean without redundant tests and similar anomalies. The output from the methods must follow the conditions described in the postconditions. The programmer must also trust that the preconditions protect the module from being used in an incorrect way and not perform redundant tests on the input parameters. The tests do not only take time to write, they consume space in the finished code, take time to execute and are useless. They also normally complicate the code and are, as all other code, extra sources of errors. Thus, no tests should be performed for conditions already fulfilled through the preconditions.

Rule: When implementing a module, respect the contracts defined.

Procedure: Do not test conditions that are already true according to the precondition. A quite common mistake is to not trust the precondition ("But what if they call it in the wrong way...") and to test it in the method. This is unnecessary and only takes a lot of time. Also, make sure that the postcondition and invariants are true when the method completes.

Example 12: Respect the contracts in the implementation

```java
/************************************************************
* @param Pre isPositionOk(position)
* @param Post the element, which was at position 'position',
* is not in the list any more
* the list size is decreased by one
************************************************************/
public List remove(int position)
{
    LinkedNode toRemove; // the element to be removed
    if(position == 0) // first element should be removed
    {
        toRemove = first;
        first = toRemove.getNext(); // remove from the list
    }
    else
    {
        // no test on first or position needed,
        // thanks to the precondition

    }
}
```

3Some people prefer the form `isPositionOk(position) == true`, which means the same thing. The expression `isPositionOk(position)` is a Boolean expression and if the expression is true, the assertion is true as well.
LinkedNode previous = first; // temporary
for(int i = 0; i < position - 1; i++)
{
    previous = previous.getNext();
}
toRemove = previous.getNext();
// pre guarantees this to be OK
previous.setNext(toRemove.getNext());
// remove from the list
}
return adjustSize(-1); // maintain internal invariant
} // remove

No test is needed inside the method to verify that position has a legal value. That is already settled in the precondition, which should be satisfied by the caller. The only testing needed in the method is for the special case of removing the first element in the list.

5.4.2 Describe the Data

Description and motivation: All important aspects of the variables used to implement the module should be described in order to avoid any doubt about how they should be used.

Rule: Describe all the variables used to implement the module in order to avoid any doubt about their intended use.

Procedure: Determine and document what each variable represents and for what purpose and how it should be used. Exceptions may be made if this is sufficiently clear from the name of the variable or if the variable is a temporary variable, for instance a loop counter.

Example 13: Data descriptions

    public LinkedList inherits List
    {
        private LinkedNode first; // the first node in the list
        private int size; // the total number of nodes
    ...
    }
    ...
    void someFunction(...)
    {
        int i; // loop variable
        int top; // index of the topmost element in array
    }
for (i = 0; i < MAX; i++)
    doSomething(i, top);
}

In the above example, the usage of each variable is described. One variable has been tagged "loop variable" to explain why it exists.

The last example shows a common source of error. Does top refer to the last element used or to the next position to be used? Does top represent the index of the topmost element of the array or is it the number of array elements used? Attempts are sometimes made to express different semantic meanings with one single variable. After a while, the programmer is no longer sure about how the variable should be used. It is important to document the intended use of a variable well, so that errors can be avoided.

5.4.3 Identify Internal Invariants

Description and motivation: An invariant is something that does not change. It is an expression that is constantly true. In programming, invariants are used to describe conditions that do not change. Such descriptions make additions or changes easier to perform by clarifying how and when a variable may be used. Every method can assume that the invariant is valid upon entry to the method and must ensure that it is valid at the exit from the method. In some cases, a method will break the invariant temporarily during its execution, to perform some test or to update some object, but it must make sure that it is reestablished before exiting the method.

For local variables defined and used in smaller, easily understood methods, it may not always be necessary to describe any invariants. For loop variables in somewhat more complicated methods, it is often useful to explicitly state an invariant to avoid "off-by-one" errors. After a while, it may be difficult to remember whether a certain variable represents the first free or the last used position or if it points before, on or after something. It is easy to use the variable differently at different places in the program.

Rule: Express all limitations for a variable in the form of invariants.

Procedure: Consider the restrictions that apply for a variable or for a group of variables. Write these down at the same time as the variable or variables are declared.

Example 14: Identify internal invariants
Assume that you want to add, to an implementation of a linked list, a variable maintaining a count of the number of elements in the list. The variable is increased by one every time an element is added to the list and decreased by one when an element is removed. This has the advantage that the whole list
does not need to be counted to get its size. This mechanism is not visible from
the outside, so it is a purely internal optimization.

```java
// LinkedList.java
public class LinkedList
{
    private LinkedNode first;
    private int size; // The number of elements currently in
    // the list, 0 <= size < Integer.MAX_VALUE
}
```

The invariants for the variable `size` are:
- `size` is equal to the number of elements in the list
- `size` is non-negative and smaller than `Integer.MAX_VALUE`

These invariants should be made explicit close to the corresponding variable
declaration. The statements above may seem evident, given the name of the
variable. However, in many similar cases, the use of a variable is not evident.
Is it for instance the number of elements contained in an array, the position of
the last element added to the array or the position of the next element to add?

### 5.4.4 Establish and Maintain Invariants

**Description and motivation:** In order for an invariant to hold, it first needs to
be established. That should be done as soon as the object is created, by the
constructor. Then, every method that modifies a variable must assure that the
invariants still hold when the method exits.

**Rule:** Make sure that the constructors establish the invariants and that all other
methods maintain them.

**Procedure:** The constructors are useful for establishing the invariants. In fact,
that is a constructor’s most important task, and it should normally be restricted
to doing just that. Identify all methods that modify the data and verify that
the invariants hold before they return.

**Example 15: Establish and maintain invariants**
The constructor must establish the invariants. Initializing the list to be empty
satisfies the condition that the list size should be non-negative. The variable
`size` should be equal to the number of elements in the list. All methods modi-
ifying the data must maintain these invariants.

```java
// LinkedList.java
public class LinkedList extends List
{
    public LinkedList() // constructor, initialize an empty list
    { // establish invariants:
        first = null; // exported invariant (empty list)
    }
```
size = 0; // internal invariant (size=list size)
}
public List add(Object anElement)
{
    ... code to add an element ...
    return adjustSize(1); // maintain internal invariant
}
// the precondition assures that the list is not empty
public List remove(int position)
{
    ... code to remove an element ...
    return adjustSize(-1); // maintain internal invariant
}
} // class LinkedList
Chapter 6

Empirical Evaluations

This chapter presents the empirical investigations undertaken to evaluate the merits of different methods for handling semantics in software development, including the Semla method presented in the previous chapter. Each section presents an evaluation in full detail and includes a subsection on the lessons learned from the evaluations.

6.1 A Controlled Experiment in an Academic Environment

This section describes an experiment designed to compare the Semla method, as presented in Chapt. 5, with a mainstream programming method based on exceptions. The purpose of the experiment was to evaluate whether using contracts would shorten development time, improve work satisfaction and increase the quality of the final programs. We chose to conduct the experiment in a course at the university, thus giving us control over the environment but at the same time sacrificing some of the possibilities for generalizing the results to an industrial setting as elaborated in Sect. 2.2.3.

The remainder of this section is structured as follows: First, the design of the experiment is presented followed by a description of the execution of both the experiment and the course. After that analysis and interpretation of the results are discussed followed by validity evaluation, summary and conclusions, and a summary of the lessons learned.

6.1.1 Experiment Design

This subsection describes the design of the experiment. The choice of baseline method, context selection and hypothesis formulation is presented along with
the selection of variables and the selection of subjects.

Choice of baseline method

The contract-based method Semla evaluated in the experiment focuses on contracts as an error handling mechanism. Apart from that, it includes mainly common sense knowledge. In selecting a baseline method we wanted a method with an alternative error-handling mechanism. All other aspects were to be kept identical to those in the contract-based method in order to identify the impact of the contracts. Since the language used in the experiment was Java, we chose a method based on exceptions. There are other ways of handling errors such as status flags, returned status codes etc, but the most accepted way in the Java-community is exceptions. An exception-based method also satisfied our criteria that the method had to be common, realistic and useful. It would have been unethical to teach a method known in advance to be inferior.

Context selection

Below is a presentation of a number of attributes of the experimental context according to [84].

- The context of the experiment was a university course given in 1999 at the Computer Science Department at Karlstad University in Sweden. The course was titled Project Work in Java and was set up for the purpose of conducting this experiment. A course in Java and project work was also requested by the students and therefore fit well into the curriculum.

- The experiment was an off-line experiment and, as such, has both drawbacks and advantages. The level of control is rather high in an off-line experiment but, since the experiment is not conducted in a production setting, the results may not be transferable directly to industrial environments.

- The participants were undergraduate students, normally in their third year of the computer science program.

- The experiment was specific since it focused on the usage of programming methods.

- The experiment addressed a real problem, i.e. the effect of programming methods on time, work satisfaction and program quality.

- The experiment was carried out as part of a course, and is therefore relatively easy to replicate.
Hypothesis formulation

As stated previously, this experiment had three experimental goals. The first was to measure the time consumption, the second to measure the perceived work satisfaction during a software development project and the third to evaluate the quality of the resulting programs. Our main hypothesis was that a contract-based method would decrease development time, improve work satisfaction and increase program quality. Below are the more formal definitions of our hypotheses.

- Null hypothesis 1 - The time needed for development will be the same in groups using the contract-based method as in groups using the exception-based method.
  \[ H_0^1 : time_c = time_x \]

  Alternative hypothesis 1 - The time needed for development will not be the same in groups using the contract-based method as in groups using the exception-based method.
  \[ H_1^1 : time_c \neq time_x \]

- Null hypothesis 2 - Work satisfaction will be the same in groups using the contract-based method as in groups using the exception-based method.
  \[ H_0^2 : satisfaction_c = satisfaction_x \]

  Alternative hypothesis 2 - Work satisfaction will not be the same in groups using the contract-based method as in groups using the exception-based method.
  \[ H_1^2 : satisfaction_c \neq satisfaction_x \]

- Null hypothesis 3 - The quality of the final programs will be the same in groups using the contract-based method as in groups using the exception-based method.
  \[ H_0^3 : program\_quality_c = program\_quality_x \]

  Alternative hypothesis 3 - The quality of the final programs will not be the same in groups using the contract-based method as in groups using the exception-based method.
  \[ H_1^3 : program\_quality_c \neq program\_quality_x \]
CHAPTER 6. EMPIRICAL EVALUATIONS

Selection of variables and measurements

The treatments were the program development methods. The main variables were time required to develop the program, subjective work satisfaction and program quality. Data on the first two variables were reported on daily forms. The students reported when and with which development activity they had been working as well as how they rated their work satisfaction. The students could choose from eight different activities. These activities were system design, subsystem design, implementation, subsystem testing, subsystem debugging, integration, system testing and system debugging. Time was recorded with a precision of half an hour. Work satisfaction was measured on an ordinal scale with the values 'bad', 'OK' and 'good' for each reported half hour. Program quality was evaluated using the built-in metric tools of the Together development environment [79]. Program quality is difficult to measure and different ways of doing this have been proposed [22]. An automatic evaluation was chosen for practical reasons but also has the advantage of being objective.

Grouping students

The unit of observation was the project group, consisting of four students. Although the students reported their work on individual forms, the group members were not independent from each other. It was therefore not possible to view them as individual subjects.

The choice of four persons per group was a trade-off. On the one hand we believed that a larger group would be needed to form a project work group and complete the assignment in time. On the other hand we wanted as many groups as possible to obtain the maximum amount of independent data for statistical purposes. Our project work model corresponds to the "Small Group Project" model described by Shaw [70].

While our intention was to compare two software development methods and we thus wanted to split the class into two equal halves, a total of 36 students was admitted to the course, which resulted in one part with five groups and one with four.

Randomization and blocking

The students were mainly third-year students but some second-year students also took the course. The difference in experience required that certain measures be taken. We wanted the groups to be fairly equivalent in terms of programming experience and thus had to make sure that each group had the same proportion of second- and third-year students. This was achieved by dividing the students into groups using stratified random sampling. We first randomly selected one person per group from the second year, then one from the third year and so on. This resulted in a group composition with one or two persons from the
second year and the rest from the third year. We could have done a pre-test to establish the level of competence for each student, and then used this knowledge to construct the groups, but chose not to for both ethical and practical reasons. If we had chosen to rank the individual students, we would have had to do this discreetly in order not to offend any of them. Instead, we chose to be explicit about what criteria we had used to block the students and even to make the random selection publicly in front of all the students.

**Statistical implications**

Most statistical tests require a large random sample or samples from a known distribution [9]. If proper blocking and randomization of the test subjects are done, however, statistical tests such as t-tests are possible in spite of the low number of test data points. These tests compensate for small sample sizes and the statistics obtained from them are therefore useful without further compensation.

### 6.1.2 Operation

This subsection presents the way the students were prepared for the experiment, the instrumentation needed and the execution of the experiment and the course. Finally, we briefly discuss data validation.

**Student preparation**

The students were told that we wanted to study the effect of different programming methods on the development work. They were also told what we wanted to measure and how the data were to be collected. Each half of the class was given one lecture on their method but received no information about the method used by the other half. The students were further instructed only to communicate with other students that used the same programming method. This was an attempt to hide the actual method being evaluated from the other half. The attitudes and the work of the students might have been affected if they had known which methods we wanted to compare. This in turn would have compromised the validity of the experiment.

**Instrumentation**

In general, there are three types of instruments for an experiment: objects, guidelines and measurement tools [84].

The instrumentation object used in our experiment was the assignment specification, which was given to the students at the start of the experiment. The assignment was to create a strategy game composed of at least three separate parts, game engine, Graphical User Interface (GUI) and computer opponent.
The game had to be constructed by two pairs of students, one pair responsible for the game engine, and one pair responsible for the GUI and the computer opponent. The reason for this was that we expected different forms of error management to have different effects on the communication over the interfaces between different parts.

The instrumentation guidelines were the method documents, one for the contract-based method and one for the exception-based method.

The measurement tools used were forms on which the students reported activity, time and work satisfaction. We tried to minimize the effort needed to fill in the forms by pre-printing all common information on the forms. All the students had to do was to indicate with an 'X' what activity they had worked with, and with a letter how they had perceived their work satisfaction for each half hour.

**Experiment execution**

The experiment was carried out over a period of ten weeks, during which the students worked half time on solving the assignment. Each week the students submitted their time reports for the activities of the past week. We also had weekly meetings with the groups, in which we helped the students in applying the methods. At the end of the course, the students submitted their source code and a written presentation of their work.

**Course execution**

This subsection discusses some additional matters concerning the course that are not directly related to the experiment. Since the course was based on project work, few lectures were given and most of the teaching was done during weekly meetings. A few initial lectures were held to help the students get started. An introductory lecture explained the goal of the course, the aim of the research and the need for student cooperation. Four lectures gave a compressed introduction to Java and one lecture focused on project work and organization. As mentioned previously, each class half also attended one lecture in which the method they would use was taught.

The remaining part of the course offered only the weekly meetings. Each group had 20 minutes per week to discuss problems with the project assignment, matters concerning the methods and issues regarding project work. Teaching assistants were also available for additional consulting.

There were only two grades for the course: pass and fail. To pass the course the students had to complete the project, make a final presentation, write a report on the project work and submit data forms and program source code. At the end of the course, the students presented their solutions to the rest of the class.
Data validation

Data was collected for the nine groups, which included all 36 students. After examining the outcome and holding discussions with the students, we later had to exclude one group from further analysis. This particular group did not finish on time. This led to a strong bias in the distribution of their time for the different activities since they never reached the later development activities, such as integration and testing. This group used the exception-based method.

This left us with eight groups for statistical analysis and treatment, three using the exception-based method and five using the contract-based method.

6.1.3 Results

In this subsection we present the descriptive and derived data gained in the experiment. First an overview of the results in the form of descriptive statistics is given, followed by the testing of the hypotheses. Each group will be referred to by a label. Groups labeled X1–X3 were in the exception-half and groups C1–C5 were in the contract-half.

Programming method and time consumption

We gathered data on time spent on the different activities involved in the program development process. Figure 6.1 shows the average time spent by the groups in the two halves of the class for each activity. As can be seen in the figure, the contract-based groups spent more time in the first two activities but
CHAPTER 6. EMPIRICAL EVALUATIONS

less time in all others. Figure 6.2 shows the same information, but for each group rather than the average for entire halves. Group X3 of the exception half stands out for activities 4-8, i.e. subsystem test through system debugging. This group spent close to twice as much time on solving the assignment as any other group, regardless of class half. One reason for this was an implementation error that they spent almost two full days on solving.

![Figure 6.2: Time consumption: hours per activity per group](image)

Programming method and work satisfaction

Figure 6.3 shows the frequency distribution per group for the reported work satisfaction. Groups in the contract half generally perceived their work satisfaction as slightly higher than groups in the exception half.

Programming method and program quality

One important aspect when evaluating different program development methods is the quality of the resulting programs. There are a number of different strategies that can be used for measuring quality, such as inspections, testing and automatic generation of metrics using a designated tool. For practical reasons, we chose to make the quality evaluation using an automatic tool, thus giving us what can be considered objective data. The particular tool used in this study was the built-in metrics suite of Together [79], which produces 47 different metrics for software programs. Some of the metrics are obtained by simply counting certain entities in the program, while others are derived or calculated. Since the range for different metrics differ to a rather great extent,
6.1. A CONTROLLED EXPERIMENT IN AN ACADEMIC ...

Figure 6.3: Work satisfaction: frequency distribution per group

Figure 6.4: Average program quality: logarithmic metric value per class half
Table 6.1: Total time per group and activity

<table>
<thead>
<tr>
<th>Activity</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>t0</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Design</td>
<td>47.5</td>
<td>9.5</td>
<td>32.0</td>
<td>32.0</td>
<td>31.0</td>
<td>38.5</td>
<td>45.5</td>
<td>48.5</td>
<td>1.01</td>
</tr>
<tr>
<td>Subsys. Design</td>
<td>29.0</td>
<td>25.5</td>
<td>39.5</td>
<td>24.5</td>
<td>66.5</td>
<td>39.5</td>
<td>52.5</td>
<td>48.0</td>
<td>1.22</td>
</tr>
<tr>
<td>Implementation</td>
<td>173.0</td>
<td>166.5</td>
<td>155.0</td>
<td>89.0</td>
<td>116.5</td>
<td>103.5</td>
<td>113.5</td>
<td>126.0</td>
<td>5.97</td>
</tr>
<tr>
<td>Subsys. Test</td>
<td>39.5</td>
<td>31.5</td>
<td>107.5</td>
<td>61.0</td>
<td>51.5</td>
<td>38.0</td>
<td>34.0</td>
<td>25.0</td>
<td>0.90</td>
</tr>
<tr>
<td>Subsys. Debug</td>
<td>22.5</td>
<td>23.5</td>
<td>122.5</td>
<td>51.5</td>
<td>23.5</td>
<td>23.5</td>
<td>30.0</td>
<td>29.0</td>
<td>0.95</td>
</tr>
<tr>
<td>Integration</td>
<td>6.5</td>
<td>76.5</td>
<td>5.0</td>
<td>26.5</td>
<td>0.5</td>
<td>18.5</td>
<td>36.5</td>
<td>0.06</td>
<td></td>
</tr>
<tr>
<td>System Test</td>
<td>9.5</td>
<td>19.5</td>
<td>81.0</td>
<td>64.5</td>
<td>18.5</td>
<td>25.0</td>
<td>20.5</td>
<td>22.5</td>
<td>0.32</td>
</tr>
<tr>
<td>System Debug</td>
<td>2.0</td>
<td>21.5</td>
<td>90.0</td>
<td>43.5</td>
<td>18.5</td>
<td>3.0</td>
<td>1.0</td>
<td>18.0</td>
<td>0.96</td>
</tr>
<tr>
<td>Total</td>
<td>323.0</td>
<td>304.0</td>
<td>704.0</td>
<td>416.0</td>
<td>352.5</td>
<td>276.0</td>
<td>314.5</td>
<td>352.5</td>
<td>1.01</td>
</tr>
<tr>
<td>Average</td>
<td>443.7</td>
<td>342.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

we present the \( \log_{10} \) value instead of the actual value. Figure 6.4 shows (the logarithm for) the average value of each quality metric per group half. What can be seen in the figure is that exception-based and contract-based groups have similar values for most of the metrics. Since the \( \log_{10} \) value of values smaller than 1 is negative, the differences between the groups appear to be greater for certain metrics even though the differences are actually rather small. There is in fact a large difference for some metrics, but this is caused by single outliers in one of the class halves rather than by a general difference between the halves. Due to space limitations, only the metrics number are shown in the diagram. The metrics are listed with their corresponding number in Table 6.4.

**Hypothesis 1 - time consumption**

We next proceed with tests of our hypotheses. Our first null hypothesis stated that the time needed for development would be the same for groups using the contract-based method as for groups using the exception-based method. As can be seen in Table 6.1, the average exception-based group spent 444 hours to complete the assignment, whereas the average for the contract-based groups was 343 hours. The difference is 23%, but this can be explained by the extraordinary group X3 mentioned in the previous section. To determine whether there was any significance in the differences between the two halves of the class we conducted a one-tailed t-test [52]. A t-test can be used to compare two sets of data when the mean and standard deviation of the two sets are not known. The first step is to calculate the so called \( t_0 \) value for the set of data points. The \( t_0 \) value is then compared to known t-values for chosen significance levels. If the \( t_0 \) value is greater than the t value for a particular level, the null hypothesis should be rejected and the alternative hypothesis accepted. The \( t_0 \)-value should be greater than 3.143 in order for the difference to be significant at the 1% level and greater than 1.943 to be significant at the 5% level. As can be seen in the rightmost column in Table 6.1, only implementation is significant according
6.1. A CONTROLLED EXPERIMENT IN AN ACADEMIC ...

to these criteria. The activity subsystem design is significant on the 10% level (larger than 1.44), thus giving a weak indication of a difference. In all other activities, the differences in the measures are not statistically significant. No statistically significant difference in total time can be observed.

The conclusion that can be drawn from the t-test is that the implementation activity shows a statistically significant difference. Our first null hypothesis can, however, not be rejected since the difference in total time is not significant.

Hypothesis 2 - work satisfaction

Our second null hypothesis stated that work satisfaction would be the same for groups using the contract-based method as for groups using the exception-based method. Table 6.2 presents the work satisfaction per group. The measures are expressed as percentages of the total rated time in each group.

<table>
<thead>
<tr>
<th>Group</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bad</td>
</tr>
<tr>
<td>X1</td>
<td>3.954</td>
</tr>
<tr>
<td>X2</td>
<td>15.48</td>
</tr>
<tr>
<td>X3</td>
<td>9.552</td>
</tr>
<tr>
<td>C1</td>
<td>0.375</td>
</tr>
<tr>
<td>C2</td>
<td>9.692</td>
</tr>
<tr>
<td>C3</td>
<td>13.28</td>
</tr>
<tr>
<td>C4</td>
<td>0</td>
</tr>
<tr>
<td>C5</td>
<td>11.14</td>
</tr>
</tbody>
</table>

To evaluate whether there were statistically significant differences between the two halves of the class, Mann-Whitney-tests [1] were done on the frequencies of "bad", "OK" and "good". The test takes as its input a ranking of the values and the number of values in each data set. The output produced is a so-called U-value, which is used as an index in a table to find the significance level. The results are shown in Table 6.3.

The tail probability is the probability of obtaining the ranks with random values. This probability should therefore be low, commonly 5% or 1%. As can be seen in Table 6.3 the tail probabilities obtained from the tests are 60%, 70% and 20% for the outcomes "bad", "OK" and "good" respectively. The results show that there is no statistically significant difference between the two halves of the class based on their rates of "bad"- and "OK". The "good" grade was selected for a larger fraction of time by the contract half with a significance level of 20%. Altogether, however, this is not sufficient to reject the null hypothesis.
Table 6.3: Mann-Whitney U-test for work satisfaction

<table>
<thead>
<tr>
<th>Grade</th>
<th>Bad</th>
<th>OK</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. exc. groups</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>No. contr. groups</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Sum of exc. ranks</td>
<td>15</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>U-metric</td>
<td>6</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>Tail probability</td>
<td>60%</td>
<td>70%</td>
<td>20%</td>
</tr>
</tbody>
</table>

We can conclude that even though there seems to be an indication that groups in the "contract" half have greater work satisfaction, it cannot be statistically verified with an acceptable level of confidence.

Hypothesis 3 - program quality

The third and final null hypothesis stated that program quality would be the same for groups using contracts as for groups using exceptions. To evaluate this objectively, the code for all groups was analyzed using the built-in metrics tools in Together. All 47 metrics were calculated for the code from all groups and a t-test was then done on the distribution of each of the metrics. Table 6.4 lists the average values for the metrics per half class together with the probability of obtaining the t value calculated for the groups in the halves ($P(t)$). For a full metrics definition, see [79].

As can be seen in Table 6.4, the $P(t)$ values indicate that the difference between the groups is not significant at the 1% or 5% levels for any of the metrics. For that to be true, the values should be lower than 0.01 and 0.05 respectively. There are, however, two metrics for which the difference is significant at the 10% level. The maximum number of levels (number 22), i.e. the depth of nested if, for and while statements is significant on the 10%-level, as is the weighted methods per class version 2 (number 47). The latter metric is derived from the number of methods per class and the number of parameters per method. The groups in the "exception" half used deeper nesting of statements and more or more complex methods per class. Neither of the method documents explicitly addresses any of these metrics, and the differences are therefore difficult to derive from the methods. More research is needed to determine in what way the methods affect these metrics. The conclusion regarding programming method and program quality is that the null hypothesis cannot be rejected.

---

1In this table, we present the probability that the differences might be random ($P(t)$) instead of the actual t value due to the default output of Together.
Table 6.4: t-test for program quality metrics

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>36.0</td>
<td>26.4</td>
<td>0.46</td>
<td>(1) Attribute Complexity</td>
</tr>
<tr>
<td>29.6</td>
<td>19.9</td>
<td>0.34</td>
<td>(2) Attribute Hiding Factor</td>
</tr>
<tr>
<td>10.0</td>
<td>9.4</td>
<td>0.96</td>
<td>(3) Attribute Inheritance Factor</td>
</tr>
<tr>
<td>14.0</td>
<td>18.0</td>
<td>0.22</td>
<td>(4) Coupling Between Objects</td>
</tr>
<tr>
<td>171</td>
<td>122</td>
<td>0.37</td>
<td>(5) Cyclomatic Complexity</td>
</tr>
<tr>
<td>14.0</td>
<td>11.0</td>
<td>0.68</td>
<td>(6) Coupling Factor</td>
</tr>
<tr>
<td>3.3</td>
<td>5.0</td>
<td>0.26</td>
<td>(7) Comment Ratio</td>
</tr>
<tr>
<td>7.0</td>
<td>8.8</td>
<td>0.92</td>
<td>(8) Data Abstraction Coupling</td>
</tr>
<tr>
<td>6.0</td>
<td>5.8</td>
<td>0.81</td>
<td>(9) Depth Of Inheritance Hierarchy</td>
</tr>
<tr>
<td>12.7</td>
<td>15.8</td>
<td>0.75</td>
<td>(10) FanOut</td>
</tr>
<tr>
<td>12.3</td>
<td>10.6</td>
<td>0.77</td>
<td>(11) Halstead Difficulty</td>
</tr>
<tr>
<td>9784</td>
<td>8344.6</td>
<td>0.64</td>
<td>(12) Halstead Effort</td>
</tr>
<tr>
<td>3097</td>
<td>4094</td>
<td>0.96</td>
<td>(13) Halstead Program Length</td>
</tr>
<tr>
<td>914</td>
<td>899</td>
<td>0.96</td>
<td>(14) Halstead Program Vocabulary</td>
</tr>
<tr>
<td>21274</td>
<td>21373</td>
<td>0.98</td>
<td>(15) Halstead Program Volume</td>
</tr>
<tr>
<td>2404</td>
<td>2196</td>
<td>0.67</td>
<td>(16) Lines Of Code</td>
</tr>
<tr>
<td>258</td>
<td>244</td>
<td>0.94</td>
<td>(17) Lack of Cohesion of Methods 1</td>
</tr>
<tr>
<td>96.0</td>
<td>87.2</td>
<td>0.82</td>
<td>(18) Lack of Cohesion of Methods 2</td>
</tr>
<tr>
<td>112</td>
<td>124</td>
<td>0.64</td>
<td>(19) Lack of Cohesion of Methods 3</td>
</tr>
<tr>
<td>16.3</td>
<td>23.4</td>
<td>0.47</td>
<td>(20) Method Hiding Factor</td>
</tr>
<tr>
<td>3.0</td>
<td>21.4</td>
<td>0.35</td>
<td>(21) Method Inheritance Factor</td>
</tr>
<tr>
<td>1.7</td>
<td>0.0</td>
<td>0.08</td>
<td>(22) Max Number Of Levels</td>
</tr>
<tr>
<td>1.3</td>
<td>5.6</td>
<td>0.26</td>
<td>(23) Max Number Of Parameters</td>
</tr>
<tr>
<td>1.8</td>
<td>19.2</td>
<td>0.86</td>
<td>(24) Max Size Of Operation</td>
</tr>
<tr>
<td>17.0</td>
<td>24.6</td>
<td>0.87</td>
<td>(25) Number of Attributes</td>
</tr>
<tr>
<td>20.0</td>
<td>17.6</td>
<td>0.59</td>
<td>(26) Number of Added Methods</td>
</tr>
<tr>
<td>17.3</td>
<td>18.8</td>
<td>0.87</td>
<td>(27) Number of Classes</td>
</tr>
<tr>
<td>5.3</td>
<td>5.8</td>
<td>0.91</td>
<td>(28) Number of Child Classes</td>
</tr>
<tr>
<td>2.0</td>
<td>2.2</td>
<td>0.79</td>
<td>(29) Number of Constructors</td>
</tr>
<tr>
<td>3.6</td>
<td>1.2</td>
<td>0.12</td>
<td>(30) Number of Import Statements</td>
</tr>
<tr>
<td>90.0</td>
<td>58.0</td>
<td>0.86</td>
<td>(31) Number of Members</td>
</tr>
<tr>
<td>14.3</td>
<td>31.0</td>
<td>0.17</td>
<td>(32) Number of Operations</td>
</tr>
<tr>
<td>1.3</td>
<td>3.6</td>
<td>0.39</td>
<td>(33) Number of Overridden Methods</td>
</tr>
<tr>
<td>2025</td>
<td>2178</td>
<td>0.75</td>
<td>(34) Number of Operands</td>
</tr>
<tr>
<td>1872</td>
<td>2016</td>
<td>0.68</td>
<td>(35) Number of Operators</td>
</tr>
<tr>
<td>125</td>
<td>59.2</td>
<td>0.44</td>
<td>(36) Number of Remote Methods</td>
</tr>
<tr>
<td>682</td>
<td>749</td>
<td>0.69</td>
<td>(37) Number of Unique Operands</td>
</tr>
<tr>
<td>11.7</td>
<td>24.6</td>
<td>0.47</td>
<td>(38) Polymorphism Factor</td>
</tr>
<tr>
<td>8.7</td>
<td>12.8</td>
<td>0.29</td>
<td>(39) Percentage Package Members</td>
</tr>
<tr>
<td>14.5</td>
<td>16.0</td>
<td>0.84</td>
<td>(40) Percentage Private Members</td>
</tr>
<tr>
<td>0.3</td>
<td>0.8</td>
<td>0.46</td>
<td>(41) Percentage Protected Members</td>
</tr>
<tr>
<td>90.1</td>
<td>54.8</td>
<td>0.44</td>
<td>(42) Percentage Public Members</td>
</tr>
<tr>
<td>129</td>
<td>27.3</td>
<td>0.41</td>
<td>(43) Response For Class</td>
</tr>
<tr>
<td>1.3</td>
<td>5.8</td>
<td>0.75</td>
<td>(44) True Comment Ratio</td>
</tr>
<tr>
<td>215</td>
<td>157</td>
<td>0.31</td>
<td>(45) Weighted Methods/Class 1</td>
</tr>
<tr>
<td>100</td>
<td>70.2</td>
<td>0.06</td>
<td>(46) Weighted Methods/Class 2</td>
</tr>
</tbody>
</table>

6.1.4 Validity

There are different areas in which validity threats must be considered, as presented in Sect. 2.2.3. For this experiment we have viewed the threats to the different areas as follows:

Conclusion validity concerns the question of whether the results obtained
are statistically significant. The major problem of the experiment is the low number of test subjects. We found that the results were significant for certain activities, but not for all. Altogether, we were not able to draw any strong conclusions from the results, only some indications on differences.

**Internal validity** concerns the relationship between the treatment and the outcome. We tried to minimize some of the background factors by randomizing the groups and thus trying to make them as similar as possible. To ensure that all students received the same information, a web-based help desk was set up in the form of an FAQ where all questions that were not related to the methods were posted and accompanied by our answers. This minimized the information gap that might otherwise have occurred because some groups asked more questions than others did. This gap could have ruined the validity of the data gathered during the course. Another problem has to do with the effect of memory on reliability; if too much time passes between the actual work and the reporting of the work, the results are less reliable since people tend to forget. To avoid this we collected data forms regularly in order to put pressure on the students to report their work promptly. We believe that we have reasonably high internal validity although the varying abilities of the students still remain a problem.

**Construct validity** refers to the experimental setting and how that reflects the construct under study. An example of this is the exception-based method that we used for comparison in the experiment. Could an exception-based method be considered a normal programming method? For reasons described in Sect. 6.1.1, we believe that the answer is yes. We also made the two method documents as similar as possible in all aspects except those concerning error-handling in order to minimize background noise resulting from unintentional differences. Another issue that might affect the construct validity is the usage of students instead of real software developers. The students can be seen as a construct representing the real entity software developer. Using students as subjects is a debated issue, [68] and the relation between students and their professional counterparts need more research to be firmly established.

**External validity** concerns generalization. One question about the external validity is whether it is possible to transfer the results to other settings, such as industrial settings. We believe that it is to a certain extent. While the assignment and work environment are perhaps not identical to those in industry, program development in groups is a common work method.
6.1.5 Summary and Conclusions

We have presented an experiment designed to evaluate whether using a contract-based program development method would shorten development time, improve work satisfaction and increase the quality of the resulting software program. We were unable to draw any strong conclusions regarding any of our hypotheses for reasons explained below. Only indications of improvements were found. The positive effect of the experiment was that it did not show any drawbacks of using the contract-based method, but rather weak positive indications. This enabled us to initiate a larger scale industrial experiment where the contract-based method was introduced in a live project. The results of this industrial introduction are presented in Sect. 6.2.

6.1.6 Lessons Learned

Due to a number of reasons, the results from the experiment were statistically weak. We have tried to identify these reasons to be able to construct future experiments in a better fashion.

- Too few data points - Since we wanted to see what effects different program development techniques had on the development process, we needed a sufficiently complex assignment. A larger assignment brings forward the strong and weak points of a development method and also implies that a number of people need to work together to reach the goal. Students worked in groups of four, thus reducing the number of data points by a factor four. This had as a consequence that very few statistically significant measures could be obtained. Most statistical methods require at least 20-30 data points to produce usable results. This problem can either be remedied by including more test subjects or to use individual test subjects as the smallest entity rather than groups. Both these solutions have drawbacks. More test subjects might not be available or only available at a too large cost. Using single test subjects instead of groups makes it impossible to mimic industrial projects, which are normally conducted in groups.

- Diversity in competence - Student competence was a large factor and clearly biased some data. This problem is universal in the sense that individual competence or lack thereof seems to have a larger effect on the outcome of any project or experiment as compared to most other factors [51]. From this, one could draw the conclusion to only hire the most competent software developers, but in reality this is difficult to assess a priori and hence methods for improving the not-so-competent employees’ abilities are still in need. Regarding experiments, one conclusion that can be drawn from this finding is to use statistical designs that eliminate or
control the individual test subjects’ abilities. As will be shown later in this chapter, a factorial experiment is one such statistical method.

- Ethical considerations - Since it was considered unethical to teach one class half a program development method that was thought to be inferior, we had to make both methods rather good. In the end, the methods had large portions in common, another factor that weakened the results. The underlying ethical dilemma was that students were used as test subjects and they hence had to invest time and money for taking the course in which the experiment was conducted. The investments for the groups using the baseline method would have been wasted had that method been teaching them a bad way of developing software. Teaching these groups an inferior method, however similar to industrial practice, was hence ruled out as unethical. A remedy to this problem is to have all test subjects use all methods being compared, thus making it possible to have one of the methods being inferior but realistic. This can also be done using factorial designs as proposed previously and explored in more detail in Sect. 6.3.

All these aspects put together made for a statistically weak result from the experiment.

### 6.2 An Industrial Case Study on Structured Semantics

This section presents an industrial evaluation designed to evaluate what benefits industry could get by using a contract-based method, Semla, as presented in Chapt. 5. This was a natural continuation of the work presented in Sect. 6.1 where the method was evaluated in an academic environment. This section presents how the method was introduced in the industrial project, how it was received and how the project members experienced working with the method.

#### 6.2.1 Environment

The study was performed at a major Swedish telecom company in a project designed to develop a platform for technical services such as virtual private networks. The project was active, i.e. producing code, for roughly eight months, followed by approximately four months testing and validation. At its largest, the project included thirty people, most of them locally employed, but also including between five and ten people from other branches of the company. A group from Italy and another from the Netherlands were active in the project for a large portion of the time. The project had members with backgrounds in both computer science and business administration. Although the project was
typical in terms of size, organization and management, it introduced a number of new issues to its members. They had a new programming paradigm (OO), a new programming language (Java) and a new process (RUP) to cope with, along with the method. From interacting with the project members, we know that these new issues were seen as burdens and this might have affected how the method was received and used.

6.2.2 Execution

In this subsection we present the introduction and follow-up of the method in the project and the selection of variables and measurements.

Introduction of the method

The method was introduced in a half-day seminar with all project members attending. In this seminar, general aspects such as semantic issues and contracts were presented as well as the method itself. A document describing the method was also handed out to the project members.

Follow-up

To monitor and guide in the application of the method, the researchers joined the project group meetings as often as possible. The researchers were also available for questions at all times via mail. In retrospect, it is clear that the collective participation of the researchers was not enough. Groups reluctant to change their way of working were hard to control and many decisions on design issues were made in the absence of the researchers.

Variables Selection and Measurements

As mentioned in Sect. 3.6, we did encounter some problems with the data collection and ended up focusing on qualitative rather than quantitative aspects. The qualitative variables we were eventually focused on were the following:

- project members’ attitudes towards using the method
- how the project members experienced working with the method, and what advantages or disadvantages the method had on their work
- perceived quality of the resulting product

These variables were measured with anonymous questionnaires that were collected for all project members, even though not all replies are included in the analysis.
6.2.3 Results

In this subsection we present the results from the questionnaires answered by the participants in the project. The project members answered the same questionnaire at four different occasions at two months intervals. All questionnaires are coded in the same fashion, 1 is less or worse, 2 is same as without, 3 is better or more as compared to not using the method. A fourth answer, no opinion was also possible and was chosen by around 30% of the respondents. This answer is not included in this section. The first four points in the graphs are arithmetic average values for the respective questionnaire answers, and since they are measured on an ordinal scale these values can only be seen as indicators of the majority of the answers. The fifth point in the graphs indicates the grand average of all answers to all four questionnaires.

Since we focused on code design and implementation, only answers from people involved in design and programming were included in the results. This group will be denominated DPs and includes 14 people. Since roles change throughout the project due to the shift in focus from development to testing, different people have answered at different occasions.

Work Satisfaction

This variable was measured to find out if working with the method would give a higher work satisfaction than when not working with the method. The question in the questionnaire was: How much satisfaction do you find in your work when using [the method]? As can be seen in Fig.6.5, the work satisfaction was initially very high and it was at all times higher than or equal to when not working with the method. The downward trend from the initial high value could perhaps indicate that the expectations of the method were high, but that the project

![Figure 6.5: Work satisfaction: measurement average and grand average](image-url)
members realized that it was not the expected silver bullet, only another good tool. The trend might also be explained by too little follow-up and monitoring.

**Intuitivity**

Intuitivity was measured to find out if working with the method felt intuitive or if it demanded a new way of thinking that was not perceived as intuitive. The question in the questionnaire was: *How intuitive is it to work with [the method]?* Fig.6.6 shows that the opinions of the DPs regarding intuitivity are mostly above 2, that is equally intuitive or more intuitive than when not using the method. This supports that contracts can be introduced without too much complexity, which is one of the design goals of the method. The no opinion-rate for this question was 47%, which makes it hard to draw any general conclusions.

**Communication Quality**

Since contracts can be seen as a communication channel between different designers and programmers, we measured this in the questionnaires. The corresponding question was: *How well does the communications with your colleagues work?* As can be seen in Fig. 6.7, the project members initially had high expectations, probably caused by the positive influence from the researchers and the managers. In spite of the somewhat negative trend, the values never drop down to 2, that is, the communication quality was always perceived as higher than without the method.

**Design and Programming Effort**

Design effort and programming effort were measured to find out what the DPs thought about the effort needed to design and implement their project when
CHAPTER 6. EMPIRICAL EVALUATIONS

Communication quality

2000-06 - 0.8
2000-08 - 0.4
2000-09 - 2.8
2000-12 - 0.8
Average

Using the method. The questions asked were: How much effort is needed during design/programming? Fig. 6.8 shows that it was perceived that more effort was needed during design as compared to not using the method, something we had expected from the increased focus on semantic issues. Writing down pre- and postconditions and thinking through the solutions may demand some extra effort and around half of the respondents thought so, as can be understood by the 2.5 grand average. Examining the results shows that half of the answers were 3 and the other half 2. Fig. 6.9 shows that programming was not perceived as requiring as much extra effort as design did when not using the method, something we had also expected. We had, however, hoped for an even larger difference, perhaps figures below 2, i.e. that programming would demand less effort than when not using the method. Why this is not the case can only be speculated, but even when programming, DPs not used to contracts might need some extra effort to use the new methodology.
6.2. AN INDUSTRIAL CASE STUDY ON STRUCTURED ... 

Figure 6.9: Programming effort: measurement average and grand average

Design and Program Quality

We believe that using contracts during design and programming will improve the quality of the design produced since working with contracts promote a better understanding of the system. The questions asked were: *How is the quality of the result from design/programming?* One issue that must be taken into account is that these two questions had a *no opinion*-rate of 60% and 52% respectively, so the figures might not be representative. Fig. 6.10 shows how the project members think that the design quality is affected by using the method. The figures were above or equal to 2, which means that the average DP believed the design quality to be higher or equal to when not using the method. Fig. 6.11 shows the program quality. The figures at all times are higher than 2, that is, on average, the DPs believe that program quality is higher than when not using the method. The overall results from the answers to these two questions is that
the DPs thought that the resulting design and program quality was improved by the method. Again, the high no-opinion-rates makes it hard to draw any general conclusions.

### 6.2.4 Validity

There are a number of threats to the validity of any experiment or case study as discussed in Sect. 2.2.3, and we have tried to identify the threats to the presented case study in the following list.

**Conclusion validity** concerns the question if the results obtained are statistically significant. Since the number of test subjects that answered our questions was low, we were unable to derive any statistically strong results and the figures should be seen as weak indications. The large rate of no-opinion answers also lowers the conclusion validity.

**Internal validity** concerns the relationship between background variables and the main variable. In a controlled experiment, say in an academic environment, we would have tried to minimize some of the background factors by randomization and blocking. This was not possible since we did not perform a controlled experiment and hence could not control background variables. A large problem here is that the project members were influenced by a variety of issues such as programming languages, deadlines, previous knowledge, attitude and others. However, since we measured the subjective opinions of the DPs we have to rely on their ability to distinguish the effects of the method from other factors.

**Construct validity** refers to how well the theory and the observation correspond. Since this case study was performed on a real project, no extra theoretical constructs were needed.
External validity concerns generalization. One question concerning the external validity is if it is possible to transfer the results to other settings, such as other industrial projects. We believe that it is to a certain extent. This particular project was largely what was considered "normal" by the project members in terms of size, organization and management. What sets this project apart from other projects in the same category is the amount of new issues involved, as discussed in Sect. 6.2.1.

6.2.5 Summary and Conclusions

We have presented a case study performed in an industrial project where a contract-based way of working and thinking was introduced. The objective for the case study was to supply empirical evidence that an informal, contract-based programming method is intuitive and increases work satisfaction and communication quality. A contract-based method has a focus on semantic specifications, so we expect its application to require a higher effort on design than without the method. We also expect an increase in work satisfaction and communication quality, due to the increased semantic precision. Its informal nature is intended to make the method intuitive. In all these respects, the case study gives a weak confirmative indication. Additional goals are to increase product quality and reduce the overall effort. The study is weakly confirmative regarding the quality, as perceived by the DPs, but not regarding the programming effort. However, an increased effort is a natural consequence of every new method, but is expected to decrease with experience. In summary, the case study lets us believe that an informal, contract-based method like the one studied, is indeed profitable, although more research is needed to verify this.

6.2.6 Lessons Learned

The case study experienced certain problems that unfortunately made it difficult to obtain statistically significant results. These problems are listed below with some comments on how to avoid them in other similar endeavors.

- Project size - The case study would have benefitted from a smaller project size, from a control point of view. As it turned out, the project assigned to us by our industrial partner was too big to handle properly by only two researchers. One might argue that another project should have been chosen, but due to time restrictions and the will of the industry partner, it was not an option. The larger size made it more difficult to inform the project members on how to incorporate the suggested method into their normal way of working. It also made it more difficult to see where more information and support was needed. The project members did not know each other well either, which made peer helping more difficult. The
simple remedy to this problem is of course to make sure that the size of the project is within reasonable limits. It is worth noting, however, that different kinds of case studies put different demands on the project size and that hard numbers are impossible to give. On the one hand, a large project is desirable to obtain as many data points as possible. On the other hand, large projects are hard to control and might hence produce lower quality data due to disturbing factors. One should, before commencing with the study of an industrial project, evaluate what project size is appropriate and what measures to take if the size is not optimal.

• Lack of control - The project was not controlled enough due to two major reasons. First, the project was too big, as discussed above. Control is time- and personnel-consuming and if the number of project participants is too large, there is simply not enough time and personnel to control it. The solution is either to chose a smaller project, which might not be possible, or to include more research personnel. The second reason was an unwillingness by certain project members, especially some in dominant positions, to change their way of thinking and working. A successful study is more easily done when all participants actively work towards the same goal. The remedy to this problem is to only include those who are genuinely interested in participating. The obvious drawback of a non-random selection of subjects is that it biases the sample and makes it difficult to separate the effects from the item under observation from the effects stemming from the bias.

• Unstable environment - Not only did the project hire people under way which made the information spreading more difficult, the project also introduced its members to a new programming language (Java) and a new software engineering methodology (RUP). Since the aim of the case study was to evaluate the effects of introducing a more semantically-oriented method and to compare the current project to previously finished projects that had used other methods, it would have been preferable had all background variables other than the methods been stable. The effects of the semantic method were to a large extent shadowed by the larger effects stemming from the other changes, thus making any conclusion regarding the merits of the method harder to draw.
6.3 A Controlled Experiment in an Academic Environment Using a Factorial Design

This section presents the third evaluation in the series of empirical evaluations presented in this thesis. The experiment was carried out in an academic environment aiming to compare how different methods of documenting semantic information affect software reuse. More specifically, we wanted to measure if there were any differences between the methods with regard to the time needed to implement changes to existing software. Four methods of documentation were used: executable contracts, non-executable contracts, Javadoc-style documentation and sequence diagrams. Executable contracts were implemented using standard assert-statements, non-executable contracts adhered to the rules presented in the Semla method as presented in Chapt. 5 and Javadoc-style was a C++ version of the well-known Java documenting style. Sequence diagrams were in the wide-spread UML form. The main difference to the other two experiments was that four methods were compared instead of just two. The reason for including more than just two methods in the evaluation was to widen the scope to include more of the commonly used methods and hence covering more territory in the evaluation. To increase the focus of the documentation, the focus was put on reusing existing software rather than on producing code from scratch. Reuse increases the complexity by having the subjects reuse code written by someone else. The increased complexity gained from the reuse focus made it possible to keep the size of the assignments down and made the assignments solvable within the given time limit of three hours.

The experimental setup used was a factorial design with documentation method as the treatment, and with test persons and assignments as blocking variables. This setup makes it possible to gauge both treatment and blocking variable effects, something that is much more difficult using common setups such as between-groups comparisons. This is perhaps the most important difference as compared to the previous evaluations.

The remainder of the section is structured as follows: first the experiment design is discussed followed by the results from the experiment. Next, validity threats are discussed and analysed followed by some conclusions and lessons learned from the experiment.

6.3.1 Experiment Design

The experiment compared four documentation methods, using 15 students as test subjects who implemented change requests to four programs written in C++. The choice of language was made based on the fact that the students had the most training in C++. The experiment was performed over a period of four consecutive days where each day contained one three-hour work session in which
the test subjects worked on one assignment. After the four days, all students had solved all assignments and worked with all methods. The order in which the assignments and methods were assigned to the subjects was randomized. More details on the specific aspects of the experiment can be found in the following subsections.

Hypotheses

The null-hypothesis was that all methods were equally good with regard to the time needed to solve the assignments, and the corresponding alternative hypothesis that there were differences between the methods.

\[ H_0 : time_{\text{one-method}} = time_{\text{other-methods}} \]

\[ H_1 : time_{\text{one-method}} < time_{\text{other-methods}} \]

The hypothesis was tested using an ANOVA-test [9] on the recorded time for completing the assignments.

Statistical Model

Every experiment is based on an underlying statistical model [9], defining the assumptions made for the experiment. The model for the experiment presented in this section is an additive model where no interaction effects are considered to exist. The model below states that the individual values \( y_{ijk} \) can be seen as the sum of deviations from the average value \( \eta \).

\[ y_{ijk} = \eta + \beta_i + \gamma_j + \tau_k + \epsilon_{ijk} \]

\[ df : 60 = 1 + 3 + 14 + 3 + 39 \]

In the formula, \( \beta \) represents the effect of assignments, \( \gamma \) the effect of the test subjects and \( \tau \) the effect of the treatments. Residuals are denoted \( \epsilon \) and represents the effects that cannot be contributed to any of the other parameters. This is often referred to as noise. Finally, \( df \) describes the degrees of freedom for the respective parameters.

Treatments

The aim of the experiment was to compare different documentation methods that focus on semantic aspects in the software. Since neither of the methods give any structural or syntactic help, this was supplied to the test subjects independent of the documentation method as class diagrams and comments in
6.3. A CONTROLLED EXPERIMENT ... FACTORIAL DESIGN

<table>
<thead>
<tr>
<th>Level of abstraction</th>
<th>Class diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>Description of assignment</td>
</tr>
<tr>
<td>Semantic</td>
<td>Documentation according to method</td>
</tr>
<tr>
<td>Syntactic</td>
<td>Comments in code</td>
</tr>
</tbody>
</table>

Figure 6.12: Overview of documentation provided to the subjects

the code. A graphical representation of the documentation provided to the test subjects can be seen in Fig. 6.12. In a comparitative experiment, such as the one presented here, it is very important that the treatments being compared actually are comparable. The treatments must have the same area of application such that each could replace any of the other treatments. In our experiment, the area of application is semantic documentation, and all treatments must therefore focus on that area. The four methods are listed below along with a short description and motivation.

- Executable contracts - Executable contracts are either written as executable commands in the same language as the actual program code or in a meta-language and later compiled into the code. A contract is written for every method comprised of a precondition expressing what should hold before calling the method and, where applicable, a postcondition expressing the effects of the method. Using this technique increases testability and adds some verifiability to the software. This method is pushed by a number of languages and methodologies such as Eiffel [48] and iContract for Java [36]. It also mainly conforms to the ideas presented in Semla. Below is an example of executable contracts for the method `getPerson`. In the example below, only two preconditions are present. No postcondition is provided in this case.

```c++
//-------------------------------------------
// getPerson(aPosition):
// returns person at position aPosition
//-------------------------------------------
Person* AddressBook::getPerson(int aPosition)
{
    assert(0<=aPosition);
    assert(aPosition<size());
```
• Non-executable contracts - Non-executable contracts are similar to executable contracts, but are written as comments that can be program code, plain English or combinations of both as described in Chapt. 5. Since non-executable contracts have a less formal structure, they are more easily read and can also express conditions that are not executable. The testability of executable contracts is naturally lost, but non-executable contracts still help the software developers focus on semantic issues, thus influencing the quality of the software in a positive way. Below is an example of non-executable contracts for the method `getPerson`.

```c++
Person* AddressBook::getPerson(int aPosition)
{
    return LL->getElement(aPosition);
}
```

• Javadoc style - Javadoc is a technique using special keywords and plain text that is written as special comments and is separately compiled to generate the documentation. It is used extensively by Java developers and is an easy way of producing and maintaining documentation. Since the application language in the experiment was C++, some of the features of Javadoc had to be emulated, but the general structure and style was the same as for Java. Below is an example of Javadoc-style documentation for the method `getPerson`.

```c++
// getPerson(aPosition): returns person at position aPosition
// param: aPosition - the position to return
// exception: an exception is thrown if aPosition is out of range
Person* AddressBook::getPerson(int aPosition)
{
    return LL->getElement(aPosition);
}
```
• Sequence diagrams - Sequence diagrams are used in UML [32] as a way of documenting interactions between software parts. They contain information on methods and their invocations as well as conditions and call sequences. Figure 6.13 contains an example of sequence diagram for the method `getPerson`.

These four methods for documenting software might at a first glance not be all too similar, but they all focus on the semantic aspects and provide roughly the same information to the developer, albeit in different ways. The four methods also manage to cover most of the mainstream software documentation methods that are currently in use.

Assignments

In the experiment, we needed four assignments that could be combined with the documentation methods such that the students never had to solve the same assignment twice. These assignments had to be solvable by the test subjects within the given time interval of three hours and could therefore not be too big or too complex. In order for the documentation methods to have any effect on the time needed for implementing the assignments, the assignments still needed to be non-trivial. Initially, we hoped to find assignments that had been used
in previous experiments, but that proved more difficult than first expected. The assignments that we did find either focused on other aspects, were too big or too small or simply demanded more work to be useful than actually writing new assignments from scratch. We managed to find one assignment, coffee machine, that we ended up using. It was found in an article by Alistair Cockburn [15] who has been investigating the effects of OO design methods on software comprehension. The other three assignments were constructed by the researchers performing the experiment. Below is a listing of the assignments along with a short description, code size and the change requests the students were to implement.

- **Coffee Machine** - This program emulates a coffee machine that accepts coins and produces coffee and other beverages. The program size was 550 lines of code (LOC). The change request was to implement functionality to change the strength of the coffee and the other beverages.

- **ATM** - The ATM program was a simple implementation of a money dispenser where the user can insert a card, enter a pin code and withdraw money. The program size was 450 LOC. The change request was to add functionality to handle more than one account tied to the same bank card and to enable transfers between accounts.

- **Address Book** - The address book assignment was a simple implementation of an address book containing people and addresses that could be edited in various ways. The program size was 700 LOC. The change request was to add handling of two or more address books as well as synchronization between them.

- **Booking system** - The booking system handles resources such as rooms, persons and times and could for instance be used in a university environment or at a larger corporation. The program size was 800 LOC. The change request was to check for double bookings and to add extraction of available times and rooms.

During and after the experiment, students argued that certain assignments were more difficult than others but as will be seen later, no statistically significant difference among them was found.

**Test Subjects and Environment**

The test subjects were computer science students in their third year with a decent background in programming using C++. They had all been exposed to all four documentation methods in previous courses, although not extensively.

The strategic environment in which the experiment was conducted was a university course and the physical environment a computer lab at the university.
6.3. A CONTROLLED EXPERIMENT ... FACTORIAL DESIGN

Metrics and Measurements

Time for completion of the assignments was the main metric since it is a good indication of how easy or hard the different treatments made implementing the changes. Time was measured by the test subjects, but the process was coordinated and supervised by a test leader. The quality of the resulting software was controlled as we wanted all subjects to produce software of roughly the same quality. To ensure this, we provided test cases that the subjects used to verify their solutions.

Statistical Setup

To avoid the problem of diversity among test subjects, all test subjects were subjected to all treatments, but the order in which this was done was varied randomly according to a so called Graeco-Latin square design [9]. Since no test subject can solve the same problem more than once without serious learning effects, the number of assignments needed was equal to the number of treatments used, i.e. four in this case. Every assignment also had to exist in four different versions, one for each treatment. This implies that 16 different assignment-treatment pairs had to be constructed. If we denote the assignments $A, B, C$ and $D$, and the treatments $1, 2, 3$ and $4$, the pairs are denoted $X_n$ where $X \in \{A, B, C, D\}$ and $n \in \{1, 2, 3, 4\}$. The corresponding graeco-Latin squares could then be constructed as in Table 6.5. The figure implies that test subject 1 would

<table>
<thead>
<tr>
<th>Occasion</th>
<th>Subject</th>
<th>First</th>
<th>Second</th>
<th>Third</th>
<th>Fourth</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>$A_1$</td>
<td>$B_2$</td>
<td>$C_3$</td>
<td>$D_4$</td>
<td></td>
</tr>
<tr>
<td>Second</td>
<td>$B_2$</td>
<td>$A_1$</td>
<td>$D_3$</td>
<td>$C_4$</td>
<td></td>
</tr>
<tr>
<td>Third</td>
<td>$C_4$</td>
<td>$D_3$</td>
<td>$A_1$</td>
<td>$B_2$</td>
<td></td>
</tr>
<tr>
<td>Fourth</td>
<td>$D_4$</td>
<td>$C_1$</td>
<td>$B_2$</td>
<td>$A_3$</td>
<td></td>
</tr>
</tbody>
</table>

start with assignment $A$ and programming method 1 ($A_1$), then $B_2$, followed by $C_3$ and $D_4$, and similarly for the other test subjects. This experimental setup is hence a factorial design with individual subjects and assignments as blocking variables and documentation method as treatment. Since the above statistical setup eliminates the learning effect that might occur over time, the analysis can be done using a somewhat simplified analytical setup as seen in Table 6.6. In this setup, the numbers denote the treatment applied to the respective assignment. The analysis for a factorial experiment such as this is an ANOVA-test [9], a test that makes it possible to analyze the effects of both
6.3.2 Results

This section presents the results derived from the experiment and begins with a few descriptive statistics followed by the more formal statistical analysis.

Not all subjects managed to complete their assignments within the given time frame. These subjects were removed from further analysis, leaving eight subjects who managed to solve all their assignments within time. Table 6.7 contains the figures on not completed assignments for each treatment. The table shows that 25% of all assignments were not completed on time and that the non-completed assignments were reasonably well distributed between the treatments, although with fewer in the sequence diagram category. The rest of the analysis includes only the eight subjects who completed all assignments on time. This data reduction might impose a threat to the validity of the experiment. Details on this can be found in Sect. 6.3.3.

Table 6.8 shows average values of the time to completion of the assignments for the documentation methods and the diversions from the grand average.
### Table 6.8: Methods and diversions from grand average

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Div. fr. GA</th>
<th>Div. fr. GA (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grand average (GA)</td>
<td>101.6 min</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Executable contracts</td>
<td>128.5 min</td>
<td>27.0 min</td>
<td>26.6%</td>
</tr>
<tr>
<td>Non-executable contracts</td>
<td>95.8 min</td>
<td>-2.7 min</td>
<td>-2.7%</td>
</tr>
<tr>
<td>Javadoc</td>
<td>89.5 min</td>
<td>-12.0 min</td>
<td>-11.8%</td>
</tr>
<tr>
<td>Sequence diagrams</td>
<td>89.1 min</td>
<td>-12.3 min</td>
<td>-12.2%</td>
</tr>
</tbody>
</table>

As can be seen in the table, the average for executable contracts is 27 minutes more than the grand average. This indicates that the test subjects demanded more time to reuse software where the semantic information was documented using executable contracts. This might be expected, since the syntax is more formalized and expressive than the other methods and thus might be more difficult to read and understand especially with limited training. Javadoc and Sequence diagrams were seen as the two methods of documentation that demanded the least time to read and understand and had average values close to 12 minutes below the grand average. This might be explained by the fact that these methods are rather easy to grasp due to their informal nature, even with limited training and time. As for the last method, non-executable contracts, the average is rather close to the grand average. Since there were differences between treatments, a more detailed analysis was conducted, using an ANOVA-test [9]. This test uses the variations caused by the treatments, blocks and residuals, and then refers the ratios between variations to the F-distribution which is used to assess the significance. If the ratio obtained is larger than the corresponding F-value for the given significance level and degrees of freedom, the variable is considered significant on that level. Table 6.9 shows the results from the ANOVA-analysis.

### Table 6.9: ANOVA-analysis

<table>
<thead>
<tr>
<th>Variable</th>
<th>Variation</th>
<th>Ratio</th>
<th>F(10%)</th>
<th>Significant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treatment(methods)</td>
<td>$s_T^2 = 2756.5$</td>
<td>$s_T^2/s_R^2 = 2.7$</td>
<td>$F_{3,18} = 2.42$</td>
<td>Yes</td>
</tr>
<tr>
<td>Blocks (subjects)</td>
<td>$s_B^2 = 2022.8$</td>
<td>$s_B^2/s_R^2 = 2.0$</td>
<td>$F_{7,18} = 2.08$</td>
<td>No</td>
</tr>
<tr>
<td>Blocks 2 (assignments)</td>
<td>$s_B^2 = 402.6$</td>
<td>$s_B^2/s_R^2 = 0.4$</td>
<td>$F_{3,18} = 2.42$</td>
<td>No</td>
</tr>
<tr>
<td>Residuals</td>
<td>$s_R^2 = 1034.2$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

As can be seen in the table, treatment is significant on the 10%-level, thus indicating that the differences initially found between treatments are statistically significant, although on a rather low level. The null-hypothesis can therefore be rejected at the 90% confidence level.
6.3.3 Validity

As in any experiment (see Sect. 2.2.3), there exists a number of threats to the validity of the experiment [84]. These threats are either inherent to a certain experimental setup, or depending on specific conditions arising only in a particular experiment. Below is a listing of the threats we have identified to this experiment.

**Conclusion validity** - Conclusion validity concerns the question if the results obtained are statistically significant. Due to practical considerations the number of data points is relatively low, thus lowering the statistical power of the analysis. It is therefore more difficult to draw strong conclusions regarding the outcome of the experiment. The statistical setup remedies this problem to a certain extent, but the significance level of 90% is still rather low as a result of the few data points.

**Internal validity** - Internal validity concerns the relationship between the treatment and the outcome. The treatments (documentation methods) themselves are not biased per se, but the implementations used in the experiment might be. This would affect the internal validity, since the experiment would not evaluate the correct treatments. We do not consider this threat as too serious as the implementations were closely scrutinized and evaluated before the experiment took place.

The students might prefer one method over the other, thus indirectly biasing the data, or have more previous experience using one of the methods. They might also have solved the assignments without using the documentation, in spite of the instructions to study and use the documentation. This would affect the internal validity of the experiment.

**Construct validity** - The construct validity, i.e. how well the constructed setting corresponds to real situations is another issue. The assignments were smaller than most real software construction projects, but might nevertheless correspond to parts of a larger project. Still, it is not clear to what extend the results scale up.

The assignments might unintentionally have favored one of the methods, but we believe that this should not be a large problem. The assignments were based on ideas that have been used in university courses utilizing different languages and paradigms. We thus assess this threat to construct validity as rather small.

**External validity** - External validity concerns the generalizability of the results. The experiment was conducted in an off-line setting, which lowers the external validity. It is more difficult to transfer the results from an
off-line experiment to real life projects than for instance from case studies or surveys.

Some of the above threats are more serious and we believe that the few data points poses the greatest threat to the validity of the experiment. The too short time for the sessions is another serious problem, causing the removal of a number of test subjects from further analysis. This in turn decimated the number of data points, thus worsening the aforementioned threat. Another serious threat is the possibility of biased students. If the students did not use the methods, the internal validity would be seriously weakened. The test supervisor saw tendencies of this happening, but did not perceive this threat as too large.

It is important to view the results from the experiment in the light of the possible threats to the validity and to not generalize the results too much.

6.3.4 Summary and Conclusions

We have presented an experiment designed to compare four methods of documenting semantic aspects of software. The aim was to evaluate whether any of the methods were superior with regard to the time needed to implement change requests to an existing program. The results were analyzed using an ANOVA-test on the output data to see what variations in the data could be contributed to the documentation methods. The outcome from the analysis is that the differences between the documentation methods were significant, although on a relatively low level of 10%. The null hypothesis, stating that there are no differences between documentation methods with regard to the time needed to implement change requests is thus rejected on a 90% confidence level.

6.3.5 Lessons Learned

Factorial designs work well, but have certain drawbacks such as time consumption and learning effects. Below, a number of problems associated with factorial designs in general or this experiment in particular, are listed.

- Complexity - A full factorial experiment is more complex to construct and to carry through. Every subject in it is exposed to all treatments and hence has to invest more time than in a traditional experiment where each subject is exposed to one treatment only. In a software development experiment where the test subjects solve assignments, one assignment is needed for every treatment used. In the experiment presented in this section, four treatments were used and hence four assignments were needed to avoid having the subjects solve the same assignment more than once. In addition, since the treatments in this case influenced the internal structure of the assignments, four different versions of each assignment had to
be constructed, resulting in sixteen assignments in total. Even though the versioning of the assignments were easier done than constructing the original four assignments, it still added to the complexity. Keeping track of and monitoring the test subjects in combination with the large number of assignments and versions was also more resource consuming as compared to standard experiments.

- Learning effects - Since all subjects are exposed to all treatments sequentially, potential learning effects must be taken into account. This is not a problem per se, but an additional issue that factorial experiments using human test subjects brings. A correct statistical treatment will of course analyze and gauge the potential learning effects.

To summarize, there are more benefits than drawbacks of using a factorial design as compared to standard comparisons between groups as long as the complexity and extra time consumption needed by the participants can be tolerated.
Chapter 7

Concluding Remarks

This thesis has presented a number of empirical evaluations conducted on software development projects. Certain empirical methods have been seen to function very well in a software development setting. Factorial designs seem very promising in that they can incorporate more aspects that are involved in an experimental setting as compared to traditional methods. The lessons learned from the empirical evaluations presented in the thesis can serve as input for software development researchers and help them choose the proper method for a particular case. The evaluations that failed to provide conclusive evidence still provide knowledge on empirical methods; the successful ones also provide positive indications and perhaps inspiration to further endeavors in the area of empirical evaluations in software development. The thesis has further presented a novel method for handling semantic aspects, that has relaxed restrictions on syntax and formalism as compared to many other methods.

7.1 Contribution

This section summarizes the main contributions produced by the work included in the thesis. The main contribution is an increased understanding of empirical evaluations in a software development setting and the development of a method for handling semantics. Other contributions include a taxonomy of methods for handling semantic aspects and an increased understanding of the relation between contracts and errors.

7.1.1 Empirical Evaluations of Software Development

As has been argued in many articles, e.g [26, 71, 78], experimentation and other empirical evaluations are important for advancing the state-of-the-art in soft-
ware development. Empirical evaluations are an important tool in the process from idea to implemented solution, first from an idea to an academically accepted solution and second from academia to an industrial solution.

Since software development is a practical field aimed at producing high-quality software at a low cost, empirical evaluations are crucial in this field and should be conducted wherever possible. Software development has a complex nature, including both numerous technical aspects and most of all, human factors. It has become apparent in the evaluations that humans, their skills and preferences, play a larger part in determining the outcome than most other factors do. This further enhances the need for proper randomization and control as well as a thought-through and well-chosen empirical methodology in order to evaluate other factors that affect software development work. This thesis illustrates the great potential of using factorial designs as compared to the more common designs where comparisons are made between groups of subjects and where disturbing effects are harder to control. As discussed in Sect. 6.3, the researcher does gain statistical power when using factorial designs, but has to cope with some additional issues. The author would, however, still argue that using factorial designs where applicable produces more reliable data and more easily interpretable results as compared to standard designs. A factorial experiment can evaluate human factors, learning effects and other factors such that their influence on the outcome can be gauged and related to the treatment effects. A bit simplified, a factorial experiment tries to incorporate and evaluate factors that otherwise often have to be seen as noise.

7.1.2 Method for Handling Semantic Aspects

When starting on the work presented in this thesis, a few methods for handling semantic aspects in software development already existed, but they all had in common that they used formal or semi-formal syntax to express the semantics. Case studies in the local industry showed that very basal ways of describing semantics were used, sometimes semantic information was totally omitted both in the code and in the documentation. The gap between existing methods and industrial practice seemed rather big. Trying to bridge this gap, a method with less formalism called Semla (see Chapt. 5 or Appendix A) was developed in cooperation with industry and evaluated in both academic and industrial environments. The aim of the evaluations was to investigate whether a semantic method with less strict syntax and rules than otherwise found, would be of benefit to the mainstream software development industry. Due to a number of external and in some cases internal factors, the evaluations produced rather few statistically significant measures. What has been shown though is that the proposed method can be applied without affecting time consumption and hence cost negatively. Longer-term effects have not been possible to measure due to time limitations. We do believe, however, that increasing the focus on semantics
7.2. IMPACT OF THESIS

In earlier phases such as development and implementation will pay back both in later phases such as maintenance and reuse as well as in terms of overall quality. The method has proven to be easy enough to learn by professional as well as non-professional software developers.

7.1.3 Taxonomy for Semantic Handling Methods

The classification done of the publications found in our surveys (see Chapt. 4) resulted in a taxonomy of how semantic aspects were handled in academia. Other taxonomies incorporating more axes have also been constructed by the author and can help in further classifications and discussions on the handling of semantics. The main purpose of a taxonomy is to structure and categorize information within a certain field to make it easier to view the field set in its own context. In this case, the taxonomy indicates where more research is needed and what methods are available. A software manager can use the taxonomy for deciding on a particular approach, a software research strategist can use it for deciding on future research projects.

7.1.4 Relation Between Contracts and Errors

Another contribution of the thesis is an increased understanding and categorization of the relation between contracts and errors. External errors, i.e. errors produced outside the system being constructed, are best handled by weak contracts, contracts that demand little from the user. Internal errors are errors that are created by the developers in the development phase. These are best handled by strong contracts, since strong contracts make it easier to detect and remove errors before the system is put to use.

7.2 Impact of Thesis

Apart from the scientific contributions discussed above, the thesis work has also had an impact on other areas.

- Impact on Curriculum - The inclusion of a method for handling semantics into the computer science curriculum at Karlstad University has had a rather profound effect on the program development courses. The first programming course the students encounter includes using non-executable contracts as parts of the required documentation for lab assignments. The method is further taught in the second programming course, where the method is stressed more and where more focus is put on correctness and proper program development technique. The method is further used in a number of selective courses, both at campus and in distance form at various educational levels. The perhaps most prominent example of such
a course is a senior year course in object oriented software design where contracts and semantics are one of the main focuses of the course.

- **Pedagogical Aspects** - Using the method in a university course helps to shift the focus from implementation details to a more functional and logical level. Instead of focusing on how an operation performs its task, the focus is on what it does. Undergraduate students often tend to dwell down in details, rather than seeing the bigger picture and the method forces them to think in a more mature way.

- **Industrial Impact** - Our industrial partners have been exposed to a more semantic-aware approach to software development in their cooperation in the research projects. The short-term impact was rather strong as they used the method in their projects, but we know little of what effects this might have had on a longer term. During the projects, some of the participants showed great interest in using the method whereas some only saw it as another distraction from the real work. Those who found the method usable have hopefully been able to continue to use some of the ideas presented to them in their everyday work.

More details on the curricular and pedagogical impact of the thesis work can be found in [6] and [8].

### 7.3 Future Work

Future endeavors include further evaluations of methods for handling semantics since no definitive answers have been found so far. The lessons learned from previous evaluations will hopefully help in designing new evaluations that can provide statistically stronger results. Both industrial full-scale case studies as well as more formalized experiments in an academic setting will probably be needed to evaluate the impact of different methods for handling semantic aspects. What has been made clear by the evaluations presented in the thesis is that the human factor plays a very important role in determining the success of a project. Outside the scope of pure software development, it would be interesting to evaluate the human factor and to see which properties separate a competent software developer from a less competent one. In performing an evaluation of human factors, help from the human sciences field would be needed. These types of joint ventures with participants from different scientific fields do seem promising and are perhaps what is needed to find success-factors in software development. If it was possible to pinpoint the most important factors that separate a good developer from a not-so-good, it would have a great impact on how software development was taught and possibly how recruitment was done.
Bibliography


Appendix A

SEMLA

A.1 Introduction

This appendix describes Semla, a method for semantic specifications during the process of software design. Semla is the result of several years of cooperation between Ericsson companies in Karlstad (Infotech and TSAC) and Karlstad University. The method is based on accepted academic methods in combination with the experience gained in two case studies at Ericsson and an experimental course given twice at the University. After being tested in a closed course environment, the method has been reworked in cooperation with Ericsson Infotech AB so that it can be easily incorporated into present development methods at Ericsson and elsewhere.

The main message from this method is that the intentions of a software designer for a module or a function\(^1\) should be emphasized, both in the design itself and in the documentation. The intentions for a function and the limitations imposed on its use are described by a contract, which all clients should respect. The contract disconnects the usage of a module from its implementation. The contract highlights the necessary conditions for correct use of a function, as well as a description of the results produced. The contract will also serve as a reference during inspections and testing.

A.1.1 Applicability of the Method

Software engineering can include activities such as business engineering, requirements engineering, specifying system requirements, system and module design, and...
programming, integration, testing, deployment and maintenance. Many methods have been developed to cope with all or parts of these activities. Semla focuses on design and programming and highlights the semantic aspects of these activities. It can be easily integrated with other more complete methods like IDM, RUP or CMM but may equally well be used by itself in smaller projects.

During the past years, a number of techniques for producing high quality software have been developed. Some of these techniques require the use of certain tools or languages while others are generally applicable. Semla focuses more on an attitude toward software development and design than on specific tools or techniques and can thus be applied to all programming languages. It requires neither special support tools nor any special syntax. The examples in this appendix are written in Java, but a version is also available that has examples in the C programming language.

A.1.2 Appendix Structure

Section A.2 defines a few central terms used in this appendix. A short definition of additional terms can be found in Sect. A.7. Section A.3 contains some guidelines for how to design a module by contracts. The guidelines are subdivided into four parts: general guidelines for module design, guidelines for interface descriptions, guidelines for use and guidelines for implementation. The general guidelines are frequently found in other methods as well and are included for reference in Sect. A.8 of this appendix. Sect. A.4 gives guidelines for how to document the semantic information of a module. The documentation of a module is subdivided into three parts: an overview, a description of the interface and a description of the implementation. Maintenance and reuse of existing code put special requirements on the designer. These are described in Sect. A.5.

The descriptive parts of the method presented in Sections A.3 through A.4 are in general structured under the four units of Description and motivation, Rule, Procedure and Example. The rules are expressed as short imperative statements. They are repeated in a quick reference guide in Sect. A.6. A complete module implementing a list, where the rules and guidelines described in this appendix are applied, can be found in Sect. A.9. Fragments of this module are used as illustrations in the examples given throughout the appendix.

A.2 Terminology

Many of the terms used in this chapter are briefly explained in Sect. A.7. This section gives a more detailed description of some of the more central concepts.
A.2. TERMINOLOGY

A.2.1 Module

A module is a collection of methods and concepts belonging logically together. A module normally contains both variables and methods, even if it may miss one or the other in some cases. Examples of the latter include a method library, which does not have any data and a struct in C, which normally does not contain any methods.

![Figure A.1: A module has both exported and internal properties](image)

Figure A.1 above shows the principles for a module. The module implements an interface by having some accessible methods, defined in the interface, and some methods that cannot be accessed from the outside. It also contains data, which can be accessed only through the exported methods.

A.2.2 Interface

An interface describes the surface of contact between a supplier module and its clients, as shown in Fig. A.1. It describes the way in which a module will be conceived by other modules. While several modules can share a common interface, it does not need to be that way. There are several possible ways to describe an interface in Java:

- Create a Java interface and several classes that implement this interface, possibly in different ways. The interface of the module is then defined by the Java interface, which declares the exported methods of the corresponding classes.
- Create a class. The interface of the module is then defined by the methods declared public in the class.
- Create a base class, possibly abstract, and let some classes inherit from the base class. These classes may have different implementations for the same methods. The interface of the module then consists of the methods...
declared public in the base class. This is how the List example in this appendix is defined.

Different solutions may be chosen on different occasions, depending on the relationships required between the implementations and the interfaces.

A.2.3 Black Box

The black box concept implies that a module is seen as a closed black box that you can see only the outside of. This can be compared to a white box where the interest is in seeing the inside of the box, in our case the module implementation. Looking at a module as a black box means that it is interesting to look only at the interface, and that the module should be used without any knowledge of its implementation. If the implementation is changed, the use of the module should not be affected. Using a black box view of a module puts high requirements on the documentation, since the user of the module should not see the implementation details. In Java, a black box module is normally supplied as one or more class files containing the compiled Java code and a separate documentation, perhaps as Javadoc files.

A.2.4 Semantics and Semantic Integrity

The term semantics has a central position in the context of programming languages and programming techniques. Informally speaking, the semantics of a program is what the program means or produces, as opposed to its syntax, which describes what the program looks like. Semantic integrity means that the desired semantics of a module should be respected throughout the system. The biggest problem is perhaps not to maintain the semantics once they are defined but to define them in the first place. If semantic properties of a program are destroyed, the program can become unstable. This will eventually cause problems.

Semantic properties are normally expressed using assertions for a complete system (invariants) or for individual methods (pre- and postconditions). The designer of a module may have designed the module in such a way that certain semantic properties must be maintained. If these properties are violated, the semantic integrity of the module as a whole is invalidated. The semantic integrity of every individual module has to be respected since the semantics for the whole system depends on the semantics of the individual modules on which it is built.

Certain programming languages, like Eiffel, have built-in mechanisms to assure that the semantic properties of modules programmed in that language are respected. However, most programming languages, including Java, leave this completely to the programmer. In any case, many semantic properties
cannot be verified automatically, so a semantic consciousness on the part of the
programmer is always needed. Even in the cases where automatic verification is
possible, it is still up to the programmer to define and express the correct test
conditions.

\section*{A.3 Guidelines for Design}

Software quality can be achieved by different means, many of which are closely
related to mathematical formalism and not easily applied in industrial projects.
This chapter presents some pragmatic guidelines that are easy to apply in indus-
try. The guidelines are divided into four categories: \textit{general guidelines}, \textit{guide-
lines for interfaces}, \textit{external guidelines} and \textit{internal guidelines}. For each guide-
line, a short description and motivation is first given, which is followed by a rule
of thumb, a working procedure and finally an example of some program code
designed according to the guideline. The examples are taken from the \texttt{List}
module found in A.9.

\subsection*{A.3.1 General Guidelines}

The general guidelines are important for defining modules and for abstracting
data and methods. These guidelines can help during system design. They have
an important semantic impact and much of the literature on software engineering
and object oriented software construction mentions them. They are, however,
not the main focus for Semla. They can be found in A.8 of this appendix.

\subsection*{A.3.2 Guidelines for Interfaces}

An interface describes how a module presents its services to other modules. It
is normally built using exported methods. They operate on the data in the
module and may use other supporting internal methods. It is important to note
that an interface is normally an abstract construction and that the exported
methods implement the interface. More than one module may share the same
interface, which is another reason to keep the interface definition separate from
the modules. As an example, an interface may define some methods and dif-
f erent classes implement this interface in different ways for different hardware,
operating systems or versions. This section describes how an interface can be
designed. Guidelines for how to document the interface are given in Sect. A.4.

\textbf{Describe all methods}

\textit{Description and motivation:} Every method should be described in such a way
that the user knows the intended and legal uses of the method. As long as the
description is valid, the implementation of the method may be changed without the user being affected by the change. If the method name that reflects the purpose of the method is used, the description can normally be kept short. However, one should always provide a short explanation of what the method does.

**Rule:** Provide an overall description of the intentions and usage of each method.

**Procedure:** If the method is well defined and simple, there is normally no problem in describing it. Write one sentence to explain its functionality and possibly a few more sentences to describe the extra information needed to use it correctly. Full guidelines for the documentation of a method are found in Sect. A.4.

**Example 1: The description of a method**

```java
/***************************************************************************/
/* add: adds anElement to the beginning of the list */
/***************************************************************************/
public abstract List add(Object anElement);

/***************************************************************************/
/* remove: Removes the element at position 'position' from */
/* the list (counted from 0) */
/***************************************************************************/
public abstract List remove(int position);
```

**Define contracts**

*Description and motivation:* A contract is exactly what it sounds like: the definition of what a user and a supplier can expect from each other. Contracts play an important role in assuring that methods are used correctly. A contract implies that the user of a module is guaranteed a correct result if he supplies the module (or normally a method) with correct input. The pre- and postconditions, which constitute the contract, define what is meant by correct. The following subsections contain more details about preconditions and postconditions.

In addition to the pre- and postconditions, there must be an agreement between the module programmer and the module user that the contract should be respected from both sides. In situations where contracts are not used, a default contract is often assumed, saying “I don’t trust anybody and nobody should trust me”. This normally leads to redundant testing and extra code. Use and trust contracts in order to make the code shorter, simpler and easier to understand.

**Rule:** Define and express contracts for all methods.

**Procedure:** Determine the acceptable values for the input parameters to your
method. In addition, look for other conditions that the client code must guarantee, for instance that a communications channel is open. Express the precondition in plain English or through program code. Then, assume that all values are correct upon designing the method code. Express in the postcondition what the method has achieved. In some cases the postcondition simply restates the method description, but in many cases it will contain further details.

Example 2: Contract

```java
/**
 * add: adds anElement to the beginning of the list
 * @param anElement the data element to be added
 * @param Pre anElement != null
 * @param Post anElement is the first element in the list,
 * the list size is increased by one,
 * the modified list is returned
 */
public abstract List add(Object anElement);

/**
 * remove: Removes the element at position 'position' from
 * the list (counted from 0)
 * @param position the position of the element to be removed
 * @param Pre position >= 0 && position < size()
 * @param Post the element which was at position 'position'
 * is not in the list any more,
 * the list size is decreased by one,
 * the modified list is returned
 */
public abstract List remove(int position);
```

Specify preconditions for methods

Description and motivation: The precondition to a method is an expression for a condition that must be true before the method can be invoked. It may be directly measurable, for example that the value of one of the parameters to the method must have a value restricted to a certain interval. It may also be subtler. For instance, it may require that some other method be called prior to the method in question or that certain data elements must have certain values.

\[2\] We use the comment conventions for Javadoc. A Javadoc comment state that the comment should start with `/**` and end with `*/`. The Javadoc utility will extract the text in the comment to a documentation file. A single `*` in the beginning of the list is not included in the documentation. The keywords `@param` is recognized as a parameter description. The first word following the keyword `param` will stand out in the documentation. Since Javadoc does not specify pre- and postconditions, we use the parameter mechanism to document them.
Rule: Express the precondition for all methods.

Procedure: To find the precondition, identify the parameter values that can cause problems for your method or that violate the invariants. The precondition can be thought of as a virtual filter. It lets the correct values pass through and filters the illegal values. The module designer should ask himself the following questions to find the relevant precondition. When and how may the method be called? Are there any values that will cause problems in the method? A method should always have a precondition. In some cases, it will simply be the condition true, meaning that the method may always be called. Examples of methods having this precondition include most so called predicates. Predicates are methods that do not affect the object in any way. They only return information about the state of an object.

Example 3: Precondition
The method getElement from the class List shows a simple example of a precondition. The method returns the data element at the given position in the list. It is not possible to return data from a position outside the current list, so the user must assure that the position is within the list limits before calling the method. The precondition for getElement can be expressed as

// Pre: position is from 0 through the number of elements in the list minus one

The precondition can also be formalized using program code, in order to avoid misunderstandings. In some cases, the text volume may also be reduced.

// Pre: 0 <= position && position < size()

Specify postconditions for methods

Description and motivation: A postcondition describes the result of invoking a method or the effect produced by the method. In the simplest cases, a variable in the object executing the method may have changed or a value may have been computed and returned. In more complicated situations, several variables or even the state of the whole system may have changed.

Rule: Express the postcondition for all methods.

Procedure: Determine the result produced by the method. This is normally the postcondition of the method. The postcondition should not be related to specific code but be kept at a more abstract level. The most important aspect is that the postcondition expresses what the method has achieved and not how it is done. Here are some questions to ask to help identify the postcondition. What has the method done? In what way is the state of the object different after a call as compared to before? Has any variable changed its value? Which value did the method return? Where convenient, the pseudo-variable result may be used to express the result returned from a method.
A.3. GUIDELINES FOR DESIGN

Example 4: Postcondition after a change in an object
The postcondition for add in Example 2 was:

```java
// Post: anElement is the first element in the list,
// the list size is increased by one
// the modified list is returned
```

The postcondition can also be more formalized in order to avoid misunderstandings.

```java
// Post: anElement == getElement(0)
// size() == old size() + 1
// result == the modified list
```

The term old should be interpreted as the value, at the start of the execution of the method.

Example 5: Postcondition for accessing the state of an object
It may be useful to define separate predicate methods, for instance isPositionOk, for the kind of preconditions that are needed by remove in Example 2 above. We will call this a guard method. Such a method improves the understanding of the program. It also makes it easier to avoid hard to spot errors produced by negligence, for instance that a programmer writes <= instead of < in a test.

```java
// Post: a boolean value telling if position is legal
// for accessing a list is returned.
// true= position ok, false=position not ok
public abstract boolean isPositionOk(int position);
```

This postcondition may seem trivial at first, but gives enough information to make it unnecessary to see the code of the method. Thanks to the postcondition, it is not necessary for the user to know the exact details needed to access the list.

Identify exported invariants

Description and motivation: An exported invariant expresses relations and conditions that a module guarantees and that the user can always trust. Since the exported invariants express requirements that the methods of the module must fulfill, they constitute an important part of the definition of the module interface. The exported invariants should not depend on any particular implementation but rather describe the concept that the module is implementing.

Rule: Identify and document any exported invariants for the module.

Procedure: Make notes of any conditions or properties for the modules that are known to the user of the module and that are always true. They are the exported invariants. Often, they seem trivial and are therefore overlooked. Sometimes they may be expressed through one of the methods of the module, sometimes through several methods and sometimes in plain English. The invariants should
be expressed using only exported properties and should not depend on any particular implementation.

Example 6: Exported invariants, informally

// Exported invariants:
// the size of a list can never be negative
// a list is empty if its size is zero
// positions are counted from zero through list size minus one
// no position in a list contains null data

Example 7: Exported invariants, formalized

// Exported invariants:
// size() >= 0
// isEmpty() == (size() == 0)
// positions are counted from zero through size() - 1
// for all legal values of position:
// getElement(position) != null

A.3.3 External Guidelines

This section contains guidelines on how to use modules as black boxes. The guidelines should be used when several modules are combined with each other or an existing module is used by one that is under development.

Satisfy the preconditions

Description and motivation: A precondition expresses what is required to use a method in the way intended. It is therefore necessary to make sure that the precondition is satisfied before the method is called.

Rule: Satisfy the precondition for all methods called.

Procedure: Every time a method is called, the user must make sure that the preconditions for that method are satisfied. Either the program flow should be such that it guarantees the precondition or there should be a test for the precondition immediately before the method invocation.

Example 8: Implicitly satisfied precondition

The precondition for the method `getElement` is that the input parameter `position` must have a legal value. If the programmer knows this to be true, for instance because there is only one possible flow of control to the call, the precondition does not need to be tested explicitly. ...

```java
myList = myList.add(data);
// the list has at least one element at position 0 now
// and it is therefore safe to call getElement for position 0
```
A.3. GUIDELINES FOR DESIGN

anElement = myList.getElement(0);
...

The above code is correct.

Example 9: Precondition satisfied through testing
If the program has more than one possible flow of control to `getElement` and not all of them guarantee the precondition, then the user must perform a test to assure that the position is legal.

    if (x == y)
        myList = myList.add(anElement);
    else
        doSomethingElse();
    if (myList.isPositionOk(pos)) // test
        anElement = myList.getElement(pos); // method invocation
    else {
        System.out.println
           ("" + pos + " out of bounds (0-" + myList.size() + ")");
    } ...

The program above either adds an element or does something else, depending on the values of \(x\) and \(y\). The validity of the call to `getElement` must therefore be tested explicitly.

In many cases, the program flow and input values are given by an end user, and end users may never be trusted. In all cases where an end user is involved, the preconditions must be tested.

Trust the postconditions

Description and motivation: To trust a postcondition simply implies that the user can assume that the method has done what it has promised through the postcondition. There is no need to test what is already guaranteed to be true according to the postcondition. This of course assumes that the user has satisfied the precondition when the call was made. If the precondition was not satisfied, nothing can be said about the situation when the method returns (if it returns at all).

Rule: If the precondition was satisfied, then trust the postcondition.

Procedure: Make sure that the conditions contained in the precondition are satisfied, either implicitly through the program flow or explicitly through a test. Then, trust that the method does what it has promised through the postcondition and do not test for conditions that are already stated to be true.

Example 10: Trust the postcondition
This example shows how the precondition is tested before the call to `remove`,
but the postcondition is not tested after the return from the method.

```java
List myList = new List();
...
myList = myList.add(aValue);
...
if (myList.isPositionOk(pos)) /*checking pre for remove */
{
    myList = myList.remove(pos);
    // no need to check outcome,
    // success guaranteed by the postcondition
    System.out.println(''Element at pos ''+pos+'' was removed'');
}
```

**Use methods correctly**

*Description and motivation:* The user of a module should only use the methods of the module in the intended way and not utilize any side effects, which may exist. If the module is implemented in a different way in the future, side effects that are exploited can disappear. The client software would then be based on effects that no longer exist. Correct use of methods is normally not a large problem, but it is still worth mentioning.

**Rule:** Only use a method according to the documentation.

**Procedure:** Normally, using methods correctly does not present problems. It is still worth mentioning that one should be aware of the issue.

**Example 11: Use methods correctly**

Assume that one wants a certain delay in a program. Assume further that experience has shown the addition of a certain number of elements to the list and the subsequent removal of the same elements to consume the necessary amount of execution time.

```java
...
{
    List delayList = new LinkedList().add(new Object());
    for (i = 0; i < delay; i++) // consume some time:
        delayList = delayList.add(delayList.size()-1);

    delayList = delayList.erase();
}
```

The List implementation may be replaced by a more efficient one in a future release, or the program may be run on a faster machine. It would therefore be a
A.3. GUIDELINES FOR DESIGN

A.3.4 Internal Guidelines

The internal guidelines contain criteria for the internal design and the implementation of a module. Since the module is restricted by the interface, it is important to respect all the conditions defined by the interface definition. It is also important to describe data in a clear way.

Respect the contracts

Description and motivation: To respect the contracts implies to trust them and not to test for conditions that are already true according to the contract. If both the programmer and the user of a module respect the contracts, then the software will be clean without redundant tests and similar anomalies. The output from the methods must follow the conditions described in the postconditions. The programmer must also trust that the preconditions protect the module from being used in an incorrect way and not perform redundant tests on the input parameters. The tests do not only take time to write, they consume space in the finished code, take time to execute and are useless. They also normally complicate the code and are, as all other code, extra sources of errors. Thus, no tests should be performed for conditions already fulfilled through the preconditions.

Rule: When implementing a module, respect the contracts defined.

Procedure: Do not test conditions that are already true according to the precondition. A quite common mistake is to not trust the precondition ("But what if they call it in the wrong way...") and to test it in the method. This is unnecessary and only takes a lot of time. Also, make sure that the postcondition and invariants are true when the method completes.

Example 12: Respect the contracts in the implementation

```java
/*******************
* @param Pre isPositionOk(position)
* @param Post the element, which was at position 'position',
* is not in the list any more
* the list size is decreased by one
***************************/
public List remove(int position)
```

Some people prefer the form `isPositionOk(position) == true`, which means the same thing. The expression `isPositionOk(position)` is a Boolean expression and if the expression is true, the assertion is true as well.
APPENDIX A. SEMLA

{  
  LinkedNode toRemove; // the element to be removed  
  if(position == 0) // first element should be removed  
  {  
    toRemove = first;  
    first = toRemove.getNext(); // remove from the list  
  }  
  else  
  {  
    // no test on first or position needed,  
    // thanks to the precondition  
    
    LinkedNode previous = first; // temporary  
    for(int i = 0; i < position - 1; i++)  
    {  
      previous = previous.getNext();  
    }  
    toRemove = previous.getNext();  
    // pre guarantees this to be OK  
    previous.setNext(toRemove.getNext());  
    // remove from the list  
  }  
  return adjustSize(-1); // maintain internal invariant  
} // remove

No test is needed inside the method to verify that position has a legal value. That is already settled in the precondition, which should be satisfied by the caller. The only testing needed in the method is for the special case of removing the first element in the list.

Describe the data

Description and motivation: All important aspects of the variables used to implement the module should be described in order to avoid any doubt about how they should be used.

Rule: Describe all the variables used to implement the module in order to avoid any doubt about their intended use.

Procedure: Determine and document what each variable represents and for what purpose and how it should be used. Exceptions may be made if this is sufficiently clear from the name of the variable or if the variable is a temporary variable, for instance a loop counter.
Example 13: Data descriptions

```java
public LinkedList inherits List
{
  private LinkedNode first; // the first node in the list
  private int size; // the total number of nodes
  ...
}
...
void someFunction(...)
{
  int i; // loop variable
  int top; // index of the topmost element in array
  for (i = 0; i < MAX; i++)
    doSomething(i, top);
}
```

In the above example, the usage of each variable is described. One variable has been tagged "loop variable" to explain why it exists.

The last example shows a common source of error. Does top refer to the last element used or to the next position to be used? Does top represent the index of the topmost element of the array or is it the number of array elements used? Attempts are sometimes made to express different semantic meanings with one single variable. After a while, the programmer is no longer sure about how the variable should be used. It is important to document the intended use of a variable well, so that errors can be avoided.

Identify internal invariants

*Description and motivation:* An invariant is something that does not change. It is an expression that is constantly true. In programming, invariants are used to describe conditions that do not change. Such descriptions make additions or changes easier to perform by clarifying how and when a variable may be used. Every method can assume that the invariant is valid upon entry to the method and must ensure that it is valid at the exit from the method. In some cases, a method will break the invariant temporarily during its execution, to perform some test or to update some object, but it must make sure that it is reestablished before exiting the method.

For local variables defined and used in smaller, easily understood methods, it may not always be necessary to describe any invariants. For loop variables in somewhat more complicated methods, it is often useful to explicitly state an invariant to avoid "off-by-one" errors. After a while, it may be difficult to remember whether a certain variable represents the first free or the last used position or if it points before, on or after something. It is easy to use the variable differently at different places in the program.
Rule: Express all limitations for a variable in the form of invariants.

Procedure: Consider the restrictions that apply for a variable or for a group of variables. Write these down at the same time as the variable or variables are declared.

Example 14: Identify internal invariants
Assume that you want to add, to an implementation of a linked list, a variable maintaining a count of the number of elements in the list. The variable is increased by one every time an element is added to the list and decreased by one when an element is removed. This has the advantage that the whole list does not need to be counted to get its size. This mechanism is not visible from the outside, so it is a purely internal optimization.

```java
// LinkedList.java
public class LinkedList
{
  private LinkedNode first;
  private int size; // The number of elements currently in
                  // the list, 0 <= size < Integer.MAX_VALUE
}
```

The invariants for the variable `size` are:
- `size` is equal to the number of elements in the list
- `size` is non-negative and smaller than `Integer.MAX_VALUE`

These invariants should be made explicit close to the corresponding variable declaration. The statements above may seem evident, given the name of the variable. However, in many similar cases, the use of a variable is not evident. Is it for instance the number of elements contained in an array, the position of the last element added to the array or the position of the next element to add?

Establish and maintain invariants

Description and motivation: In order for an invariant to hold, it first needs to be established. That should be done as soon as the object is created, by the constructor. Then, every method that modifies a variable must assure that the invariants still hold when the method exits.

Rule: Make sure that the constructors establish the invariants and that all other methods maintain them.

Procedure: The constructors are useful for establishing the invariants. In fact, that is a constructor's most important task, and it should normally be restricted to doing just that. Identify all methods that modify the data and verify that the invariants hold before they return.
Example 15: Establish and maintain invariants
The constructor must establish the invariants. Initializing the list to be empty satisfies the condition that the list size should be non-negative. The variable size should be equal to the number of elements in the list. All methods modifying the data must maintain these invariants.

// LinkedList.java
public class LinkedList extends List
{
  public LinkedList() // constructor, initialize an empty list
  {
    // establish invariants:
    first = null; // exported invariant (empty list)
    size = 0; // internal invariant (size=list size)
  }
  public List add(Object anElement)
  {
    // ... code to add an element ...
    return adjustSize(1); // maintain internal invariant
  }
  // the precondition assures that the list is not empty
  public List remove(int position)
  {
    // ... code to remove an element ...
    return adjustSize(-1); // maintain internal invariant
  }
} // class LinkedList

A.4 Guidelines for Documentation

This chapter contains guidelines for how to document a module in order to encourage correct use of the module. The focus is on how to document the semantic aspects.

Not all the important system aspects that need to be documented are covered. Aspects not covered include the system structure, the requirements on both development and execution environments, the routines for linking and building the system, and history records. For these and other aspects, the guidelines for your company should be followed. The same is true as regards where to write the documentation, whether it should be embedded in the source code or whether it should be in a separate document. We only note that the documentation should be easily synchronized with the design and implementation during both development and maintenance. The documentation should also be readily available to both users and programmers when needed.
A.4.1 The Purpose and the Parts of the Documentation

The purpose of documenting is to transmit information to some target group. A software module has two target groups, its users and its programmers, and these two groups have different information needs. These needs and differences should be reflected in the documentation. That is achieved by separating the documentation into three parts, an overview, a description of the interface and a description of the implementation. The overview describes the objectives for the module and the general rules for its use. It is primarily of interest to the users but is also useful to the programmers. The interface part describes the data types and the exported methods defined by the module. It is of primary interest to both of the target groups, since it should be used by the users and implemented by the programmers. It is the most central part, so it should have a predominant position in the documentation and needs to be described very carefully. The implementation part describes the details of the implementation. It is the area of the programmers. It is produced by the designer and used during implementation and maintenance. This part is not intended for the user, who could otherwise take advantage of it in his client software. That, in turn, would break the encapsulation and jeopardize the maintainability of the system.

The primary target group for the documentation is normally the user, so his needs should be satisfied first. This implies that, if a separate documentation is made for the intention of the user, it should contain the overview and the interface parts but not the implementation part. One should always think in terms of the needs of the users when documenting the overview and the module interface.

The rest of this chapter describes what the three different parts of the documentation should contain in order to satisfy the need for semantic information.

A.4.2 Documentation of the Overview

The purpose of the overview is to give a general understanding of the module. It should give a general description of the purpose of the module, the data structures used by the module and the rules that govern the use of the module. This part of the documentation is primarily for the user but is also useful for the programmer.

The purpose of the module

*Description and motivation:* Both users and programmers need a quick introduction to the issues the module is addressing. This includes the purpose of the module and possibly a description of the contexts where it can be used.

*Rule:* Give a short description of the purpose of the module.
A.4. GUIDELINES FOR DOCUMENTATION

Procedure: Mention the most important methods and services of the module. Be generous with illustrations to make the description easier to understand.

Example 16: The purpose of the module
The overview of the list module may be:

The List module implements dynamically sized lists and operations to manipulate them. The content of a list can be of any object type and is supplied to and from the module through an Object reference.

General data description

Description and motivation: In order to use data correctly they must be understood. If the module defines new data types, then they should be presented in such a way that the user can understand how to use them. The overview part of the documentation should contain a general description of the data. The details should be described in the interface part, as explained in Sect. 4.3 below. Neither this overview nor the interface description should give any clue about how the data types are implemented. That would jeopardize the semantic integrity of the module, since it would invite the user to manipulate data directly or to take advantage of various side effects.

Rule: Present the data types defined by the module and their general properties. Include any exported variables defined by the module.

Procedure: Put yourself in the situation of a user. What does he need to know about the data types defined by the module in order to get a first impression of how to use them? Those are the properties that should be described in the overview. The description should show the general properties of the data types without revealing their implementation.

Example 17: General data description
A general description of the data type List may be as follows.

The list module defines one data type, List, with the following general characteristics:

- A list may contain data of any object type.
- Creating and managing the contained data objects are the responsibility of the client.
- The maximum size of a list is only limited by the amount of working memory available.
General usage description

Description and motivation: The methods of a module often require that a certain usage pattern is followed. Typically, something needs to be created, initialized or opened before it can be used and in some way terminated or closed after use. Sometimes the use must also follow some sequential, circular or other usage pattern. These dependencies define the underlying semantic conditions for use and need to be described.

Rule: Present the main functional groups defined by the module and their usage patterns.

Procedure: Identify the different main stages or states that an object from the module goes through. Look for and document any pattern that must be followed when using the module. The description should be kept on an overall level. Focus on the conditions for use. No information about the implementation should be revealed.

Example 18: General usage description
The use of the methods in the List module may be described in the following way:

The module contains three categories of methods: to create, manage and erase lists.

- Creation is done through the constructor, which also initializes the data structure.
- The managing methods are those used to add, remove and get list elements and to retrieve other list information.
- When the list is no longer needed, it should be erased.

A.4.3 Documentation of the Interface

While the purpose of the overview is to give a short, general introduction to the module and its use, the interface part should give all the details a user needs to use the module correctly. It should contain a clear and complete description of how to use every method and data type exported from the module. This part of the documentation is the main reference for both developing clients for the module and for implementing it. This section addresses the documentation of the interface. The design and definition of the interface are described in Sect. A.3.2.
A.4. GUIDELINES FOR DOCUMENTATION

Defined data types and their external invariants

Description and motivation: Any details needed to use the declared data types correctly must be supplied. Sometimes, some invariants can be found, which the user can always rely on. They should be documented here. These invariants are often expressed in plain text or by some of the methods defined by the module.

Rule: Describe how the data defined by the module should be used, together with any external invariants.

Procedure: Every time you think "this is something that the module user must know about the data types defined by this module", that information should be written down. Describe how to declare variables of the data types defined by the module. If there are any invariants for the declared data types, they should also be written down. Often, these properties become evident during the work on the problem, and this section is a good place to collect them. They are often expressed through some of the exported methods defined by the module.

Example 19: The data type List and its invariants
This example shows a possible documentation of the data type List and its invariants.

Description
The data type represents a list.
A List variable is a pointer to an instance of the class List.

Declaration example
List myList;

General properties
A list is created and initialized through a constructor and assigned to a List variable. All List methods are applied to a valid List variable.
When a list is no longer needed, it should be erased.
A data element in a list is identified through its position.
The position numbers start at zero.
Position zero is said to be the front of the list.
All positions in the list have real data elements.
No data element in the list is null.

Invariants (in plain text)

- The number of elements in a list, i.e. the size of the list, cannot be negative.
- If the list size is zero, then the list is said to be empty.
• The numbering of the positions in the list go from zero through the number of elements in the list minus one.

Invariants (more formalized)
For all valid lists:
• size() >= 0
• isEmpty() == (size() == 0)
• positions are counted from zero through size() - 1

Description of every method

Description and motivation: Each individual method exported from the module needs to be described in such a way that it can be used correctly. The semantics for each method is defined through the contract as given by its pre- and postcondition.

Rule: Describe every exported method completely.

Procedure: The example below can be used as a template if the module should be documented in a separate document. Start with a simple presentation of the methods followed by the syntactic and semantic details and any non-functional specifications. The syntax specification is the same as in the Java source file. Sect. A.3.2 describes how to define the contracts for the semantic description. If the method returns a value, it is described in the postcondition.

Example 20: Method description
The method to add an element to a list could be documented as follows.

Add an element
Description
Adds a new data element at the first position in the list and returns the new list.
All data already in the list are moved up by one position.

Syntax public List add(Object anElement)

Parameters
anElement the data element to be added to the list

Precondition
• anElement is not null
Postcondition

- anElement is the first data element in the list
- the list size is increased by one
- all data elements previously found in the list are moved up by one position
- the modified list is returned

A.4.4 Documentation of the Implementation

The description of the implementation is for the sole purpose of the programmers of the module. It should be kept separate from the documentation given to the user of the module. That will prevent the user from taking advantage of any known implementation details in his client programs. This documentation corresponds to the information defined in Sect. A.3.4. It should contain sufficient information to allow a correct implementation, to verify that the implementation is correct in relation to the defined interface and to explain the choices made during the implementation. The implemented objects and the associated invariants should be documented, along with any special considerations affecting the choice of implementation and an explanation of local variables used in the methods of the module.

Implemented data structures and invariants

*Description and motivation:* Every implementation will be supported by concrete data structures and algorithms. The exported properties of the module have already been described through the description of the interface. The purpose of the description of the implementation itself is to supply internal details to support the implementation and the maintenance of the module. This is where all the secrets are revealed.

An implementation normally adds additional constraints and dependencies to those described in the interface. Internal invariants are used to describe these new constraints and dependencies. These internal invariants depend only on the actual implementation chosen and should not be disclosed to the user of the module. The invariant part is a good place to collect "information to remember" about the variables used for the implementation.

*Rule:* Describe the data structure used in the implementation. Include constraints, dependencies and invariants.

*Procedure:* Describe all parts of the data structures in the module, how they should be used and any relationship between them. Look for constraints and dependencies on the variables and document them. Such dependencies typically appear in composed data structures or when redundancies are introduced, often
for optimization purposes, such as storing the size of a linked list in a separate variable to save counting the elements when the size is requested.

**Example 21: Data structures and invariants**
The description of a linked list may be as follows, perhaps using comments in the source code for the documentation. In this example, the class `LinkedList` inherits from and implements the methods defined in the abstract class `List`.

```java
// File LinkedList.java
public class LinkedList extends List {
    ...
    // Private data structure
    //
    // A list is represented by a chain of linked nodes
    // The data contents is held by Object pointers
    // in the nodes
    //
    // Internal invariant for LinkedList:
    // size == the number of nodes in the list
    // first points to the first element in the list (if any)
    // first == null iff size == 0
    // nodes are allocated locally by the list methods
    // data objects are allocated by the client
    private ListNode first;
    private int size;
} // class LinkedList

// File ListNode.java
class ListNode {
    // A linked node holds sequence information and data for a
    // linked list.
    // This class is defined at the package level and will be
    // invisible outside of the package where it is defined.
    // This node defines a single linked list.
    //
    // Invariant for ListNode, must be maintained by the list
    // next == null iff there is no following node
    {
        private ListNode next;
        private Object data;
    } // class ListNode
```
Special considerations for the implementation

*Description and motivation:* There may be other requirements and conditions for use not described in the previous sections. They may be described under this header.

*Rule:* Describe non-functional considerations, requirements and priorities.

*Procedure:* Consider whether the module is intended for a specialized usage or whether there are other particular conditions for use.

*Example 22: Special considerations*
The following special considerations may be documented to motivate the redundant `size` variable in the linked list:

```plaintext
/************************************************************
* It is assumed that the size operation will be used
* frequently for the list, since it appears in several
* preconditions.
* This is the reason for the redundant size variable,
* which must be maintained at every addition and removal.
 ************************************************************/
```

Explanations of local variables

*Description and motivation:* The implementation of a method often requires additional local variables. These too need to be explained to support those who will be maintaining the program. A good explanation can avoid semantic problems in the future. Some cases may also benefit from an invariant being defined for local variables.

*Rule:* Describe the usage and possible invariants for local variables in the methods of the module.

*Procedure:* Put yourself in the place of a programmer assigned to maintain the code after a few months or even a year. Explain the purpose and limitations of each local variable. If you need to take special considerations into account, it often indicates that an invariant should be defined. Describe such invariants, so that the variable can continue to be used in accordance with its intentions.

*Example 23: Local variables*
The method `remove()` introduces three local variables. The intended use is documented in comments. There is also an invariant for the relation between the variable called `previous` and the position counter `i` used while searching for the element that should be removed. This invariant is established as soon as the variables are declared. It makes it possible to reason about the correctness of the code during and after the search for the element that should be removed.

```plaintext
LinkedNode toRemove = null; // the element to be removed
```
A.5 Guidelines for Maintenance and Reuse

Programming is not only about creating new code. It is also about maintaining and continuing to develop existing code, including reusing such code in new contexts. Existing code may, or may not, have the qualities described in Sect. A.3.

This chapter describes semantic concerns related to maintenance and reuse. We begin, in Sect. A.5.1, by explaining what we mean by maintenance and reuse. Sect. A.5.2 gives guidelines for how to maintain the semantic integrity of a module and of individual methods during maintenance. Finally, Sect. A.5.3 considers reuse of methods.

A.5.1 Maintenance vs. Reuse

In this appendix, we use the term maintenance in a broad sense. It covers all activities that lead to a change in the interface or in the implementation of an existing module. The term includes activities that are traditionally called maintenance, such as error corrections, minor improvements and adjustments for new or changed environments. In addition, it includes the development of new and improved functionality.

We use the term reuse to mean that an existing module or method is used unchanged in a new context, possibly in a context that the original developer did not have in mind. A module released as a standard library is an example of code that is being reused.

A.5.2 Maintenance of a Module

During maintenance of a module, some methods are changed and some added. Before changing an existing method, both the method itself and the invariants for the module containing the method must be understood. If the module and the method are not well documented, with invariants and contracts, that should first be done. This is described in Sect. A.5.3 below. Throughout the rest of this section, we will assume that the module, or at least the part of it affected by the change, is well documented.

The main question to ask when modifying an existing method is whether it can continue to be used in the same way after the change. How can one be...
A.5. GUIDELINES FOR MAINTENANCE AND REUSE

sure that there are no harmful effects on the client code? The answer lies in comparing the pre- and postconditions for the original and the changed versions of the method. If neither the precondition nor the postcondition is changed, then the methods are of course interchangeable. If the precondition and/or the postcondition have changed, then the following rule summarizes the condition for using the changed method in place of the old method:

*The new method has a weaker precondition and a stronger postcondition than the old one.*

We can call this rule *the golden rule of substitution*. Figure A.2 illustrates how the preconditions and postconditions are interpreted when a method is modified, from the point of view of both the supplier and the client. Version A denotes the method before the change and version B after the change. The client is unaffected by the change if the client can continue to act as though it were calling version A of the function. The pre- and postconditions of version B must then live up to this illusion.

![Figure A.2: The interpretation of pre- and postconditions](image)

The rest of this section will develop the consequences of the golden rule of substitution and the interpretation of Fig. A.2.

**Changes to weaker preconditions**

*Description and motivation:* The client in Fig. A.3 satisfies A’s precondition. If it then automatically satisfies B’s precondition as well, then the precondition for B is said to be weaker than the one for A. In that case, version B of the method can replace version A.

*Rule:* Make sure that the new precondition is weaker than (or equal to) the old one when changing an existing method.

*Procedure:* Identify the precondition required for version B. Assume that the precondition for version A holds and see whether the precondition for version B
is also satisfied. (If you fancy formalism, you should show that \( \text{pre}_A \Rightarrow \text{pre}_B \).)

**Example 24: Weaker precondition**

The precondition for `add` is that `anElement` should not be `null`. We wish to modify the module to allow null data in the list, so we define a new version of the `add` method with the new precondition `true`. Can this new version be used in place of the old one?

Assume that the client satisfies A’s precondition. Then, the precondition for B is also satisfied. In fact, B’s precondition is the expression `true`, so it is always satisfied. Therefore, B’s precondition is weaker than A’s. The modified version of the method may then be used in place of the old one. This is illustrated in Fig. A.3. The comparison between the preconditions for versions A and B may be put another way. The situations in which `anElement != null` are only some of the situations that satisfy B. The other ones are those in which `anElement == null`.

**Changes to stronger postconditions**

**Description and motivation:** The client in Fig. A.4 is expecting A’s postcondition. If it is automatically satisfied through B’s postcondition, then the postcondition for B is said to be stronger than the one for A. In that case, version B of the method can replace version A.

**Rule:** Make sure that the new postcondition is stronger than (or equal to) the old one when changing an existing method.

**Procedure:** Identify the postcondition guaranteed by version B and assume that it holds. Examine whether this will make the postcondition for version A hold as well. (If you fancy formalism, you should show that \( \text{post}_B \Rightarrow \text{post}_A \).)

**Example 25: Stronger postcondition**
A.5. GUIDELINES FOR MAINTENANCE AND REUSE

Assume that our data elements can be part of several lists at the same time and that we want to keep track of the lists in which they are included. Therefore, we want to extend the functionality of the add method to update that information as well. This situation is illustrated in Figure A.4. In this case, the

postcondition for B - that \textit{anElement} is included in both the list and in some global inventory - implies that \textit{anElement} is at least included in the list. The postcondition for A is therefore satisfied by version B, so the changed version may be used instead of the original one.

Other changes in the contract for an existing method

\textit{Description and motivation:} Not all changes follow the golden rule of substitution. If the contract of a method is changed in a way that does not follow this rule, then the clients of this method may be the victims of changed results when invoking the method. If the clients want to be sure that they get the correct result from the method, they need to be modified and adapted to the new contract.

Sometimes it is not possible to modify all the clients at the same time as the change is made. In that case, a new method has to be defined in the module according to the new contract. The existing method cannot be changed. The new and the old methods will then live side by side and each individual client can be adapted to the new contract when and if it is suitable.

\textit{Rule:} Define a new method when the golden rule of substitution does not hold.

\textit{Procedure:} Compare the pre- and postconditions for the original and the new contracts. If either of the rules "weaker precondition" and "stronger postcondition" is violated, then the modified method needs a different name than the old one. It cannot simply replace it. The only exception is if all the clients can be adapted to the new definition at the same time as the change is made.
Showing that the golden rule of substitution is broken is normally much easier than showing that it holds. In fact, to show that it is broken, it is sufficient to find one situation in which it does not hold. Look for a case where the precondition holds for A but not for B or where the postcondition holds for B but not for A. If you find one, then version B of the method cannot replace version A without affecting the client routines.

Example 26: Changing a contract
Assume that we want to add a general insert method, which will add a new data element anywhere in the list, not only at the beginning as add does. We already have a guard method isPositionOk for referencing the list. Can we redefine it so that it can be used for insertions as well?

To answer that question, we need to identify the pre- and postconditions for version B of isPositionOk and see whether version B satisfies the golden rule of substitution. The position parameter for a general method insert(anElement, position) should be allowed to go from zero to one position behind the last element. Therefore, the guard method isPositionOk\(_B\) (position) should have the postcondition

post\(_B\):
the value of \((0 \leq \text{position} \&\& \text{position} \leq \text{size()}\)
is returned

As a reminder, the postcondition for isPositionOk\(_A\) is post\(_A\):
the value of \((0 \leq \text{position} \&\& \text{position} \leq \text{size()} - 1)\)
is returned

We need to verify that post\(_B\) \(\Rightarrow\) post\(_A\). However, we can find one value for position, namely size(), which satisfies post\(_B\) but not post\(_A\). Therefore, the postcondition for version B is not stronger than (or equal to) the postcondition for version A. The method isPositionOk can therefore not be used, even in a modified form, for our new purpose. We need to define a new guard method for the new purpose.

Addition of new functionality

Description and motivation: A module can develop through the addition of new methods. These new methods may build on the existing data structures or add new data types with associated methods. This kind of change is really an addition rather than a change. The new method or methods should be developed according to the guidelines for development of new software in Sect. A.3. However, during that work, it is imperative that the new methods respect any existing invariants.

Rule: Follow the guidelines in Sect. A.3 when adding new methods or new data types to a module and be careful not to violate any existing invariants.
A.5. GUIDELINES FOR MAINTENANCE AND REUSE

Procedure: Find all existing data types and data structures affected by the changes and check the invariants that govern their use. Make sure that all these invariants are respected by your additions.

Example 27: Addition of new functionality
A general method \texttt{insert(anElement, position)} to insert an element at a freely chosen position in the list may be added to the \texttt{List} module at any time. A guard method for the associated precondition, for instance \texttt{isInsertPositionOk}, may also be added. These additions will not affect anything else in the module as long as they are developed according to the guidelines for module design. The existing invariants, for example those that concern the list size and the position of the data elements, should also be respected.

A.5.3 Reuse of a Method

Existing methods can be defined with or without the use of contracts, and the procedure for reusing them will vary accordingly.

Reuse without contracts

Description and motivation: If a method or a module is defined without contracts, then the first step before using it is to try to understand how it is supposed to be used. It often pays off to go through the part of the code needed to to document the related pre- and postconditions for every method. This makes both the immediate use and future maintenance easier. The probability that the methods will be used again is probably high, since they obviously already have proven to be reusable.

Rule: Before writing client software for a method or a module with no documented contracts, identify and document their contracts.

Procedure: Below is a list of some steps to go through before reusing a method.

- Read the description of the method if there is one.
- If there is no description, try to figure out what it does from the name or the context of the method and make your own description.
- Identify the constraints for using the method, either by reading the description or by studying the code.
- Add pre- and postconditions for the method.
- Try to detect and document any invariants for the module according to Sect. A.3.2, final paragraph.
A more thorough description of the procedure is given in the Sections A.3.2 and A.4.3. Once the contract is documented, proceed according to the next section.

**Reuse with contracts**

*Description and motivation:* If the method or module for which you want to develop client software is documented with contracts, then you have an easy task reusing them. Call the method according to its intentions and make sure that the precondition is satisfied before the call.

*Rule:* If a method or module to be reused is documented with contracts, then follow the contracts when developing client software for it.

*Procedure:* More information about this issue is found in Sect. A.3.2.

**A.6 Quick Reference Guide**

This section contains a list of all the rules from this appendix, arranged chapter for chapter. Use it as a quick reference while working. Check the relevant section for further details.

**A.6.1 Quick Reference for Module Design**

**General guidelines**

- Split the program into modules, which should be well defined, complete and coherent.
- Let a module communicate with as few other modules as possible and send as little information as possible between modules.
- Locate variables and the methods that operate on these variables in the same class.
- Hide data variables and methods that can be misused.
- Define and abstract the data to a level above the implementation level. Let each method in a module perform one well defined, limited task.

**Guidelines for interfaces**

- Provide an overall description of the intentions and usage of each method.
- Define and express contracts for all methods.
- Express the precondition for all methods.
• Express the postcondition for all methods.

• Identify and document any exported invariants for the module.

**External guidelines**

• Satisfy the precondition for all methods called.

• If the precondition was satisfied, then trust the postcondition.

• Only use a method according to the documentation.

**Internal guidelines**

• When implementing a module, respect the contracts defined.

• Describe all the variables used to implement the module in order to avoid any doubt about their intended use.

• Express all limitations for a variable in the form of invariants.

• Make sure that the constructors establish the invariants and that all other methods maintain them.

### A.6.2 Quick Reference for Documentation

**Overview**

• Give a short description of the purpose of the module.

• Present the data types defined by the module and their general properties.

• Include any exported variables defined by the module.

• Present the main functional groups defined by the module and their usage patterns.

**Interface**

• Describe how the data defined by the module should be used, together with any external invariants.

• Describe every exported method completely.
Implementation

- Describe the data structure used in the implementation. Include constraints, dependencies and invariants.

- Describe non-functional considerations, requirements and priorities.

- Describe the usage and possible invariants for local variables in the methods of the module.

A.6.3 Quick Reference for Maintenance and Reuse

Maintenance of a module

- Make sure that the new precondition is weaker than (or equal to) the old one when changing an existing method.

- Make sure that the new postcondition is stronger than (or equal to) the old one when changing an existing method.

- Define a new method when the golden rule of substitution does not hold.

- Follow the guidelines in Sect. A.3 when adding new methods or new data types to a module and be careful not to violate any existing invariants.

Reuse of a method

- Before writing client software for a method or a module with no documented contracts, identify and document their contracts.

- If a method or module that will be reused is documented with contracts, then follow the contracts when developing client software for it.

A.7 List of Terms

This list gives a short explanation of some of the terms used in this appendix. A few of them are explained more fully in Sect. A.2. They are marked with an asterisk (*).

Assertion Statement - assurance about something.

Black box* - A model of a software part where only the outside is visible.

Client - Software that is currently calling a supplier method.

Client module - A module in the role of being the client of another module.
A.7. LIST OF TERMS

Contract - Pre- and postconditions in combination with a consistent usage.

End user - A person who will use the software system that is being constructed.

Exported method - A method intended to be used by a client outside the module.

Guard method - A method expressing a precondition for another method.

Interface* - A description of how a method or a module should be used.

Internal method - A method intended to be used only from inside the module.

Invariant - A condition that is always true during the execution of a program.

Method - 1) A set of guidelines for achieving certain goals.
2) Java terminology for a function.

Module* - A software part. Examples include a method, a method library, a class and a package of classes.

Module designer - A person designing a module and its interface.

NUTEK - The Swedish National Board for Industrial and Technical Development.

Postcondition - A condition that should be true after the execution of a method.

Precondition - A condition that should be true before a method is called.

Programmer - A person implementing or maintaining a module.

Semantics* - Meaning, implication of something.

Semantic integrity* - Maintaining and respecting the semantics of a module.

Supplier method - A method that is intended to be called from a client module.

Supplier module - A module in the role of supplying services to other modules.

Syntax - Language constructions, grammatical rules.

User - A person developing a client module. Not to be mistaken for an end user.

White box - A model of a software part where both the outside and the inside are visible.
A.8 General Guidelines

This section contains important guidelines for defining modules and for abstracting data and methods. The guidelines under this header can help during system design. They have an important semantic impact and much of the literature on software engineering and object oriented software construction mentions them. Refer to these guidelines if the method you are normally using does not already include them.

A.8.1 Use a Modular Design

*Description and motivation:* Modularity implies that something is split into small, well-defined parts. This is desirable since small parts are easier to read, understand and maintain than larger ones. Well-defined modules can easily be combined. Modularity also helps to understand the program as a whole, since the individual parts are easier to understand.

*Rule:* Split the program into modules, which should be well defined, complete and coherent.

*Procedure:* A good module should define one concept and do it well. A way to test a candidate module is to try to describe it in one short sentence. A module is complete if its functionality can be explained without resorting to other modules. A module is coherent if it does not include several concepts. In Java, a module is normally a class. Sometimes it can be an aggregation of tightly cooperating classes, defined as a package.

*Example 28: Modularity*

This example illustrates a reasonable division into modules. A list, represented by data and operations, is defined in an abstract class. The actual implementation will be defined by a subclass. The data type for the list is defined as Object, so that the list can contain data of any object type.

```java
/* List.java */
public class List {
    ...
    public abstract List add(Object anElement);
    public abstract List remove(int position);
    ...
} // class List
```

A.8.2 Keep Down the Number of Dependencies

*Description and motivation:* There should be as few dependencies as possible between modules. This will improve the understanding and limit the propagation of errors. According to this rule, the ideal module would be one that had...
no dependencies at all. No errors could propagate to any other module and no change could affect any other module. Such a module should completely fulfill this criterion but would unfortunately also be totally useless. It is, however, reasonable to send information between as few modules as possible and to send as little information as possible between those modules.

Rule: Let a module communicate with as few other modules as possible and send as little information as possible between modules.

Procedure: Consider carefully which modules have to communicate with each other. Try hard to let each module be as independent as possible.

Example 29: Many vs. few dependencies

\begin{figure}[h]
\centering
\begin{tabular}{cc}
\textbf{Many dependencies} & \textbf{Fewer dependencies} \\
\end{tabular}
\caption{Different dependencies between modules}
\label{fig:dependency}
\end{figure}

Figure A.5 shows how five different software modules may be connected in different ways. The example to the left has more dependencies than the one to the right. The designer wants to reduce the propagation of errors and changes from one module to the others. He has therefore organized the modules to the right so that they need to communicate with as few other modules as possible.

A.8.3 Encapsulate Data and Methods

Description and motivation: The data variables should be located in the same context as the methods using them. The co-location is normally syntactical, implying that variables and functions are declared physically close to each other in the source code. Bundling variables and methods that have something in common makes future additions and maintenance of the module easier. In object-oriented languages like Java, encapsulation is easy and natural using classes.

Rule: Locate variables and the methods that operate on these variables in the same class.
**Procedure:** Start with the modules defined. For each module, see if it can be implemented as a class. Most of the time, a class will implement one module. One measure of a good class is to see whether the methods of the class use the variables defined in the same class most of the time.

**Example 30: Encapsulation**
A list module can be a linked list. The data structure for the linked list and the methods operating on it are encapsulated in a class `LinkedList` that implements the abstract module definition `List`.

```java
public class LinkedList extends List {
    private LinkedNode first;
    private int size;
    ...
    public LinkedList() {
        /* actual implementation of constructor */
    }
    public List add(Object anElement) {
        /* actual implementation */
    }
    public List remove(int position) {
        /* actual implementation */
    }
    ...
}
```

Example 30 shows how variables and methods are encapsulated by belonging to the same class.

**A.8.4 Hide Data and Methods**

**Description and motivation:** Data and methods are hidden to clarify for the user what should be known and accessible from outside a module and what should not. Hiding data and methods does not necessarily mean that they cannot possibly be reached, but it should make it more difficult to use them in a wrong way. It will never be possible to achieve total security in a system, but following this rule is a way to increase system security. At least, one signals clearly to the users what is appropriate and what is not.

**Rule:** Hide data variables and methods that can be misused.

**Procedure:** An interface is normally implemented through the methods exported from a module. Methods that may put the data in an unstable condition if used incorrectly or that break the encapsulation and data abstraction rules (see Sect. A.8.5) should be made internal and only be reachable through the defined interface. This way it will be possible to control the data access, since only “legal” access is made possible. All instance variables and `static` class variables should normally be declared `private` and only the exported methods should be
declared public. Internal methods may be declared private or protected.

Example 31: Data and method hiding
Example 31 illustrates how a special method is used to adjust a variable in the list module. The variable keeps track of the size of the list. If this method is misused, this variable will not reflect the size of the list any longer, and the semantic integrity is violated. The method is hidden in order to prevent incorrect usage.

```java
/* LinkedList.java */
public class LinkedList extends List {
    /* Exported methods */
    public List add(Object anElement) {
        ...
        return adjustSize(1);
    }
    ...
    /* Internal methods */
    private List adjustSize(int adjustment) {
        size += adjustment;
        return this;
    } // adjustSize
} // class LinkedList
```

A.8.5 Use Abstractions of Data and Methods

*Description and motivation:* Abstracting data means focusing on what the data represent rather than how they are implemented. It makes it easier both to use the data correctly and to modify the implementation. An insufficient or missing abstraction, on the other hand, may make it harder to understand the intention of a certain data element. Concerning the functional abstraction, one method should, as far as possible, implement one task only. This makes it easier to understand the purpose of each method.

*Rule:* Define and abstract the data to a level above the implementation level. Let each method in a module perform one well defined, limited task.

*Procedure:* Study and understand what the data element should represent and try to disregard how it will be implemented. This is of course not possible for simple data types, but it will pay off for structures that are more complex. As concerns the methods, they should be kept as simple as possible. If the method seems trivial and almost ridiculous, then there is probably good hope that its functionality can be understood even after a few months. It should be possible to describe the functionality of a method in one sentence. If not, the method is probably too complex.
Example 32: Abstraction

```java
class List {
    /* list methods */
    public abstract List linkedListAdd(ListData anElement);
    public abstract List linkedListRemove(int position);
    public abstract ListData linkedListGetElement(int position);
} // class List
```

The methods in the above example make explicit the fact that the class defines a linked list. However, the main point here is hardly to emphasize that the list is in fact a linked one, but that it is a list. The following example shows an abstraction, which is better for the user.

```java
class List {
    /* list methods */
    public abstract List add(Object anElement);
    public abstract List remove(int position);
    public abstract Object getElement(int position);
} // class List
```

If the user has to know that it is a linked list in order to use it, then the abstraction level is probably too low. The next version of the list may be based on dynamic arrays. Then the names will be wrong and all client code must be rewritten to fit the new list definition. This should not be necessary and can be avoided with the right level of abstraction.

A.9 Documentation and Implementation of a List

This section shows a documented and commented code for the example used throughout this method description. The module that is shown designs and implements the concept of a list. A list is an unbound sequence of data elements. New data elements may be added and existing ones may be retrieved or deleted. To limit the size of the example, some restrictions are made, for instance that an element may only be added at the first position of the list or that no mechanism for searching an element in the list is provided.

The structure of this section follows the documentation guidelines from A.4. The documentation part is followed by one possible implementation of a list. Other implementations of the same interface are possible. The implementation is accompanied by its invariants, data descriptions and constraints, according to Sect. A.4.4.

For the sake of showing alternative ways to describe the properties of a module, some preconditions and postconditions have been given more than once. In those cases, the condition is first expressed as a plain text and then is followed by one or two expressions that are more formal. The alternative descriptions are easy to recognize, since they are indented as compared to the first description.
A postcondition may contain the construction “old anExpression”. It means that the value of anExpression at the entry to the method should be used and not the current value at the end of the method. The code “anExpression@pre” has the same meaning. The style “old anExpression” is taken from Bertrand Meyer and “anExpression@pre” is defined by OCL, the constraint language for UML. It is difficult to say which of these will be used in the future.

In an actual situation, one of the styles for describing the semantic properties should be chosen. The choice will depend on the inclination of the designer in the informal or formal direction and on the readability and precision of the result. Some people prefer to use a formal language. On the other hand, the reader should also be able to understand the formalism so that he can profit from the description. With some care it is often possible to be just as exact using plain English, but that often requires more words to express an idea than what is needed using a formal constraint language.

A.9.1 Overview

The purpose of the module

The List module implements dynamically sized lists and operations to manipulate them. The content of a list can be of any object type and is supplied to and from the module through an Object reference.

General data description

The list module defines one data type, List, with the following general characteristics:

- A list may contain data of any object type.
- Creating and managing the contained data objects is the responsibility of the client.
- The maximum size of a list is only limited by the amount of working memory available.

General instructions for use

The module contains three categories of methods: to create, manage and erase lists.

- Creation is done through the constructor, which also initializes the data structure.
- The managing methods are those used to add, remove and get list elements and to retrieve other list information.
When the list is no longer needed, it should be erased.

A.9.2 Interface

Defined data types and their invariants

List

Description
The data type represents a list. A List variable is a pointer to an instance of the class List.

Declaration example
List myList;

Properties
A list is created and initialized through a constructor and assigned to a List variable. All List methods are applied to a valid List variable. When a list is no longer needed, it should be erased.

A data element in a list is identified through its position. The position numbers start at zero. Position zero is said to be the front of the list. All positions in the list have real data elements. No data element in the list is null.

Invariants
- size() >= 0
- isEmpty() == (size() == 0)
- positions are counted from zero through size() - 1
- for all legal values of position: getElement(position) != null

Description of every method

Create a new list

Description
Constructor, initializes a newly created list to be empty.

Syntax
public List()
Precondition
none

Postcondition
the list is empty

Add a new data element to the front of a list

Description
Adds a new data element at the first position in the list and returns the new list. All data already in the list are moved up by one position.

Syntax
public List add(Object anElement)

Parameters
anElement the data element to be added to the list

Precondition
• anElement is not null
  • anElement != null

Postcondition
• anElement is the first data element in the list
getElement(0) == anElement
• the list size is increased by one
size() == listSize(list)@pre + 1
  size() == old size() + 1
• all data elements previously found in the list are moved up by one position
• the modified list is returned

Retrieve the data element at a given position in a list

Description Gets the data element at the position indicated in a list.

Syntax
Object getElement(int position)

Parameters
position the position of the data element requested

Precondition
• position has a value from zero through the number of list elements minus one
  \(0 \leq \text{position} < \text{size()}
  \)
  \(0 \leq \text{position} \text{ && position < size()}
  \)

• The list contains at least one element (also follows from the previous condition)
  \(\text{size() > 0}
  \)
  \(!\text{isEmpty()}
  \)

Postcondition

• the data element at position \text{position} is returned

Remove the data element at a given position from a list

Description
Removes the data element at the position indicated from the list and returns the modified list. All data elements that were previously behind the element removed are moved down by one position. The data element itself is not affected by the operation. It is only removed from the list.

Syntax
List remove(int position)

Parameters
position the position of the data element to remove

Precondition

• position has a value from zero through the number of list elements minus one
  \(0 \leq \text{position} < \text{size()}
  \)
  \(0 \leq \text{position} \text{ && position < size()}
  \)

• the list contains at least one element (also follows from the previous condition)
  \(\text{size() > 0}
  \)
  \(!\text{isEmpty()}
  \)

Postcondition

• the occurrence of the data element which was previously at the position \text{position} is no longer in the list.

• all data elements that were located behind \text{position} are one position lower than before
• the changed list is returned

Interrogate for the size of a list

Description
Reports the number of elements contains in a list. This is called the size of the list with the new contract.

Syntax
int size()

Parameters
none

Precondition
• none

Postcondition
• the number of elements currently in the list is returned

Interrogate if a list is empty

Description
Gives a report as to whether the list is currently empty. A list is empty if it does not contain any data elements, that is, if its size is zero.

Syntax
boolean isEmpty()

Parameters
none

Precondition
• none

Postcondition
• the Boolean value if the list is empty is returned

Inquire if a position is a valid list index

Description
Reports whether the position indicated exists in the list.

Syntax
boolean isPositionOk(int position)
APPENDIX A. SEMLA

Parameters
position the position to be tested

Precondition
• none

Postcondition
• the Boolean result if position is a valid, existing position in the list is returned
  result == (0 <= position && position < size())

Erase a list

Description
Erase a list and return a result that cannot be used in list operations. The data elements that may be left in the list at the time of the call are not affected by the call.

Syntax
List erase()

Parameters none

Precondition
• none

Postcondition
• the list in which the operation was performed is emptied. Any data elements previously in the list are deleted from the list but are otherwise unaffected.

• a value that cannot be used in list operations is returned

A.9.3 Implementation and Internal Description

The comments in front of the classes and methods of the following implementation as well as the @param part of the comments are used by Javadoc to produce a documentation for the class and its methods.

The implementation consists of three classes, each in its own source code file.
List.java contains the abstract class List, which defines the exported interface for the List data type. The list implemented as a linked list in the class
A.9. DOCUMENTATION AND IMPLEMENTATION OF A LIST

LinkedList (in LinkedList.java) implements this interface. As part of its implementation, it uses node objects of the class LinkedNode. The classes List and LinkedList are public, while the class LinkedNode is restricted to the package scope.

The documentation related to the implementation is embedded as comments in the code.

List.java

/***************************************************************************/
* The abstract class List defines a limited list concept, with
* operations to add an element first in a list and to retrieve
* or delete an element of choice from the list.
* <p>
* A list can contain elements of any object-type.<br>
* Invariant:<ul>
* <li>list positions are counted from 0 through size() - 1<br>
* <li>size() &gt;= 0<br>
* <li>isEmpty() == (size() == 0)<br>
* <li>for all legal values of position:<br>
* getelement(position) != null<br>
* </ul>
* <p>
* A list default constructor should initialize the object to
* represent an empty list.
***************************************************************************/
public abstract class List

/**
 * constructor: initializes an empty list
 *
 * @param Pre true
 * @param Post a new, initialized, empty list is returned
 */
public List() {}

/**
 * add: adds anElement to the beginning of the list
 *
 * @param anElement the data element to be added
 * @param Pre anElement != null
 * @param Post anElement is the first element in the list,
 * the list size is increased by one,
 * the modified list is returned
 */
public abstract List add(Object anElement);
* the list (counted from 0)
* * Opam position the position of the element
* to be removed
* * Opam Pre isPositionOk(position)
* * Opam Post the element which was at position 'position'
* is not in the list any more,
* the list size is decreased by one,
* the modified list is returned
*****************************************************/
public abstract List remove(int position);
/**
* getElement: Returns a pointer to the element at
* position 'position' (counted from 0)
* * Opam position the position of the element
* to be returned
* * Opam Pre isPositionOk()
* * Opam Post pointer to data element at position
* 'position' is returned
*****************************************************/
public abstract Object getElement(int position);
/**
* isPositionOk: Tests if position 'position'
* is within range for reference or removal
* * Opam position the position whose legality should be
* assessed
* * Opam Pre true
* * Opam Post a boolean value telling if position is legal
* for accessing a list is returned.
* true=position ok, false=position not ok
*****************************************************/
public boolean isPositionOk(int position)
{
    return 0 <= position && position < size();
} // isPositionOk
/**
* isEmpty: Tests if list is empty
* * Opam Pre true
* * Opam Post a boolean value representing the state of the
* list is returned.
* True=list is empty, false=list is not empty
*****************************************************/
public boolean isEmpty()
A.9. DOCUMENTATION AND IMPLEMENTATION OF A LIST 179

```java
{
    return size() == 0;
} // isEmpty

/*****************************************************************************/
* size: Returns the number of elements in the list
* @param Pre true
* @param Post an integer value representing the number of
* elements in the list is returned.
*****************************************************************************/
public abstract int size();

/*****************************************************************************/
* erase: Deletes whatever is left of the list
* @param Pre true
* @param Post the remaining part of the list is deleted.
* no data element is touched, but they are removed
* from the list,
* a value which cannot be used as a list is
* returned
*****************************************************************************/
public List erase()
{
    List temp = this;
    while (!temp.isEmpty())
        temp = temp.remove(0);
    temp = null;
    return temp;
}

} // class List

LinkedList.java

/**
 * The concrete class LinkedList implements the list concept
 * defined by the abstract class List.
 */
public class LinkedList extends List
{
    //
    // Private data structure
    //
    // A list is represented by a chain of linked nodes
    // The data content are held by Object pointers
    // in the nodes
    //
// Internal invariant for LinkedList:
// size == the number of nodes in the list
// first points to the first element in the list (if any)
// first == null iff size == 0
// nodes are allocated locally by the list methods
// data objects are allocated by the client
//
// It is assumed that the size operation will be used
// frequently for the list, since it appears in several
// preconditions. This is the reason for the redundant
// size variable, which must be maintained at every
// addition and removal.
//
private LinkedNode first = null;
// starting condition, empty list
private int size = 0;
// starting condition, empty list
/**
* constructor: initializes an empty list
*
* @param Pre true
* @param Post a new, initialized, empty list is returned
*/
public LinkedList()
{
    // The default initialization will do to initialize
    // an empty list
} // LinkedList
/**
* add: adds anElement to the beginning of the list
*
* @param anElement the data element to be added
* @param Pre anElement != null
* @param Post anElement is the first element in the list,
*   the list size is increased by one,
*   the modified list is returned
*/
public List add(Object anElement)
{
    LinkedNode newNode = new LinkedNode(anElement);
    newNode.setNext(first);
    first = newNode;
    return adjustSize(1);
} // add
/**
* remove: Removes the element at position 'position' from
* the list
* @param position the index of the element to remove
* @param Pre position < size
* @param Post the modified list is returned
*/
public List remove(int position)
{...}
A.9. DOCUMENTATION AND IMPLEMENTATION OF A LIST

* the list (counted from 0)

* @param position the position of the element to be removed
* @param Pre isPositionOk(position)
* @param Post the element which was at position 'position'
  * is not in the list any more,
  * the list size is decreased by one,
  * the modified list is returned
*********************************************************/
public List remove(int position)
{
    LinkedNode toRemove; // the element to be removed
    if(position == 0) // remove first element
    {
        toRemove = first;
        first = toRemove.getNext();
    }
    else
    {
        LinkedNode previous = first; // used while searching,
        // stops one position before the
        // node to be removed
        int i = 0; // index counter
        // Invariant:
        // previous is the node at position i
        for(; i < position - 1; i++)
            previous = previous.getNext();
        // termination guaranteed by pre
        // previous is the node at the position (position-1)
        toRemove = previous.getNext();
        previous.setNext(toRemove.getNext());
    }
    toRemove.setNext(null); // satisfy the node invariant
    toRemove = null; // release the node
    return adjustSize(-1); // maintain internal invariant
} // remove

/***********************************************************/
public Object getElement(int position)
{
```java
{ LinkedList temp = first;
    int i;
    for(i = 0; i < position; i++)
        temp = temp.getNext(); // termination guaranteed by pre
    return temp.getData();
} // getElement

/******************************************************************************
* size: Returns the number of elements in the list
*
* @param Pre true
* @param Post an integer value representing the number of
* elements in the list is returned.
*******************************************************************************/
public int size()
{
    return size; // according to internal invariant
} // size

/******************************************************************************
* Internal methods
*******************************************************************************/
/******************************************************************************
* adjustSize: Adjusts the value of the size attribute
*
* @param Pre the physical list size has changed
* by adjustment
* @param Post the attribute size has been
* adjusted accordingly
*******************************************************************************/
private List adjustSize(int adjustment)
{
    size += adjustment;
    return this;
} // adjustSize

} // class LinkedList

LinkedList.java

class LinkedList
{
    /******************************************************************************
    * A linked node holds sequence information
    * and data for a linked list
    * This class is defined at the package level and will be
    * invisible outside of the package where it is defined.
    * This node defines a single linked list.
    ******************************************************************************/
```
* Invariant for ListNode, must be maintained by the list
  * next == null iff there is no following node
  ******************************************************************************/
private ListNode next;
private Object data;
/*******************************************************************************/
* constructor: Constructs a node with the data given
* * @param Pre true
* @param Post (getNext() == null) &&
* (getData() == initialData)
*******************************************************************************/
public ListNode(Object data)
{
  setNext(null);
  setData(initialData);
}
/*******************************************************************************/
* getNext: Gets the node following this one in a list
* * @param Pre true
* @param Post the node following this one is returned
*******************************************************************************/
public ListNode getNext() {return next;}
/*******************************************************************************/
* setNext: Sets the list node which should follow this one
* * @param Pre true
* @param Post getNode() == newNext
*******************************************************************************/
public void setNext(ListNode newNext)
{
  next = newNext;
}
/*******************************************************************************/
* getData: Gets the data stored in this node
* * @param Pre true
* @param Post the data stored in this node is returned
*******************************************************************************/
public Object getData() {return data;}
/*******************************************************************************/
* setData: Stores data in this node
* * @param Pre true
* @param Post getData() == newData
*******************************************************************************/
public void setData(Object newData) { data = newData;}
} // class ListNode

APPENDIX A. SEMLA
Empirical Evaluations of Semantic Aspects in Software Development

This thesis presents empirical research in the field of software development with a focus on handling semantic aspects. There is a general lack of empirical data in the field of software development. This makes it difficult for industry to choose an appropriate method for their particular needs. The lack of empirical data also makes it difficult to convey academic results to the industrial world.

This thesis tries to remedy this problem by presenting a number of empirical evaluations that have been conducted to evaluate some common approaches in the field of semantics handling. The evaluations have produced some interesting results, but their main contribution is the addition to the body of knowledge on how to perform empirical evaluations in software development. The evaluations presented in this thesis include a between-groups controlled experiment, industrial case studies and a full factorial design controlled experiment. The factorial design seems like the most promising approach to use when the number of factors that need to be controlled is high and the number of available test subjects is low.

Another contribution of the thesis is the development of a method for handling semantic aspects in an industrial setting. A background investigation performed concludes that there seems to be a gap between what academia proposes and how industry handles semantics in the development process. The proposed method aims at bridging this gap. It is based on academic results but has reduced formalism to better suit industrial needs. The method is applicable in an industrial setting without interfering too much with the normal way of working, yet providing important benefits. This method is evaluated in the empirical studies along with other methods for handling semantics.