Otto von Guericke University Magdeburg

Faculty of Computer Science
Department of Technical and Business Information Systems

Thesis

Empirical Comparison of FOSD Approaches Regarding Program Comprehension – A Feasibility Study

Author:
Janet Feigenspan
August 3, 2009

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Thomas Leich,
Dipl.-Wirt.-Inform. Christian Kästner,
Dr.-Ing. Sven Apel

Otto von Guericke University Magdeburg
Faculty of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Feigenspan, Janet
Empirical Comparison of FOSD Approaches Regarding Program Comprehension - A Feasibility Study
Thesis, Otto von Guericke University Magdeburg, 2009.

Acknowledgements

I would like to thank my advisors for their helpful comments regarding my work.

Furthermore, I would like to thank all those persons that made the experiment in Passau possible: Sven Apel and J¨org Liebig, who helped me to prepare and conduct the experiment in Passau. Andy, Andreas, Matthias, Martin, and Thomas for patiently being my pretest subjects. All students of the programming course in Passau for being my subjects.

I thank Marcus and Feffi for reading and commenting my work. Additionally, Thomas, Christian, and Chris for helping me to find programming experts.

Finally, Jan, Christian, and Knollo, who kept my spirit up during the tedious process of designing and writing this thesis.

Contents

Contents iii

List of Figures vii

List of Tables ix

List of Abbreviations xi

1 Introduction 1

2 Background 5

2.1 Feature-oriented software development 5

2.1.1 Goals of feature-oriented software development 5

2.1.2 Physical separation of concerns 10

2.1.3 Virtual separation of concerns 15

2.1.4 Summary 19

2.2 Program comprehension 20

2.2.1 Top-down models 21

2.2.2 Bottom-up models 21

2.2.3 Integrated models 22

2.2.4 Measuring program comprehension 22

2.2.5 Summary 23

2.3 Conducting experiments 24

2.3.1 Objective definition 25

2.3.2 Design 27

2.3.3 Analysis 32
2.3.4 Interpretation 42
2.3.5 Summary 43

3 Confounding Variables for Program Comprehension 45
3.1 Example scenario 45
3.2 Selection of the variables 46
3.2.1 Review of the literature 47
3.2.2 Consultation of experts 47
3.3 Personal parameters 51
3.3.1 Programming experience 53
3.3.2 Domain knowledge 56
3.3.3 Intelligence 57
3.3.4 Education 59
3.3.5 Miscellaneous 60
3.4 Environmental parameters 61
3.4.1 Training of the subjects 62
3.4.2 Motivation of the subjects 63
3.4.3 Tool support 64
3.4.4 Position and ordering effect 66
3.4.5 Effects due to experimenter 67
3.4.6 Hawthorne effect 68
3.4.7 Test effects 69
3.4.8 Miscellaneous 70
3.5 Task-related parameters 71
3.5.1 Structure of the source code 71
3.5.2 Coding conventions 72
3.5.3 Difficulty of the task 73
3.5.4 Comments and documentation 74
3.6 Programming language 75
3.7 Summary 78

4 Feasible Scope of Comparing FOSD Approaches **81**
4.1 Comparing four approaches . . . 81
4.1.1 Creating a program for each FOSD approach . . . 82
4.1.2 Recruiting and managing a sufficient number of subjects . . . 82
4.1.3 Reducing number of required subjects (and introducing test effects) . . . 83
4.1.4 Assuring generalizability of our results . . . 84
4.2 Comparing two programming languages . . . 85
4.3 Realistic comparison . . . 87
4.4 Agenda . . . 89
4.5 Summary . . . 90

5 Experiment **91**
5.1 The experiment in a nutshell . . . 91
5.2 Objective definition . . . 94
5.2.1 Independent variable . . . 94
5.2.2 Dependent variable . . . 94
5.2.3 Hypotheses . . . 95
5.3 Design . . . 96
5.3.1 Controlling personal parameters . . . 96
5.3.2 Controlling environmental parameters . . . 98
5.3.3 Controlling task-related parameters . . . 99
5.3.4 Programming language . . . 102
5.3.5 Tasks . . . 102
5.4 Execution . . . 107
5.4.1 Data collection . . . 107
5.4.2 Conducting . . . 107
5.4.3 Deviations . . . 108
5.5 Analysis . . . 109
5.5.1 Descriptive statistics . . . 109
5.5.2 Hypotheses testing . . . 111
5.6 Interpretation . . . 118
5.6.1 Evaluation of results and implications . . . 120

5.6.2 Threats to validity 122
5.6.3 Next steps 124
5.7 Summary 125

6 Related Work **127**

7 Conclusion **131**

A Additional Material **135**

Bibliography **141**

List of Figures

2.1 Tangled and object-oriented implementation of the stack example. 6
2.2 Elements of a feature diagram. 8
2.3 Feature diagram of the stack example. 9
2.4 Collaboration diagram of the stack example. 11
2.5 Aspect-oriented implementation of *Safe* and *Top* of the stack example. 13
2.6 Tangled version of the aspect-oriented implementation of *Safe* and *Top*. . . . 14
2.7 CPP implementation of the stack example (Antenna). 17
2.8 CIDE implementation of the stack example. 18
2.9 Stages of an experiment. 24
2.10 Box plot. 35
2.11 Overview of significance tests. 43

3.1 Sample page of the expert survey. 50
3.2 Sample implementation of observer pattern. 53
3.3 Implementation of quicksort algorithm in Haskell, Java, and C++. 77

4.1 CPP and CIDE implementation of the stack example. 88
4.2 AspectJ and AHEAD implementation of the stack example. 89

5.1 Comparison of CPP and CIDE version of SPL in our experiment. 93
5.2 Sample HTML file with preprocessor statements in Lines 12 and 15. 100
5.3 Feature diagram of the MobileMedia SPL in the sixth scenario. 101
5.4 Template of collaboration used in our experiment. 104
5.5 Feature interaction (shared code) in the class PhotoController. 105
5.6 Bug for M1: `bubbleSort` is not implemented. 105

5.7 Bug for M2: `increaseNumberOfViews` is not implemented. 105
5.8 Bug for M3: `viewFavoritesCommand` is not added. 106
5.9 Bug for M4: potential null pointer access (Line 11) 106
5.10 Histrograms for programming experience, age, and years since subjects study. 110
5.11 Box plots for response times. 112
5.12 Relationship between programming experience and response time for M4. . . 123

A.1 Correctly marked collaboration diagram template. 137
A.2 Introduction to task M2. 138
A.3 Task description for M2. 139

List of Tables

2.1 Overview of physical and virtual SoC. 20
2.2 Simple one-factorial design. 31
2.3 One-factorial design with repeated measures. 31
2.4 Two-factorial design. 32
2.5 Scale types and allowed operations. 33
2.6 Cases in statistical decisions. 36
2.7 Example of observed and expected frequencies. 38
2.8 Example response times for Mann-Whitney-U test. 40
2.9 Summary of relevant terms for experiments. 44

3.1 Characteristics of programmers in scenario. 46
3.2 Confounding parameters in literature. 48
3.3 Number of experts that rated a parameter as no, little, & considerable influence. 52
3.4 Two-factorial design. 55
3.5 Design for controlling position effect. 67
3.6 Overview of confounding parameters. 79

4.1 Overview of problems with confounding variables. 86

5.1 Descriptives of sample by group for nominal scaled variables. 110
5.2 Descriptives of sample by group for metric scaled variables. 111
5.3 Descriptives of response times. 113
5.4 Descriptives of subjects' opinion. 114
5.5 Mann-Whitney-U test for response times of S1 and S2. 115
5.6 Mann-Whitney-U test for response times of M1 - M4. 115

5.7 χ^2 test for number of correct tasks. 117
5.8 Mann-Whitney-U test for version. 118
5.9 Mann-Whitney-U test for motivation. 119
5.10 Mann-Whitney-U test for difficulty. 119

A.1 Questions of programming experience questionnaire used for diverse purposes. 135
A.2 Questions and their coding of programming experience questionnaire. 136

List of Abbreviations

AOP Aspect-Oriented Programming

CIDE Colored Integrated Development Environment

CPP C/C++ Preprocessor

FOP Feature-Oriented Programming

FOSD Feature-Oriented Software Development

GUI Graphical User Interface

IDE Integrated Development Environment

LOC Lines of Code

OOP Object-Oriented Programming

OS Operating System

SoC Separation of Concerns

SPL Software Product Line

Chapter 1

Introduction

The first programmable computers were developed around 1945 (NEUMANN [Neu45]). In order to program them, byte codes had to be used, which are hard to get accustomed to. This lead to a discrepancy in human and computerized way of thinking, which is referred to as *semantic gap*.

The problem with the semantic gap is that not all design information can be expressed with computerized thinking, so that during implementation, information is lost. The greater this semantic gap is, the more information gets lost. Consequences are that understandability decreases and the *legacy problem* (adjusting an existing source code to new platforms) and *evolution problem* (software systems deteriorate over time instead of being constantly improved) occur, which produce high costs for software development (CZARNECKI [Cza98]). In order to solve these problems and thus reduce software development costs, the semantic gap should be reduced.

Since the development of the first programmable computers, several software development approaches have been introduced to reduce the semantic gap, including assembler languages (SALOMON [Sal92]), procedural programming languages like Ada (CARLSON ET AL. [CDFW80]) and Fortran (BACKUS ET AL. [BBB$^+$57]), and *Object-Oriented Programming (OOP)* (MEYER [Mey97]). Today, OOP is the state of the art programming paradigm (MEYER [Mey97]). Several experiments proved the positive effect of OOP on program comprehension (e.g., HENRY ET AL. [HHL90], DALY ET AL. [DBM$^+$95]).

However, new requirements of contemporary software products exceed the limit of OOP: Today, variability of software products is crucial for successful software development (POHL ET AL. [PBvdL05]). One mechanism to provide the required variability are *Software Product Lines (SPLs)*, which are inspired by product lines in industry, like used in the production of a car or a meal at some fast food restaurants (POHL ET AL. [PBvdL05], p. 5). In order to implement SPLs, several approaches can be applied, for example *Aspect-Oriented Programming (AOP)*, *Feature-Oriented Programming (FOP)*, *C/C++ Preprocessor (CPP)*, and *Colored Integrated Development Environment (CIDE)*, which some researchers summarize as *Feature-Oriented Software Development (FOSD)* approaches (APEL AND KÄSTNER [AK09]). However, realizing variability in software also introduces new challenges to program

comprehension, because now, programs are not developed for a fixed set of requirements, but for several requirements, such that different products composed from the same SPL can fulfill different requirements (POHL ET AL. [PBvdL05]).

Although issues like scalability and design stability of FOSD approaches have been evaluated (e.g., BATORY ET AL. [BSR04], FIGUEIREDO ET AL. [FCM+08]), understandability was mostly neglected during according evaluations. Despite theoretical discussions about effects of FOSD approaches on understandability and anecdotical experience (e.g., MEZINI AND OSTERMANN [MO04], LOPEZ-HERREJON [LHBC05], APEL ET AL. [AKT07], KÄSTNER ET AL. [KAB07]), sound empirical evaluations beyond anecdotical results from case studies are missing.

It is important to gather empirical evidence regarding program comprehension to identify whether FOSD approaches provide benefit on program comprehension and under which circumstances. This would allow us to focus on those approaches that actually benefit program comprehension and to enhance them further. This way, we can reduce the semantic gap between programmers and computers and thus reduce software development costs.

With our work, we aim at increasing the knowledge base of benefits and shortcomings of different FOSD approaches on program comprehension. To the best of our knowledge, we perform the first experiment assessing the effect of different FOSD approaches on program comprehension.

We explain the particular goals of our thesis next.

Goals

In our thesis, we show whether and how FOSD approaches can be compared regarding program comprehension. In order to succeed, we defined three goals:

- Evaluate the feasibility of comparing FOSD approaches regarding their effect on program comprehension.
- Create an agenda for evaluating FOSD approaches.
- Demonstrate our results with an experiment.

Firstly, we assess the feasibility of comparing FOSD approaches to give an estimate about how time consuming and costly this is. A problem regarding feasibility emerges from the nature of program comprehension: Since it is an internal problem solving process (KOENEMANN AND ROBERTSON [KR91]), it consequently has to be assessed empirically. However, experiments can be very time consuming and costly, because lots of parameters need to be considered to draw sound conclusions (TICHY [Tic98], SHADISH ET AL. [SCC02]). Thus, a sound estimate on feasibility gives directions on how to proceed in comparing FOSD approaches.

Secondly, based on the result of the feasibility assessment, we develop an agenda for assessing the effect of different FOSD approaches on program comprehension. This way, we

and other researchers can traverse our agenda to state which FOSD approach has positive, negative, or no effect on program comprehension under which circumstances.

Thirdly, we demonstrate our results with an experiment. This way, we intend to illustrate our explanations regarding feasibility, start to evaluate the effect of FOSD approaches on program comprehension, and hope to encourage other researchers to join us in this tedious endeavor.

Structure

This thesis is structured as follows:

Chapter 2 In Chapter 2, we introduce FOSD and program comprehension, which are the focus of interest in our work. Furthermore, we give an introduction to conducting experiments.

Chapter 3 In Chapter 3, we identify confounding parameters for program comprehension and show how they can be controlled.

Chapter 4 Based on the results of Chapter 3, we evaluate the feasibility of comparing FOSD approaches regarding their effect on program comprehension in Chapter 4. Furthermore, we develop on agenda for creating a body of knowledge regarding understandability of FOSD approaches based on empirical research.

Chapter 5 We describe our experiment in Chapter 5, in which we assessed the effect of CPP and CIDE on program comprehension.

Chapter 6, 7 In Chapter 6, we relate our work to others. Eventually, we summarize our work in Section 7 and show how our work can be continued.

Chapter 2

Background

In this section, we provide the theoretical background that is necessary to understand the rest of this thesis. Since we evaluate the effect of different FOSD approaches on program comprehension, we introduce FOSD, its goals, and several approaches in Section 2.1. We continue with defining program comprehension in Section 2.2. Then, we introduce basic terms for planning, executing, analyzing, and interpreting experiments in Section 2.3. We regard this as necessary, because conducting experiments usually does not belong to the curriculum of the School of Computer Science at the University of Magdeburg, where this thesis is developed. However, readers familiar with conducting experiments may skip this section or refer to the summary presented at the end.

2.1 Feature-oriented software development

Since one our goal is to assess the understandability of FOSD approaches, we need to specify what it is. Thus, we introduce FOSD in this chapter. It describes the design and implementation of applications based on features (APEL ET AL. [ALMK08], KÄSTNER ET AL. [KTS+09]). A feature is a user-visible characteristic of a software system (CLEMENTS AND NORTHROP [CN01], p. 114). Currently, almost 20 years later, FOSD provides formalisms, methods, languages, and tools for building variable, customizable, and extensible software (APEL AND KÄSTNER [AK09]). FOSD has three goals: *Separation of Concerns (SoC)*, the development of SPLs, and stepwise development. We explain all of them in this section.

2.1.1 Goals of feature-oriented software development

FOSD aims at creating structured and reusable source code. As side effect of structured source code, different variants based on the same code base can be created, which is referred to as SPL. In this section, we explain SoC, which suggests how software should be structured in order to create reusable source code. Then, we introduce SPLs, which benefit from reusable

```
1  public class Stack {
2    public void main (String [] args){
3      LinkedList elements =
4        new LinkedList();
5      private String element1 = "e1";
6      private String element2 = "e2";
7      private String element3 = "e3";
8      elements.addFirst(element1);
9      elements.addFirst(element2);
10     elements.addFirst(element3);
11
12     System.out.println(
13       elements.getFirst());
14
15     System.out.println(
16       elements.removeFirst());
17     System.out.println(
18       elements.removeFirst());
19     System.out.println(
20       elements.removeFirst());
21
22     System.out.println(
23       elements.size());
24   }
25 }
```

(a) Tangled

```
1  public class Stack {
2    LinkedList elements =
3      new LinkedList();
4    public void push (Element element) {
5      elements.addFirst(element);
6    }
7    public Element pop () {
8      if (elements.getSize() > 0)
9        return elements.removeFirst();
10   }
11   public Element top () {
12     if (elements.getSize() > 0)
13       return elements.getFirst();
14   }
15 }
16
17 public class Element {
18   private String content;
19   public Element(String content) {
20     this.content = content;
21   }
22   public String getContent() {
23     return this.content;
24   }
25 }
```

(b) Object-oriented

Figure 2.1: Tangled and object-oriented implementation of the stack example.

source code. Finally, we explain stepwise development, which aims at reducing the complexity of a program.

2.1.1.1 Separation of concerns

A first goal of FOSD is SoC. SoC is an important concept in software engineering, which aims at increasing readability and maintainability of software systems (PARNAS [Par72], DIJKSTRA [Dij76]). In order to illustrate SoC, we show a preliminary version of a stack implementation in Java in Figure 2.1a, without SoC. Such a version could be a first test, e.g. to see if the idea of how to implement the stack works at all. In this version, it is relatively hard to identify different concerns.

In contrast to the preliminary version, we present the object-oriented implementation in Figure 2.1b. Now, several concerns can be more easily identified: The stack consists of the methods `push`, `pop`, and `top`. The method `push` puts an element on the stack, `pop` removes an element from a non-empty stack, and `top` returns an element from a non-empty stack without removing it. Different concerns of the stack are now reflected by the methods. Thus, the concerns are separated. In addition, there is a class `Element`, which assures that only elements of a certain type can be stored in the stack. Hence, we have the stack, which encapsulates the operations of the stack, and the class `Element`, which encapsulates the elements

that can be stored on a stack.

However, conventional software engineering paradigms like OOP fail to modularize a special kind of concerns, called *crosscutting concerns*. A typical example for a crosscutting concern is logging. If we extend our stack so that every access of the stack is logged, we would have to add an according statement after every piece of source code that accesses the stack, i.e. in Lines 5, 8, 9, 12, and 13 (cf. Figure 2.1b). However, this would mean that everything that has to do with logging is scattered all over the class `Stack`. This is called *code scattering*. As a consequence, if we wanted to alter the behavior of logging, we would have to consider every statement that contains logging source code. This is a tedious task and is referred to as the *feature traceability problem* (ANTONIOL ET AL. [AMGS05]).

In order to avoid the feature traceability problem, we could encapsulate everything that has to do with logging in one class. However, everything that accesses the stack would also be included in that class. Hence, the logging class now contains not only source code for logging, but is also tangled with source code that accesses the stack. This problem is called *code tangling* (KICZALES ET AL. [KLM+97]).

Thus, no matter how we modularize the source code of the stack, there is always code scattering and tangling. This is referred to as *tyranny of the dominant decomposition* (TARR ET AL. [TOHS99]). The concern that is modularized, e.g. logging, forbids other concerns to be modularized at the same time.

Of course, in this small example, code scattering and tangling are manageable. However, in a larger application, in which one concern is implemented by several classes and one class consists of several hundred *Lines of Code (LOC)*, it is impossible to keep an overview of all scattered concerns or all concerns that one class is tangled with. Examples for larger applications could be an embedded data base management systems or a web server, with logging as crosscutting concern. We chose the small stack example, because the benefits and shortcomings of the FOSD approaches can be demonstrated concisely on this small implementation.

A further benefit of modularizing concerns besides improving comprehension and maintainability is that different products can be created by simply omitting one or more concerns. For example, if it is not necessary to log every access of the stack, the logging module can be omitted. Then, there are two variants of the stack that emerge from the same code base: one with logging and one without logging. This kind of software is referred to as SPLs and constitutes the second goal of FOSD, which we explain in the next section.

2.1.1.2 Software product lines

An SPL can be defined as "[...] a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of [...] assets in a prescribed way." (CLEMENTS AND NORTHROP [CN01], p. 5). The "particular market segment" or *domain* the SPL is developed for has to be agreed on by the stakeholders of the domain, e.g. managers, developers, or customers. Examples of domains are software for diesel engines, satellite ground control system,

or software for mobile phones[1]. *Assets* can be understood as the modules that encapsulate different concerns.

Since managers and developers have a different view of the same SPL, a way of communicating features between the different stakeholders has to be found. One way are feature models. A *feature model* can be defined as hierarchically organized set of features (KANG ET AL. [KCH+90], p. 37). It describes commonalities and differences of products. A *feature diagram* is a visualization of a feature model. The elements of a feature diagram are depicted in Figure 2.2. A feature can be mandatory (filled circle) or optional (empty circle). If features are connected by *and*, it means that all features have to be selected. From features linked with *or*, at least one has to be selected, whereas from features that are linked with *alternative*, exactly one has to be selected. Products that are composed from the same SPL, but with different features, are referred to as *variants*.

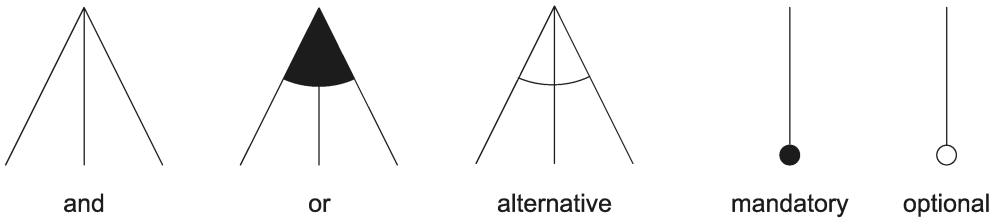


Figure 2.2: Elements of a feature diagram.

In order to make this clearer, we show the feature model of the stack example as SPL in Figure 2.3. We decomposed the object-oriented stack into four features: *Base*, *Safe*, *Top*, and *Element*. *Base* is a mandatory feature that implements the methods *push* and *pop*. The optional feature *Safe* assures that no elements can be popped or returned from an empty stack. The optional feature *Top* implements the method *top*, which returns the first element from a stack without removing it. Last, the optional feature *Element* specifies the type of the elements that can be stored on the stack. All features are linked by *or*, which means that at least one feature has to be selected. Furthermore, the feature *Base* needs to be part of every stack variant. For example, a valid variant could consist just of the feature *Base* or of the features *Base* and *Safe*.

The generation of variants can be done automatically. After the selection of features, the product can be assembled without further effort of the developer. Of course, if a customer requests a feature that is not implemented yet, this feature needs to be developed. The automatic generation of a product distinguishes SPLs from components (HEINEMANN AND COUNCILL [HC01]). Using components, the source is separated into modules (like for SPLs), however the final product needs to be assembled manually.

The development of SPLs can be regarded as a special case of stepwise or change based development, which we explain next.

[1] http://www.sei.cmu.edu/productlines/spl_case_studies.html

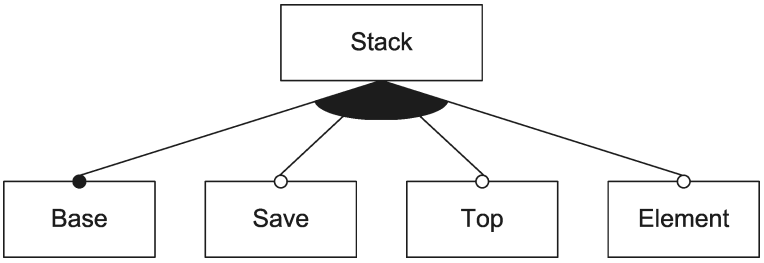


Figure 2.3: Feature diagram of the stack example.

2.1.1.3 Stepwise development

Stepwise development constitutes the third goal of FOSD. Similar to SoC, an application is split into several modules, thus reducing its complexity. This allows us building complex and powerful applications without having to manage the complexity of a large application. However, the kind of modularization in stepwise development differs from SoC.

In stepwise development, a small program is incrementally refined with details, resulting in a complex program (DIJKSTRA [Dij76], BATORY ET AL. [BSR04]). This supports the development of software, e.g., that after every refinement the program can be proven correct or its execution can be tested. Due to the incremental increase in complexity, the proofs and tests can also be incrementally refined, starting from simple and growing more complex as the program is increased in complexity.

In order to fulfill the goals of FOSD, a large number of approaches and tools were developed, which include:

- FOP (PREHOFER [Pre97])
- AOP (KICZALES ET AL. [KLM+97])
- Frameworks (JOHNSON AND FOOTE [JF88])
- Hyper/J (TARR AND OSSHER [TO01])
- Components (HEINEMANN AND COUNCILL [HC01])
- CaesarJ (ARACIC ET AL. [AGMO06])
- Aspectual Feature Modules (APEL ET AL. [ALS08])
- CPP[2]
- CIDE (KÄSTNER ET AL. [KAK08])

[2]http://www.ansi.org

- XVCL (JARZABEK ET AL. [JBZZ03])
- FEAT (ROBILLARD AND MURPHY [RM03])
- pure::variants (BEUCHE ET AL. [BPSP04])
- GEARS (KRUEGER [Kru08])

Each of those approaches focuses on one or more goals of FOSD.

Since in our work we want to evaluate the effect of several FOSD approaches on comprehension, we introduce those approaches in the next sections. We start by explaining approaches for physical SoC, which physically separate different modules encapsulating different concerns. Then, we continue with approaches for virtual SoC, which annotate source code belonging to concerns and thus virtually separating them.

2.1.2 Physical separation of concerns

In this section, we explain physical SoC. Using physical SoC, modules encapsulating one concern are physically separated, e.g. in different files or folders. In contrast to physical SoC, in virtual SoC source code for several concerns is not physically separated, but only annotated according to the concern it belongs to.

Well known approaches that use physical SoC are – among others – FOP (PREHOFER [Pre97]), AOP (KICZALES ET AL. [KLM+97]), Hyper/J (TARR AND OSSHER [TO01]), and CaesarJ (ARACIC ET AL. [AGMO06]), and aspectual feature modules (APEL ET AL. [ALS08]). Since we are interested in the program comprehension of FOP and AOP, we introduce them in the next sections. We chose FOP and AOP, because they are well known and their advantages and disadvantages with respect to readability and maintainability are discussed extensively (e.g., MEZINI AND OSTERMANN [MO04], and LOPEZ-HERREJON ET AL. [LHBC05], APEL ET AL. [AKT07]). After introducing FOP and AOP, we introduce further, related approaches for physical SoC in order to have a more complete overview.

2.1.2.1 Feature-oriented programming

The idea of FOP was first discussed by Prehofer in 1997 (PREHOFER [Pre97]). It is an extension to OOP and is designed to modularize crosscutting concerns. In FOP, concerns are encapsulated in units, called feature modules. In our example, the source code for each of the features *Base*, *Safe*, *Top*, and *Element* would be encapsulated in one feature module per feature. If the feature *Log*, which logs every access to the stack, was introduced, it would also be encapsulated in one feature module. This way, code scattering would be avoided, because everything that has to do with logging is encapsulated in one feature module. Furthermore, there is no code tangling, because only source code that handles logging is in this feature module.

How are feature modules implemented to provide a modular structure? There are several tools supporting FOP, e.g. AHEAD (BATORY ET AL. [BSR04]), FeatureC++ (APEL ET AL.

[ALRS05]), and FeatureHouse (APEL ET AL. [AKL09]). We explain the approach AHEAD uses, because its mechanism is easily understandable. We do not explain further approaches, because we only want to give an impression on the implementation of feature modules and the other tools work similarly. Hence, it suffices to explain how one of those tools works. We use Java examples to explain AHEAD, however the concepts can be applied to other programming languages, too.

In AHEAD, feature modules are stored in folders. For every feature module, one folder exists. In the stack example, there would be the folders *Base*, *Safe*, *Top*, and *Element*. Each folder contains only those parts of source code that implement the according feature. This way, the modularization is also visible to the user. AHEAD uses the concept of classes supported by Java for modularization. In order to enable the separation of crosscutting concerns, one class is separated into several *roles* and stored in different feature modules. Each role contains only source code that belongs to the feature module it is stored in.

In order to make this clearer, we show a collaboration diagram and according source code of the stack implementation in Figure 2.4. The solid lines denote the classes `Stack` and `Element`. The dashed lines denote the feature modules *Base*, *Safe*, *Top* and *Element*. The dotted lines constitute the roles that contain the implementation of a feature module of a class. All roles belonging to one feature module are referred to as *collaboration*.

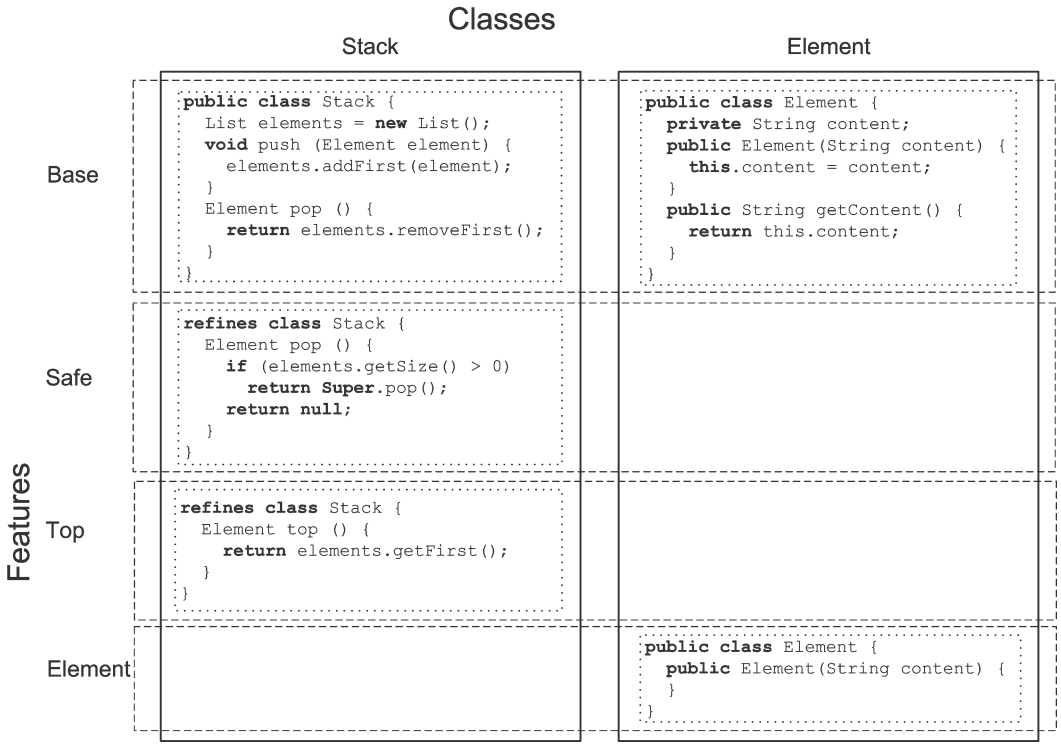


Figure 2.4: Collaboration diagram of the stack example.

Since the classes are now split into several roles, they have to be composed to build an executable program. In order to compose a program, the features that should be contained in

the stack have to be selected. For example, the features *Base* and *Safe* could be selected for the stack, or the features *Base*, *Top*, and *Element*. The selection of features must not violate the conditions stated in the feature model, e.g. that *Base* has to be selected for every stack variant.

In AHEAD, the technical realization of the composition is enabled by the Jak language, which extends Java by a few keywords (BATORY ET AL. [BLS98]). For example, in the role of the class `Stack` in the collaboration *Safe*, the keyword `refines` is used. This means that this role extends the class `Stack` of the collaboration *Base*. In the same role, the keyword `Super` is used, which refers to the method `pop` of the super class. When composing a program, AHEAD builds inheritance chains of the roles, resulting in the program according to the selected features (SMARAGDAKIS AND BATORY [SB98]).

FOP supports the creation of SPLs, because a program is composed of different feature modules, according to the selected features. Furthermore, the complexity of a program can be increased incrementally, supported, e.g., by the inheritance mechanism of AHEAD. This way, stepwise development is enabled (BATORY ET AL. [BSR04]).

To summarize, FOP divides source code into feature modules and thus allows modularizing (crosscutting) concerns. There are several tools supporting FOP, e.g. AHEAD. FOP is one approach we are interested in that uses physical SoC. A further approach is AOP, which we explain next.

2.1.2.2 Aspect-oriented programming

AOP was introduced by Kiczales et al. in 1997 (KICZALES ET AL. [KLM+97]). The primary goal for AOP is SoC. Recently, some researchers examine if AOP is also suitable for the development of SPLs (FIGUEIREDO ET AL. [FCM+08]). Like FOP, AOP was designed to modularize crosscutting concerns and thus avoid code scattering and tangling. However, the mechanism it uses to achieve this aim is different.

In AOP, concerns are encapsulated in aspects. An *aspect* contains source code that allows us to alter the behavior of a base program during runtime. There are three different kinds of source code in an aspect: inter-type declarations, pointcuts, and advice. An inter-type declaration adds a method to the base source code. An *advice* implements the new behavior that should be applied to the base program. In a *pointcut*, the events of the base source code at which the new behavior should be applied are specified. Events occurring during the execution of a program are referred to as *joint points*. Examples are the initializing of a variable or the call to a method.

In order to make those terms clearer, we implemented the stack example in AspectJ, a language supporting AOP for Java (KICZALES ET AL. [KHH+01]). We chose AspectJ, because it is one of the most sophisticated languages for AOP and has well developed tool support (STEIMANN [Ste06]). In Figure 2.5, we show the source code of an aspect implementing the feature *Safe* and *Top*. The implementation of the feature *Base* remains the same (cf. Figure 2.4).

In Lines 1–5, the feature *Top* is implemented. It adds the method `top` via inter-type declaration (`Stack.top`; Line 3) to the class `Stack`.

```
1  public aspect Top {
2    public Element Stack.top() {
3        return elements.getFirst();
4    }
5  }
6  public aspect Safe {
7    pointcut safePop(Stack stack): execution(Element pop()) && this(stack);
8    Element around(Stack stack): safePop(stack) {
9      if (stack.items.size() > 0) return proceed(stack);
10     return null;
11   }
12 }
```

Figure 2.5: Aspect-oriented implementation of *Safe* and *Top* of the stack example.

The feature *Safe* is implemented in Lines 7–16. In Line 8, the pointcut is defined that describes the execution of the methods `pop` and `top`. The keyword `execution` means that this pointcut is activated, when the methods `pop` or `top` are starting to be executed. Using `||`, the two join points for the execution of the methods `pop` and `top` are grouped. In Lines 10–15, the advice is defined, which ensures that no elements can be popped or topped from an empty stack. The keyword `thisJoinPoint` allows us to access all information that is relevant during the execution of the join point, e.g. the instance of the stack that calls the method `pop` or `top`. Then, we check if the stack is empty and proceed (`proceed()`, Line 13), if this is not the case.

An aspect weaver *weaves* the source code of an aspect into the base source code. This weaving can be statically, i.e., at compile time, or dynamically, i.e., at runtime. However, dynamic weaving is only possible if the underlying language supports changing the behavior of a program during runtime. Since Java alone does not allow such a dynamic change, the weaving in AspectJ is static and the compiled program can then be executed.

The small stack example showed that AspectJ allows us to extend a base program at far more point than FOP languages. Furthermore, AspectJ provides much more ways than the few we showed to alter a base program. Due to the numerous ways to extend a program, AOP languages are rather complex and take time to learn. A further problem is that there are no restrictions on how to structure an aspect. In our example, we defined for the features *Safe* and *Top* one aspect, each. However, this way of modularizing is left to the discipline of the programmer. It is also possible to encapsulate the source code for all features in one aspect. In Figure 2.6, we show this version of the aspect, which is now tangled with the two concerns *Top* and *Safe*.

Those problems of AOP do usually not occur in FOP. A modular structure is enforced, e.g., that for every feature a feature module exists. Furthermore, FOP does not provide so many ways to extend a base program like AOP, thus limiting the complexity. Some approaches of FOP even allow a composition without introducing new keywords, e.g., FeatureHouse (APEL AND LENGAUER [AL08]). However, this leads to restricted possibilities to extend a base program, i.e., extensions are only possible for a limited number of events in the base program.

With expressiveness and modularity, we addressed two of the issues that are discussed

```
1  public aspect Features {
2    public Element Stack.top() {
3        return elements.getFirst();
4    }
5    pointcut safePop(Stack stack): execution(Element pop()) && this(stack);
6    Element around(Stack stack): safePop(stack) {
7      if (stack.items.size() > 0) return proceed(stack);
8      return null;
9    }
10 }
```

Figure 2.6: Tangled version of the aspect-oriented implementation of *Safe* and *Top*.

when FOP and AOP are compared. A detailed evaluation of both approaches can be found in APEL [Ape07]. Researchers still disagree on what approach is better under which circumstances, especially regarding subjective issues like program comprehension or maintainability (e.g., MEZINI AND OSTERMANN [MO04], and LOPEZ-HERREJON ET AL. [LHBC05], APEL ET AL. [AKT07]). In our thesis, we want to provide the first step in concluding this discussion.

With FOP and AOP, we presented two approaches for physical SoC that we are interested in this thesis. For a more complete overview of physical SoC, we present some further approaches in the next section.

2.1.2.3 Other approaches for physical separation of concerns

Further approaches for physical SoC are, for example, aspectual feature modules (APEL ET AL. [ALS08]), Hyper/J (TARR AND OSSHER [TO01]), CaesarJ (ARACIC ET AL. [AGMO06]), frameworks (JOHNSON AND FOOTE [JF88]), and components (HEINEMANN AND COUNCILL [HC01]).

Aspectual feature modules can be seen as combination of FOP and AOP (APEL ET AL. [ALS08]). They support SoC into feature modules like in FOP, but also the flexibility for extending a base program, like in AOP.

Hyper/J supports multi-dimensional SoC for Java (TARR AND OSSHER [TO01]). It provides a complex composition mechanism similar to AHEAD (BATORY ET AL. [BLS03]), however the focus lies on dimensions, to which SoC should be applied. In addition to one-dimensional SoC, multi-dimensional SoC allows to have different modularizations of the same software system at the same time. This way, the modularization that is needed, e.g. according to classes or to features, can be produced. This allows us to structure the software system according to the need of the development task (TARR ET AL. [TOHS99]).

For realizing multi-dimensional SoC, *hyperslices* are defined. Hyperslices encapsulate concerns in different dimensions and can consist of overlapping content. They can be composed to new hyperslices or *hypermodules*, which themselves can be composed to new hyperslices or hypermodules. The composition can be defined as needed by rules. Subject-oriented programming is one paradigm realizing this kind of multi-dimensional SoC. It decomposes source code according to subjects, which are equivalent to hyperslices. A *subject* is under-

stood as collection of software units, providing a specific view on a domain (HARRISON AND OSSHER [HO93]). Hence, the composition of software units to different subjects provides different views on the same domain, without implementing for every view a new software module.

CaesarJ is a Java-based aspect-oriented programming language. It is designed to support the reuse of software (ARACIC ET AL. [AGMO06]). The benefit compared to AOP is that CaesarJ provides better support for modularity. For example, several classes implementing one concern can be grouped to new a class. For this new class, one or more interfaces can be defined. Then, in another module, the binding of the new class to a software application can be defined. This improves modularity and reusability, since the behavior of the component is decoupled from a software application. For another application, another binding of the same group of classes with the same set of interfaces can be defined.

There are several further approaches that aim at SoC, e.g. frameworks (JOHNSON AND FOOTE [JF88]) and components (HEINEMANN AND COUNCILL [HC01]). Both approaches are typical in praxis, e.g. for Eclipse (www.eclipse.org). However, all those concepts are not able to modularize crosscutting concerns, which is why we do not explain them further.

This concludes the introduction of physical SoC. We give an overview of all approaches presented in this section in Table 2.1 on page 20. In the next section, we present approaches for virtual SoC, where different concerns are not physically separated, but virtually by using annotations.

2.1.3 Virtual separation of concerns

In the last section, we explained physical SoC, where concerns are physically separated, e.g., in different feature modules. This approach is suitable for the development of new software. However, if a legacy application should be refactored into features, e.g. in order to create a software product line, alternatives have been discussed. The problem with using physical SoC is that it influences the existing code base and development process (KÄSTNER AND APEL [KA08]). Hence, approaches for virtual SoC were developed, because they allow to refactor source code without changing the actual code base.

Like for physical SoC, there are several approaches that aim at virtual SoC, including CPP[3], CIDE (KÄSTNER ET AL. [KAK08]), XVCL (JARZABEK ET AL. [JBZZ03]), FEAT (ROBILLARD AND MURPHY [RM03]), pure::variants (BEUCHE ET AL. [BPSP04]), and GEARS (KRUEGER [Kru08]). We first present CPP and CIDE in detail like FOP and AOP, since those are the two approaches we are interested in. We choose the CPP, because a large number of applications, especially for embedded systems, is implemented in C or C++ (BARR [Bar99]). CIDE is an interesting approach, because colors are recognized easily by humans (GOLDSTEIN [Gol02], p. 165) and thus can aid the comprehension of programs. For completeness, we conclude with a short overview of the other approaches.

[3]http://www.ansi.org

2.1.3.1 C/C++ preprocessor

The CPP is part of the C programming language.[4] It is executed before the compiler. It can be used to define constants, include files, or allow conditional compiling, which is supported by several macros. Conditional compiling is used to create SPLs, one of the goals of FOSD (KÄSTNER ET AL. [KAK08]). CPP is a generic term for several preprocessors, for example Munge[5] and Antenna[6], which constitute preprocessors for Java.

In order to define constants, the macro `#define` can be used. For example, in C there is no boolean data type. However, using `#define`, `True` and `False` can be defined by the programmer, e.g. `#define True = 1` and `#define False = 0`. Then, both constants can be used.

Next, there is an `#include`, which is used to include further files. This is useful for modular source code, because source code implementing different concerns can be stored in different files. For example, we could store the implementation of the feature *Base* in a file called `base`, and the implementation of the feature *Top* in a file called `top`. In the base file, we would have to add the statement `#include top`. When the preprocessor is called, it replaces this statement with the content of the according file, in this case the implementation of the method `top`.

Finally, there are macros that are used for conditional compiling of source code, i.e., `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`. Conditional compiling can be used to create different configurations of a program. For example, we could use those macros for our stack to create different variants of the stack, e.g. one variant that contains all features, or a variant that only contains the features *Base*, *Safe*, and *Top*.

In order to aid understanding of those macros, we show our stack example in C and with the CPP statements in Figure 2.7. The implementation of the class `Element` remains unaltered (cf. Figure 2.1b).

In Lines 13–18, the method `top` is defined, surrounded by `#if TOP` and `#endif`. This means that the method `top` would only be compiled, if the feature *Top* was selected. The feature *Safe* is implemented in the Lines 7–9 and 14–16. If the feature *Safe* was selected, those lines would be compiled, otherwise they would be deleted by the CPP.

The CPP is not based on the specification of C or C++, but parses the directives line based. Hence, the CPP can also be used for other languages, e.g. Java. However, as a further consequence, the CPP allows to annotate everything, e.g., an opening bracket, but not the according closing one. Thus, the CPP does not enforce disciplined annotation, which is often criticized and made responsible for obfuscated source code (ERNST ET AL. [EBN02]). The ways of creating obfuscated yet functional source code are so numerous that there even is a yearly competition to create the most obfuscated C program[7].

Because of the undisciplined annotation approach, further approaches were developed,

[4] http://www.ansi.org
[5] http://weblogs.java.net/blog/tball/archive/munge/doc/Munge.html
[6] http://antenna.sourceforge.net/wtkpreprocess.php
[7] http://www.ioccc.org

```
1  public class Stack {
2    LinkedList<Element> elements = new LinkedList<Element>();
3    public void push(Element element) {
4      elements.addFirst(element);
5    }
6    public Element pop() {
7      // #ifdef SAFE
8      if (elements.size == 0) return null;
9      // #endif
10     return elements.removeFirst();
11   }
12   // #ifdef TOP
13   public Element top() {
14     // #ifdef SAFE
15     if (elements.size == 0) return null;
16     // #endif
17       return elements.getFirst();
18   }
19   // #endif
20 }
```

Figure 2.7: CPP implementation of the stack example (Antenna).

e.g., CIDE. In addition to enforcing disciplined annotations, CIDE is also language independent and uses colors instead of macros. We explain the underlying concepts of CIDE in the next section.

2.1.3.2 Colored Integrated Development Environment

CIDE was developed at the University of Magdeburg (KÄSTNER ET AL. [KAK08]). Originally, it was designed for annotating Java programs. For better applicability, a language-independent extension also exists (KÄSTNER ET AL. [KAT+09]). Using CIDE, SoC and the development of SPLs is supported.

Instead of textual macros like with the CPP, in CIDE, features are annotated in a tool infrastructure and are represented with colors. CIDE provides a source code editor, in which software developers can select source code from a file and associate this source code with a feature. Then, the chosen source code is highlighted in the editor with a background color that is associated with the feature. If a source code belongs to two features, then the background color of this source code is the color that results when blending the two colors. For example, if one feature is annotated with red, the other with blue, then the source code belonging to both features is annotated with violet.

For internal representation, CIDE uses the abstract syntax tree of the according source code file. Only *optional* nodes of the abstract syntax trees can be marked. This assures that not arbitrary text, but only classes, methods, or statements can be annotated. This is in contrast to the CPP, which allows to annotate everything.

For creating a program, the software developer chooses the features that should be contained in the program. Then, CIDE creates a copy of the project and removes the source code of all features that are not selected. The removal of the source code is implemented as

```
1   class Stack {
2     LinkedList elements = new LinkedList();
3     void push (Element element) {
4       elements.addFirst(element);
5     }
6     Element pop () {
7       if (elements.getSize() == 0) return null;
8         return elements.removeFirst();
9     }
10    Element top () {return elements.getFirst();}
11  }
```

Figure 2.8: CIDE implementation of the stack example.

operation on the according abstract syntax trees.

In order to make this clearer, we present the stack example with the features *Base*, *Safe*, and *Top* in Figure 2.8. Since the feature *Base* must be part of every stack variant, it is not colored and thus never deleted during creating a program. Now, if, e.g., the features *Base* and *Safe* would be selected, the implementation of the method `top` would be removed from the class, leaving only the field `elements` and the two methods `push` and `pop`.

One of the most often discussed problems of the CPP is the possibility to annotate everything, leading to obfuscated source code (ERNST ET AL. [EBN02]). With CIDE, a *disciplined* annotation of source code is enforced, e.g. when an opening bracket is annotated, the closing bracket has to be annotated, too. Hence, it is assumed that the approach of CIDE increases the comprehension of a program. A further assumed benefit is the use of colors instead of macros. This is based on the fact that colors are processed by the brain preattentive, whereas text based statements take more time to process (GOLDSTEIN [Gol02], p. 165).

However, problems with CIDE occur if there are feature interactions. For example, when a red and a blue annotated feature interact, source code belonging to both features is annotated with violet, the color resulting when blending red and blue. What if three, four, or ten features interact? The resulting color of the blending process would become grey. This makes the identification of all interacting features very difficult. Hence, the advantage of colors compared to textual annotations would vanish. In our thesis, we want to evaluate if CIDE indeed provides the claimed benefits compared to the CPP regarding program comprehension and to what degree of feature interactions this benefit holds.

The CPP and CIDE are the two concepts for virtual SoC we are concerned with in this thesis. In contrast to CPP, in CIDE annotations are realized with colors and it is only possible to annotate elements of the AST, which restricts the ways to produce obfuscated and hence badly comprehensible source code. In order to have a more complete overview, we present some further approaches for virtual SoC in the next section.

2.1.3.3 Other approaches for virtual separation of concerns

Further approaches for virtual SoC include, XVCL (JARZABEK ET AL. [JBZZ03]), FEAT (ROBILLARD AND MURPHY [RM03]), pure::variants from pure::systems (BEUCHE ET AL. [BPSP04]), and GEARS from BigLever (KRUEGER [Kru08]).

XVCL aims at increasing reusability and maintainability (JARZABEK ET AL. [JBZZ03]). It represents recurring software artifacts in a generic form. The according source code of those artifacts is annotated by an *x-frame*. From the generic representation of source code, concrete instances can be generated according to specified requirements. Furthermore, mechanisms like #ifdefs are supported.

Another tool is called FEAT and was developed to locate concerns in Java programs (ROBILLARD AND MURPHY [RM03]). For the localization, FEAT provides a *Graphical User Interface (GUI)*. In FEAT, concerns are described as a graph, called *Concern Graph*. Classes, methods, or fields belonging to one concern can be annotated and are then added to the Concern Graph (ROBILLARD AND MURPHY [RM02]). This supports developers in locating source code belonging to a certain concern and maintaining the according source code.

Pure::variants and GEARS are both commercial tools for the development of software product lines (BEUCHE ET AL. [BPSP04], KRUEGER [Kru08]). Like for CPP and CIDE, their annotation approach is language independent. The typical way of annotating source code is similar to the CPP, however both tools use a generic preprocessor that is not restricted to source code written in C or C++. All approaches and tools, except FEAT, support the development of SPLs.

Further approaches supporting virtual SoC include explicit programming (BRYANT ET AL. [BCVM02]), software plans (COPPIT AND COX [CC04]), and Spoon (PAWLAK [Paw06]). In the next section, we summarize the FOSD approaches we presented here.

2.1.4 Summary

All approaches we discussed are depicted in Table 2.1. In the column *in this thesis*, we present those approaches that we are concerned with in our work. The approaches we mentioned for completeness are shown in the column *other*. We selected FOP and AOP, because their benefits and shortcomings regarding subjective factors like comprehensibility are theoretically discussed for several years now (MEZINI AND OSTERMANN [MO04], or LOPEZ-HERREJON ET AL. [LHBC05], APEL ET AL. [AKT07]). However, except for some case studies (e.g., LOPEZ-HERREJON ET AL. [LHBC05] or APEL AND BATORY [AB06]), we found no papers reporting systematical empirical evidence. Although case studies suffice to prove the concept of something, they are unsuitable as sole empirical evidence, because they constitute just one example. Experiments with several subjects and several replications, on the other hand, provide the base for drawing reliable and valid conclusions.

The CPP is of interest, because today most of the applications, especially for embedded systems, are implemented in C or C++ (BARR [Bar99]), yet there are some concerns when using the CPP (ERNST ET AL. [EBN02]). As for CIDE, it is assumed that it overcomes

SoC	Approaches in this thesis	Further approaches
Physical	AOP	frameworks
	FOP	Hyper/J
		components
		CaesarJ
		aspectual feature modules
Virtual	CPP	XVCL
	CIDE	FEAT
		GEARS
		pure::variants
		explicit programming
		software plans
		Spoon

Table 2.1: Overview of physical and virtual SoC.

some shortcomings of the CPP, especially the chance of producing obfuscated source code. However, empirical evidence about the advantage regarding comprehensibility is lacking – at least to the best of our knowledge.

We presented the other approaches, because we want to give an impression on the variety of FOSD approaches. Comprehensive overviews on tools and languages supporting SoC can be found in APEL [Ape07], KÄSTNER ET AL. [KAK08], or APEL ET AL. [AKGL09].

In this section, we explained several approaches we are interested in. In the next section, we introduce program comprehension, because we are interested in the effect of FOSD approaches on program comprehension. In order to soundly analyze the effect on program comprehension, we need to be clear on what it is.

2.2 Program comprehension

Program comprehension is a hypothesis-driven problem solving process that can be defined as 'the process of understanding a program code unfamiliar to the programmer' (KOENEMANN AND ROBERTSON [KR91]). The concrete specification of this process depends on how much knowledge a programmer can apply to understand a program. The more knowledge of a program's domain a programmer has, the more he can apply. There are numerous models that explain the process of understanding, depending on the domain knowledge of a programmer. They can be divided into three different groups: top-down models, bottom-up models, and integrated models. In order to understand program comprehension, we need to understand how the process of program comprehension can be influenced by domain knowledge. Hence, we take a closer look at those models.

2.2.1 Top-down models

If a programmer is familiar with a program's domain, the process of understanding the program is a top-down process. *Top-down* comprehension means that first, a general hypothesis about a program's purpose is derived. This can only be done if a programmer is familiar with a program's domain, because otherwise he has no knowledge of examples and properties of that domain. Hence, he could not compare the current program with programs he know, at least not without closely examining what a program does. During deriving this general purpose hypothesis, the programmer neglects details, but only concentrates on relevant aspects for building his hypothesis.

Once a hypothesis of the general purpose of a domain is stated, a programmer evaluates his hypothesis. Now, he starts to look at details and refines his hypothesis stepwise by developing subsidiary hypotheses. These subsidiary hypotheses are refined further, until the programmer has a low level understanding of the source code, such that he can verify, modify, or reject his hypotheses. An example of how a top-down process looks like can be found in SHAFT AND VESSEY [SV95]. They let programmers think aloud while they were understanding a program, which made the process of understanding visible for the experimenters.

Based on top-down comprehension in general, several examples of top-down models exist: BROOKS [Bro78] introduces the term *beacons*, which are defined as 'sets of features that typically indicate the occurrence of certain structures or operations in the code' (BROOKS [Bro83]). Programmers search beacons to verify their hypotheses. Similar to beacons, programming plans ('program fragments that represent stereotypic action sequences in programming') are used to evaluate hypotheses about programs (SOLOWAY AND EHRLICH [SE84]).

If a programmer is not familiar with a program's domain, the process of understanding the program is bottom up, which we explain next.

2.2.2 Bottom-up models

Without domain knowledge a programmer cannot look for beacons or programming plans, because he does not know how they look. Hence, he needs to examine the code closely to be able to state hypotheses of a program purpose. In this case, a programmer starts to understand a program by examining details of a program first: the statements or control constructs that comprise the program. Statements that semantically belong together are grouped into higher level abstractions, which are called *chunks*. If enough chunks are created, a programmer can leave the statement level and integrate those chunks to further higher level abstractions.

For example, a programmer examines several methods of a program. If we finds out that some of those methods have a higher level purpose (e.g., sorting a list of data base entries), he groups them to chunks. Now, he does not think of these methods as single entities anymore, but as the chunk that 'sorts data base entries'. He examines the program further and discovers further chunks (e.g., inserting entries in a database, deleting entries from a database). Now, these chunks are integrated in a larger chunk, which the programmer refers to as 'data base manipulating' chunk. This process continues until the programmer has a high level hypotheses

of a program' purpose.

Examples of bottom-up theories differ on the kind of information that is integrated to chunks. PENNINGTON [Pen87] states that control constructs (e.g., sequences or iterations) are used as base for chunking, whereas SHNEIDERMAN AND MAYER [SM79] claim that chunking begins on the statements of a program. Now, what if a programmer has little knowledge about a program's domain, so that neither bottom-up nor top-down models can explain program comprehension? The answer to this question can be found in integrated models.

2.2.3 Integrated models

A solely top-down or bottom-up approach is often insufficient to explain program comprehension. Hence, integrated models were developed, which combine both approaches. For example, if a programmer has domain knowledge about a program, he forms a hypothesis about its purpose. During the comprehension process, he encounters several fragments he cannot explain using his domain knowledge. Hence, he starts to examine the program statement for statement, and integrates the newly acquired knowledge in his hypotheses about the source code. Usually, programmers use top-down comprehension where possible, because it is more efficient than examining the code statement by statement (SHAFT AND VESSEY [SV95]).

One example for integrated models can be found in MAYRHAUSER AND VANS [vMV93]. They divide the process of comprehension into four processes, where three of them are comprehension processes that construct an understanding of the code and the fourth provides the knowledge necessary for the current comprehension task. All four processes are necessary to understand an unfamiliar program.

Why is it important to deal with different models of program comprehension? It is necessary because they explain the process of understanding differently. If we do not take this into account, we cannot soundly assess the effect of different FOSD approaches on program comprehension: It could be possible that one FOSD approach has a positive effect on one model and a negative effect on the other model. Since we have no empirical knowledge about the understandability of FOSD approaches on program comprehension, we need to carefully consider the models of program comprehension when designing experiments. Otherwise, we cannot draw sound conclusions from our result.

For the same reasons it is necessary to know how program comprehension measured, which we discuss in the next section.

2.2.4 Measuring program comprehension

Program comprehension is an internal process or *latent variable* (i.e., we cannot observe it directly), but have to find *indicator* variables to measure it (COHEN AND COHEN [CC83], p. 374). One technique that was developed in cognitive psychology is called *introspection* or *think-aloud protocol* (WUNDT [Wun74]), in which individuals say their thoughts out loud.

This way, the comprehension process can be observed. Usually, think-aloud sessions are recorded on audio- or videotape, which are used to analyze the comprehension process.

In order to ensure an objective analysis of the data, several mechanisms were developed: The analysis can be done by persons that do not know what we expect from the results so that they are not influenced (intentionally or unintentionally) to interpret statements in favor or against our expectations. Furthermore, we could let several persons analyze the think-aloud protocols and compare their answers. Another way is to state specific rules on how to analyze the data. Examples of rules can be found in SHAFT AND VESSEY [SV95] and LANG AND MAYRHAUSER [LvM97].

One major problem with think-aloud protocols is that they are time consuming and costly (SOMEREN ET AL [SBS94], p. 45): sessions of subjects can audio- and videotaped, subjects' statements need to be analyzed carefully according to rules and/or by different persons to ensure objective analysis of the data. Hence, they can only be applied if we have the according resources for our experiment or we restrict our sample to only a few subjects.

Due to the effort in applying think-aloud protocols, other techniques for assessing comprehension processes have developed. An overview of such measures as well as their reliability is reported in DUNSMORE AND ROPER [DR00]. They found four techniques: maintenance tasks, mental simulation, static, and subjective rating.

In maintenance tasks, subjects are asked to add or remove statements or debug source code. In mental simulation, pen-and-paper execution of source code or optimization tasks is used. Static techniques include grouping code fragments to bigger fragments in a stepwise manner, labeling/explaining code fragments, and call graphs. Subjective rating requires subjects to estimate how well they understand source code. According to DUNSMORE AND ROPER [DR00], mental simulation and certain maintenance tasks are reliable measures, whereas static techniques are either unreliable or very hard to develop.

Secondly, after subjects worked with source code, we have to assess whether and how subjects understood source code. Since program comprehension is an internal process, also called *latent variable*, we cannot directly assess it, but we have to find *measured variables* or *indicators* (SHADISH [SCC02], p. 400). Typical measures for program comprehension usually are completeness, correctness, or time subjects needed to solve a task (DUNSMORE AND ROPER [DR00]).

The tasks and measures for program comprehension determine the analysis methods that can be applied. This is an important decision, because depending on the hypotheses of the experiment, certain analysis methods have to be applied in order to be able to verify or reject hypotheses.

2.2.5 Summary

In this section, we introduced program comprehension as hypothesis-driven problem solving process. The nature of this process depends on the amount of domain knowledge and can be top down, bottom up, or a mixture of both. In order to draw sound conclusions from experimental results, it is necessary to consider those different models. Furthermore, we introduced

think-aloud protocols, maintenance tasks, mental simulation, static tasks, and subjective rating as techniques to measure program comprehension.

Having defined our variables of interest, we can start to design experiments. We introduce necessary terms for conducting experiments in the next section. Readers familiar with conducting experiments may skip this section.

2.3 Conducting experiments

In this section, we introduce basic terms regarding experiments. We regard this as necessary, because this work was developed at the School of Computer Science at the University of Magdeburg, where conducting experiments is not part of the curriculum. However, readers familiar with conducting experiments may skip this section or refer to the summary at the end.

An experiment can be described as a systematic research study in which the investigator directly and intentionally varies one or more factors while holding everything else constant and observes the results of the systematic variation (WUNDT [Wun14], WOODWORTH [Woo39]).

From this definition, three criteria for experiments can be derived (WUNDT [Wun14]). Firstly, experiments must be designed in a way that other researchers can replicate the experiment. This way, researchers can control each other. Secondly, the variations of the factors must be intended by the investigator. This is necessary to allow replications, because random variations cannot be replicated. Thirdly, it is important that factors can be varied. Otherwise, an effect on the result cannot be observed depending on the variations of the factors. Although those criteria can be deduced from the definition, it is important to state them explicitly, because every good experiment should fulfill them in order to build a body of knowledge with the help of experimental research (WUNDT [Wun14]).

In order to conduct experiments that fulfill the above introduced criteria, the process of conducting experiments is divided into five stages (JURISTO AND MORENO [JM01], p. 49). Firstly, the variables and hypotheses of the experiment need to be specified. Secondly, a detailed plan for conducting the experiment needs to be developed. Thirdly, the experiment needs to be run according to the developed plan. Fourthly, the data collected during the execution needs to be analyzed. Finally, the results need to be interpreted and their meaning for the hypotheses evaluated. For better overview, we visualize this process in Figure 2.9.

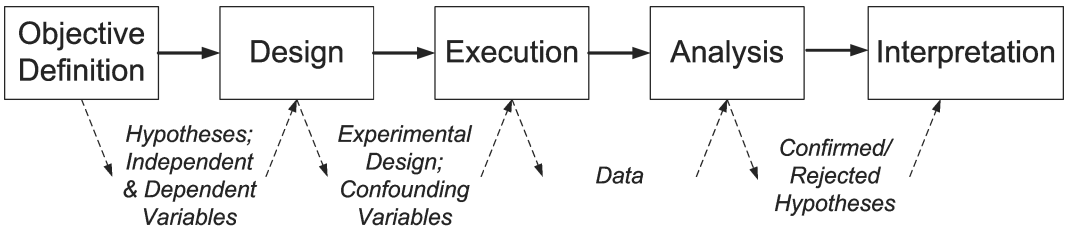


Figure 2.9: Stages of an experiment.

In this section, we traverse these stages and introduce relevant terms for each stage. We apply the information to our objective: assessing the understandability of FOSD approaches.

We start with objective definition in the next section. Then, we discuss the design stage in Section 2.3.2 and methods for analyzing the results in Section 2.3.3. We conclude with interpreting the results in Section 2.3.4. We omit the execution stage, because we do not need to introduce specific terms.

2.3.1 Objective definition

During objective definition, the variables of interest are defined (JURISTO AND MORENO [JM01], p. 49). After defining the variables, we need to develop hypotheses about the relationship between our variables. This is important so that we can choose a suitable experimental design and analysis methods. If we developed our hypotheses after we ran the experiment, we might discover that our material, tasks, or subjects were not suitable to test our hypotheses, leaving us with useless data. The process of defining variables and hypotheses is also referred to as *operationalization*, because a set of operations is defined with which we measure our variables and test our hypotheses (BRIDGMAN [Bri27]).

In this section, we introduce independent and dependent variables in Sections 2.3.1.1 and 2.3.1.2, respectively. Then, we explain how hypotheses for experiments should be stated in Section 2.3.1.3.

2.3.1.1 Independent variable

An *independent variable* is intentionally varied by the experimenter and influences the outcome of an experiment (JURISTO AND MORENO [JM01], p. 60). It is also referred to as *predictor (variable)* or *factor*. Each independent variable has at least two *levels* or *alternatives*.

In our case, the independent variable is the set of FOSD approaches. The levels of our independent variable are the different approaches we presented in Chapter 2. For example, if we would conduct an experiment regarding understandability of AOP, FOP, CPP, and CIDE, our independent variable had four levels. We would evaluate the effect of each approach on program comprehension by varying the approach. Any variations in program comprehension can then be attributed to variations of the FOSD approach.

2.3.1.2 Dependent variable

A *dependent variable* is the outcome of an experiment (JURISTO AND MORENO [JM01], p. 59). It depends on variations of the independent variable. It is also referred to as *response variable*.

The dependent variable in this thesis is program comprehension, because we want to examine its variation depending on variations of the FOSD approach.

2.3.1.3 Hypotheses

Having decided on the variables of interest and their measurement, the hypotheses that should be evaluated with the experiment need to be specified. It is important to specify the hypotheses during the objective definition for several reasons: Firstly, they influence the decisions of the remaining stages of the experiment, e.g., the subjects we recruit and the analysis methods we need to apply (BORTZ [Bor00], [p. 2]). Secondly, since we define beforehand what we want to answer with our experiment, we prevent us from fishing for results in our data and thus discovering random relationships between variables (EASTERBROOK ET AL. [ESSD08]).

One important criterion for a scientific hypothesis is its falsifiability (POPPER [Pop59]). This means that a hypothesis should be specified so that it can be empirically tested and rejected, if it is wrong. Hypotheses that are continually resistant to be falsified are assumed to be true, yet it is not proven that they are. The only claim we can make is that we found *no evidence to reject* our hypotheses. In order to be falsifiable, a hypothesis needs to be as clear and specific as possible. The variables need to be defined exactly, as well as the expected relationships between them. Otherwise, the same result can be used to reject and falsify a hypothesis, depending on the specification of variables and their relationship.

Example. In order to make this clearer, we give an example hypothesis and refine it in a stepwise manner. A statement like:

- Some FOSD approaches provide more benefit on program comprehension than others.

is not a suitable hypothesis, because it is not specific enough. The first problem is the word *some*, because it does not say which or how many. Hence, if our results would be that three, four, or five approaches are better, then all those results either verify or falsify our hypothesis, depending on the definition of *some*. In addition, we did not specify which approaches are more understandable. Thus, a result showing that AOP is better understood than FOP can be used for verifying and falsifying our result. The first refinement could be:

- FOP provides more benefit on program comprehension than AOP.

because now we specified the approaches.

Secondly, we take a closer look at *program comprehension*. In section 2.3.1.2, we introduced several models for program comprehension, which are applied by programmers depending on their domain knowledge. We also stated that top-down comprehension is more efficient than bottom-up comprehension. Hence, we need to specify the model we use or explicitly state that we refer to all models at once. Our refined hypothesis could be:

- FOP provides more benefit on bottom-up program comprehension than AOP.

Thirdly, we need to clarify *provide more benefit*. This means that we have to specify how we measure program comprehension. Otherwise, if we obtained the result that there are no differences in response times, but in correctness of the answers, we could interpret them both in favor and against our hypothesis. Thus, we have to refine our hypothesis further:

- With FOP compared to AOP, subjects are faster and make fewer errors while maintaining a program of an unfamiliar domain (forcing them to understand a program bottom up).

This hypothesis can be tested, because we specified the levels of our independent variable, defined our dependent variable, and specified the technique and measure that we use to assess program comprehension. We could refine our hypothesis further, e.g., by stating that subjects are at least five minutes faster or make at least three error less. However, in this case, we need to have some preliminary results about the magnitude of the effects, which we could obtain by literature research or prior experiments.

With defining our variables of interest and stating our hypotheses, we finished defining our objective. The next step is to design our experiment so that we can test our hypotheses.

2.3.2 Design

Based on our independent and dependent variables as well as the hypotheses, we can design a plan for running our experiment. Our plan must ensure that our experiment is valid (2.3.2.1), which requires that we eliminate the influence of all variables we are not interested in (2.3.2.2). This includes choosing an appropriate experimental design (Section 2.3.2.3).

2.3.2.1 Validity

There are two kinds of validity we are concerned with in this thesis: internal and external validity. Further kinds of validity include construct, discriminant, convergent, and ecological validity, but their importance rather lies in the development of tests (e.g., intelligence tests) than for conducting experiments (MOOSBRUGGER AND KELAVA [MK07]).

Internal validity describes the degree to which the value of the dependent variable can be assigned to the manipulation of the independent variable (SHADISH ET AL. [SCC02]). In order to assure that the result can be attributed to the independent variable, confounding factors need to be controlled, because they also influence our dependent variable. Examples of confounding parameters are noise and light conditions during the experiment, experience of subjects, etc. Without controlling the influence of confounding factors, we cannot be sure that our result can solely be attributed to our variation of the independent variable. As a consequence, we can neither reject nor confirm our hypotheses, since other parameters that are not included in our hypotheses could as well influence our result.

The degree to which the results gained in one experiment can be generalized to other subjects and settings is referred to as *external validity* (SHADISH ET AL. [SCC02]). The more realistic an experimental setting is, the higher its external validity is. Hence, we could conduct our experiment in a company under real working conditions with employees of the company. Now, however, our internal validity is threatened, because we have no influence on the light and noise level and the experience of our subjects.

How do we deal with this dilemma? Either we can attribute the variations in program comprehension to the FOSD approach, or we can generalize our result. One way is to conduct

both types of experiments (SHADISH ET AL. [SCC02]): First, if we have no knowledge about how an FOSD approach affects program comprehension, we design an experiment with high internal validity, allowing us to examine the influence of FOSD on program comprehension under controlled conditions. Second, if we have gained knowledge about the influence, we can design experiments with lower internal, but high external validity, which allows us to test our knowledge under real conditions and thus generalizing our results.

2.3.2.2 Controlling confounding parameters

In order to have a high degree of internal validity, we need to control the influence of confounding parameters. A *confounding parameter* is a variable that influences the depending variable besides variations of an independent variable (GOODWIN [Goo99], p. 143). Examples of confounding parameters for program comprehension are programming experience, intelligence, or programming language.

Since there is a large number of confounding parameters on program comprehension, we discuss all of them not in this section, but in Chapter 3. Here, we describe several methods for controlling the influence of confounding parameters.

Several techniques were developed to control the influence of confounding parameters, which we explain in this section:

- Randomization (GOODWIN [Goo99])
- Matching (GOODWIN [Goo99])
- Keep confounding parameter constant (MARKGRAF ET AL. [MMW+01])
- Use confounding parameter as independent variable (MARKGRAF ET AL. [MMW+01])
- Analyze influence of confounding parameter on result (SHADISH ET AL. [SCC02], p. 62)

Since it is hard to explain the techniques in an abstract way, we use programming experience as example parameter in this section. A detailed description of programming experience can be found in the next chapter.

Randomization

The first technique is called *randomization*. It can be applied when our *sample*, i.e., the set of persons that take part in our experiment (ANDERSON AND FINN [AF96], p. 18), have to be split in two or more groups. With this technique, we randomly assign our *subjects* or *experimental unit* (JURISTO AND MORENO [JM01]) to the experimental groups. Due to randomization, it assumed that we create homogeneous groups according to programming experience, which means that our groups do not differ in their programming experience.

This assumptions is based on the fact that statistical errors even out with a large amount of replication (BORTZ [Bor00], p. 8). For example, if a coin is flipped four times, chances are that we receive 'heads' four times. However, flipping a coin hundred times, the error evens out, so that the results 'heads' and 'tails' occur about fifty times each. Hence, the statistical error only evens out, if our sample is large enough.

Unfortunately, there is no specification on what large means – at least to the best of our knowledge. In BORTZ [Bor00], p. 103, a large sample is defined as at least 30 subjects. However, if our independent variable had five levels, we would have six subjects per level, which is defined as small in other references (e.g., GOODWIN [Goo99], p. 174). Hence, we cannot give an advice on how large our sample should be to be able to apply randomization. The only advice we can give is that if we want to apply randomization, the sample should be as large as possible. Otherwise, we need other group assignment techniques.

Matching

For the case that our sample is too small, *matching* was developed. This process requires that the confounding variable is measured. Then, subjects are ranked according to the measure and assigned to two groups, for example stratifying the sample (SHADISH ET AL. [SCC02]): The first subject is assigned to the first group. The second and third subjects are assigned to the second group, and the fourth subject is again assigned to the first group. This is continued, until all subjects are assigned. Another group assignment technique creates subset of subjects based on the ranking (e.g., four subsequent subjects are assigned to one subset) and then randomly assigns the subjects of each subset to one of the experimental groups. Both techniques assure that both experimental groups are homogeneous regarding programming experience.

There are further randomization variants for generating experimental groups, e.g., stratified, cluster, or staged sampling, which can be applied depending on the hypotheses as well as human and financial resources of the experiment (ANDERSON AND FINN [AF96], p. 639).

Keep confounding parameter constant

Another way to deal with a confounding parameter is to keep it constant in our sample. We can accomplish this by selecting only subjects with a certain level in the confounding parameter. This requires a pretest, so that all persons not matching this criterion can be excluded beforehand. As a shortcoming, more effort in acquiring subjects is necessary, because we exclude several individuals and thus need to contact more in order to create our sample. Furthermore, the population of the sample is diminished to persons with the level of the confounding parameter we used for our experiment, reducing the external validity of our experiment. In addition, the selection of subjects is more prone to bias than usual, because we use a stricter selection criterion (MARKGRAF ET AL. [MMW+01]).

Use confounding parameter as independent variable

A further way is to declare a confounding parameter as independent variable in our objective definition. In this case, we have to adapt our hypotheses. Since we already have one independent variable, the FOSD approach, we now have to vary two independent variables. If we chose to use two levels for both independent variables, we had to consider every combination of those levels, resulting in four conditions to test.

In addition to more effort in varying the independent variable, the analysis of the data would also get more complicated. Without programming experience as independent variable, we could use a simple t test to evaluate the difference (STUDENT [Stu08]). However, with two independent variables, we need to conduct a two-factorial ANOVA (ANDERSON AND FINN [AF96]), thus consider two main effects and analyze possible interactions between our two independent variables.

Analyze influence of confounding parameter on result

If we have to deal with a fixed sample, e.g., from a company or a programming course at the university, we cannot randomize our sample. In this case, programming experience can be measured and its influence on program comprehension can be analyzed afterwards. This way, we can evaluate the influence on programming experience on our results and include it in our interpretation (SHADISH ET AL. [SCC02]).

So, which of these techniques should be applied? Unfortunately, there is no general answer to this question. It depends on the hypotheses of our experiment as well as the human and financial resources. In Chapter 3, we discuss for each parameter how the presented techniques can be applied and give some examples under which circumstances which technique to apply. In the next section, we present some typical experimental designs that can be applied to control typical confounding parameters that occur in nearly all experiments.

2.3.2.3 Experimental designs

Experimental designs define how levels of the independent variables are applied to subjects (JURISTO AND MORENO [JM01]). Usually, designs are grouped into one-factorial (one independent variable) and two-factorial (two independent variables) designs. It is possible to have three and more factorial designs, however they are time consuming and costly, because they require a large sample and specific analysis methods. First, we describe some one-factorial designs. Then, we present a two-factorial design to show how complexity increases with two independent variables instead of one.

One-factorial designs. In one-factorial designs, we have one independent variable with two or more levels. We have to decide what we do with our sample: We could apply all levels to all subjects or split our sample into groups according to the number of levels of our independent

FOSD	Group
AOP	1
FOP	2

Table 2.2: Simple one-factorial design.

Group	Trial 1	Trial 2
1	AOP	FOP
2	FOP	AOP

Table 2.3: One-factorial design with repeated measures.

variable (GOODWIN [Goo99], p. 237). A simple design is shown in Table 2.2 with AOP and FOP as example levels. In this case, we have to make sure that both groups are comparable, e.g., that they are homogeneous according to program comprehension. Furthermore, we cannot exclude that the order of the application of the approaches influences program comprehension (referred to as ordering effect). For example, subjects could comprehend an FOP program better, because they do not need to get used to it, whereas AOP needs some time to get used to, but then outperforms FOP.

In order to avoid the problems, we extend this design as shown in Table 2.3, which is referred to as repeated measures. We conduct two trials, i.e., a further the measurement, but inversing the levels of our independent variable. This way, we can check for ordering effects and have a more accurate measurement of program comprehension, because we have two measurements instead of one (cf. Table 2.2). The problem with a repeated measure is that test effects can occur, i.e., that subjects learn from their first program. In order to avoid test effects, we either can create two different tasks for each level (and make sure that they are comparable) or let a sufficient time interval pass between both trials. The greater the time interval is, the smaller test effects are, but the higher *mortality* is, i.e., that subjects drop out (SHADISH ET AL. [SCC02], p. 59).

One-factorial designs can also be applied for more than two levels of an independent variable, however the analysis methods are different in those cases (BORTZ [Bor00]). Depending on the hypotheses and resources of the experiment, one of those designs can be chosen. In our experiment, which we describe in Chapter 5, we chose the simple one-factorial design because of our limited resources.

Two-factorial designs. In order to show how complexity increases if we choose to include a confounding parameter as independent variable, we show a two-factorial design. An example with programming experience is shown in Table 2.4. This design looks similar to the one in Table 2.3, however in the current design, we have four groups and still do not control the order effect (which we could do by conducting a second trial with a different order). In addition to the more complicated design, the analysis methods are more complicated, too (BORTZ

	AOP	FOP
little programming experience	group 1	group 2
much programming experience	group 3	group 4

Table 2.4: Two-factorial design.

[Bor00]).

Carefully designing experiments is crucial for drawing sound conclusions from our data. It helps to decide whether our data are valid and thus the evaluation of our hypotheses is unbiased. A further step is to choose the right analysis method for our data, which we explain next. We omit the execution stage, because no psychological concepts or terms need to be explained.

2.3.3 Analysis

Having defined the objectives and design for our experiment, we can collect and analyze the data. During analysis, several methods can be applied. First, we explain scale types of variable in Section 2.3.3.1, because the decision for an analysis method for a variable depends on its scale type. Then, we present some descriptive statistics for describing our data and sample in Section 2.3.3.2. Finally, we introduce significance tests in Section 2.3.3.3, which can be applied to test our hypotheses.

2.3.3.1 Scale types

In order to be able to choose the right analysis method, we have to know the *scale type* of a variable. There are four common scale types: nominal, ordinal, interval, and ratio (FENTON AND PFLEEGER [FPG94]).

Nominal scales are classification of values. A typical example is the coding of gender with numbers, e.g., male with 0 and female with 1. The numbers have no qualitative or quantitative meaning.

Ordinal scales are used to describe a ranking. An example is the order of subjects according to their response time. The ranks have no quantitative meaning, but indicate the order of subjects.

On *interval* scales, numbers have a quantitative meaning. The difference between two consecutive numbers is constant for each pair of numbers. There is no absolute zero, in contrast to *ratio* scales, which include an absolute zero. Both scales are often referred to as *metric*. It is important to know the scale type of a variable, because most analysis methods require a certain scale type. To clarify scale types, we present an overview in Table 2.5, in which we show allowed operations for each scale type (BORTZ [Bor00], p. 23).

	Computations			
Scale type	Frequencies	Ordering	Differences	Quotient
Nominal	yes	no	no	no
Ordinal	yes	yes	no	no
Interval	yes	yes	yes	no
Ratio	yes	yes	yes	yes

Table 2.5: Scale types and operations (adapted from BORTZ [Bor00], p. 23).

2.3.3.2 Descriptive statistic

Descriptive statistics are important to understand our data and support other researchers in replicating our experiments.

Central tendency

If we have a series of measure for our subjects, we look for a value that describes the distribution of our measures best (BORTZ [Bor00], p. 35). Depending on the scale type of our data, different measures can be used, which are referred to as measures for *central tendency*. The two most frequent measures are the median and the arithmetic mean.

The *median* splits a frequency distribution in half. The sum of the absolute values of the deviation from the median is minimal. For example, if we have the ordered list of numbers 1, 5, 5, 6, 10, the median of those numbers is 5, because it lies in the middle of this list. It requires that data are at least ordinal scaled.

Second, there is the *arithmetic mean*, which divides the sum of all values by the number of values and assumes that data have at least interval scale type:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Further values include mode, geometric mean, and harmonic mean, which are sometimes more suitable to describe the data. An overview including application and requirement for those further measures can be found in (BORTZ [Bor00], p. 38).

Dispersion

While different distributions can have similar central tendency, they can still look very different. For example, compare the series 99, 100, 101 with the series 0, 100, 200. Both have the same median and arithmetic mean, yet their distributions are rather different. Hence, central tendency measures are not sufficient to describe our data, but we need information about the distribution. Two commonly used measures are variance and standard deviation, which both require the data to be metric.

Variance describes the sum of squared deviations from the arithmetic mean divided by the number of all measures:

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}$$

The deviations have to be squared, because otherwise the sum would add up to zero. Furthermore, the larger the deviation, the higher the influence of this value on the variance is.

One problem is that the underlying unit of our data (e.g., second for response time) is also squared, which makes it hard to interpret (BORTZ [Bor00], p. 41). Hence, the square root of the variance is often used, which is called *standard deviation (s)*. For interpreting the standard deviation, it is necessary to know that between $\overline{x} - s$ and $\overline{x} + s$, 68% of the values can be found, assumed that our data are normally distributed (i.e., they can be described with $f(x) = \frac{1}{\sqrt{2\pi \cdot s^2}} \cdot e^{-\frac{x-\overline{x}^2}{2s^2}}$). Hence, the larger the standard deviation, the more widely spread our values are. This also counts for not normally distributed data, but the probability that a value deviates from the mean by one standard deviation depends on the underlying distribution (BORTZ [Bor00], p. 43).

If our data are not metric, we have to use other measures to describe dispersion, for example the *interquartile range*, which is the difference between the first and third quartiles (ANDERSON AND FINN [AF96], p. 110). A *quartile* divides a series of measurement in to quarters (ANDERSON AND FINN [AF96], p. 75) (similar to the median, which divides a series of measurement in halves).

Further measures for dispersion are series range and mean deviation (BORTZ [Bor00], p. 40). Next, we show a commonly used technique to visualize central tendency and dispersion of our data, which does not require a metric scale type of our data.

Box plots

Box plots were developed in 1977 (TUKEY [Tuk77]). A *box plot* displays the median, extremes, and interquartile range of a series of measurement. In Figure 2.10, an example is shown. The median and interquartile range are depicted as box, which contains 50% of the data, whereas all other values are depicted as 'whiskers'. Instead of depicting the minimum and maximum observed value, the length of the whiskers can be defined depending on the interquartile range (e.g., one and a half times) (ANDERSON AND FINN [AF96], p. 126). This allows us to identify outliers in the data, because they are depicted as extra data points and not as part of the whiskers.

From boxplots, the following information about the data can be derived: Firstly, the median as measure for central tendency is depicted by the line that splits the box in halves. Secondly, the position of the median provides information about the *skewness* of the data: The more unequal the halves of the box are, the more our data are skewed, which gives us a hint that our data might not be normally distributed. Additionally, the whiskers also provide information about skewness: The longer one whisker is compared to the other, the more the data are skewed in direction of the longer whisker. Thirdly, dispersion of our data can be derived

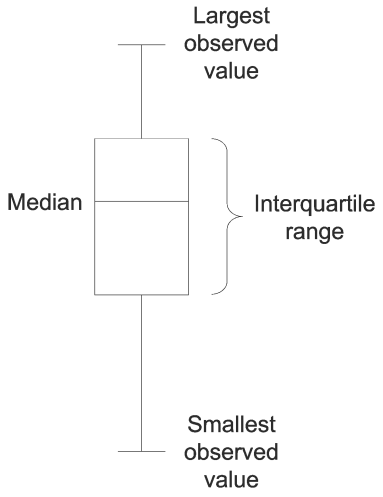


Figure 2.10: Box plot.

from the interquartile range, because the larger the range is, the higher the dispersion of our data is (BORTZ [Bor00], p. 40).

Having described our data sufficiently, we can start to test our hypotheses with significance tests, which we explain next.

2.3.3.3 Significance tests

In order to test our hypotheses, we need to transform them into statistical hypotheses. Furthermore, we have to know the errors we can make in statistical decisions. Then, we need to know which significance test we can apply depending on the scale type and distribution of our data. We explain some significance tests in detail, because understanding their mechanism helps to understand why hypotheses are confirmed or rejected. Readers already familiar with the mechanism of significance tests may refer to Figure 2.11 on page 43, in which we present an overview of significance tests and their application.

Statistical hypotheses

In order to be able to test our hypotheses, we have to transform them into *statistical hypotheses*, because significance tests test those statistical hypotheses. We have to make sure that our hypotheses are represented as accurately as possible by one or more statistical hypotheses.

There are two kinds of statistical hypotheses: alternative hypotheses and null hypothesis.

An *alternative hypothesis*, also called H_1, states that there is a difference between values. For example, we transform our hypothesis of Section 2.3.1.3

- With FOP compared to AOP, subjects are faster and make fewer errors while maintain-

	H_0 is valid	H_1 is valid
Decision for H_0	correct	type 2 error
Decision for H_1	type 1 error	correct

Table 2.6: Cases in statistical decisions.

ing a program of an unfamiliar domain (forcing them to understand a program bottom up).

into several alternative hypotheses. We split our two measures for program comprehension into two hypotheses (we omit several specifications of our hypotheses to concentrate on relevant aspects):

- H_{11}: Subjects are faster with an FOP program.
- H_{12}: Subjects make fewer errors with an FOP program.

In contrast to H_1, the *null hypothesis*, also called H_0, states that there is no difference. For every alternative hypothesis, an according null hypothesis can be stated. For our example, they are

- H_{01}: Subjects with an FOP program are as fast as subjects with an AOP program.
- H_{02}: Subjects with an FOP program make the same amount of errors as subjects with an AOP program.

Next, we discuss the errors we can make when we reject or confirm H_0 or H_1.

Kinds of errors

With statistical decisions, four cases can occur:

The first error we can make is to reject H_0, although it is true. This is also referred to as α *error* or *type 1 error*. The second kind of error, called β *error* or *type 2 error*, is to reject H_1 although it is true, or in other words, confirm H_0, although it is false.

Which of the errors is easier to live with? Both lead to problems: When a type 1 error is made, we incorrectly assume that our null hypothesis is wrong and that the variation of our independent variable indeed has an effect on our dependent variable. If we took consequences from our result and started eliminating the FOSD approach that negatively affects program comprehension according to our result, this effort would be wasted, because it actually makes no difference for comprehension how a program is developed.

On the other hand, if we make a type 2 error, we incorrectly assume that our alternative hypothesis is wrong and that the variation of our independent variable has no effect. As a

consequence, we could support both FOSD approaches we tested in our experiment, because it makes no difference on program comprehension according to our result. However, since program comprehension actually is influenced by the FOSD approach, we would support an approach that negatively affects program comprehension, which could lead to higher maintaining costs than necessary (because comprehending a program is required for maintaining it).

Depending on the possible consequences of our result, we have to decide which error is more acceptable. In order to provide comparability and quality of statistical decisions, the probability of the type 1 error has become a standard for making statistical decisions (BORTZ [Bor00], p. 113). The probability of the type 1 error is referred to as *significance level* or α. It denotes the probability that our outcome occurred under the assumption that H_0 is valid. Hence, it is a conditional probability.

If the probability that an observed result under the assumption that H_0 is valid is smaller than 5%, a result is called *significant*, if it is smaller than 1%, *very significant*. In that case, we reject our null hypothesis and assume that our alternative hypothesis is true.

Having introduced relevant terms for significance tests, we discuss some frequently used significance test depending on the scale type of our data. Generally, significance tests compute a test statistic and compare it with a theoretical value, i.e., one from an underlying distribution function (depending on the concrete significance test). The theoretical value of the function depends on the significance level. We start by introducing the χ^2 test in the next section.

Nominal scale

Data with nominal scale type allow only to compare frequencies (cf. Table 2.5). The most commonly used test for this case is the χ^2 test, which checks whether observed frequencies significantly deviate from the expected frequencies. The underlying distribution function is the χ^2 distribution (BORTZ [Bor00], p. 817).

In order to apply the χ^2 test, some requirements need to be fulfilled (BORTZ [Bor00], p. 177):

- The observations must be independent from each other.
- Every experimental unit must be unambiguously assigned to one level of the independent variable.
- The expected frequencies should not be smaller than ten.

We have to make sure that those requirements are fulfilled before we conduct the test. Otherwise, the χ^2 test could incorrectly confirm or reject the null hypothesis. In case one of the requirements is not met, we cannot apply the χ^2 test, but have to adjust it or use other tests (BORTZ [Bor00], p. 177)

For the case that our independent variable has two levels (e.g., male and female), the expected frequencies are calculated according to:

	AOP	FOP
Observed	7	14
Expected	10.5	10.5

Table 2.7: Example of observed and expected frequencies.

$$f_{e(1)} = f_{e(2)} = \frac{f_{o(1)} + f_{o(2)}}{2}$$

where $f_{e(1/2)}$ denotes the expected frequencies and $f_{o(1/2)}$ the observed frequencies. Based on both frequencies, the χ^2 value is calculated according to

$$\chi^2 = \sum_{j=1}^{2} \frac{(f_{o(j)} - f_{e(j)})^2}{f_{e(j)}}$$

where j describe the levels of our independent variable.

This calculated value is compared with a value of the χ^2 distribution, which depends on the specified significance level and the *degrees of freedom (df)*. The df is one less than the number of categories of our variable we are analyzing. Hence, it is 1 in our case, since our dependent variable has two categories (e.g., male and female). If we specify our significance level with 5%, the according χ^2 value from the distribution is $\chi^2_{(\alpha=.05;df=1)} = 3.84$ (BORTZ [Bor00], p. 817). If our calculated value is larger than the value from the distribution, we reject our null hypothesis, because the probability that our result occurred if H_0 is valid is smaller than 5%.

Example. In order to clarify the mechanism of the χ^2 test, we use an example. We assume that we conducted an experiment with forty subjects, whereas twenty got an AOP program, the other twenty an FOP program. Our H_0 states that there are no differences in the number of errors between both programs. In Table 2.7, we show the observed number of errors (row *Observed*) and the expected number of errors (row *Expected*), which are calculated according to

$$f_{e(AOP)} = f_{e(FOP)} = \frac{f_{o(AOP)} + f_{o(FOP)}}{2} = \frac{7 + 14}{2} = 10.5$$

If we take a first look on the observed frequencies, we can see that they differ between the AOP and FOP group. However, is this difference statistically significant or random? In order to answer this question, we compute the χ^2 value:

$$\chi^2 = \sum_{j=1}^{2} \frac{(f_{b(j)} - f_{e(j)})^2}{f_{e(j)}} = \frac{(7 - 10.5)^2}{10.5} + \frac{(14 - 10.5)^2}{10.5} = 2.33$$

If we compare our observed value, $\chi^2_{\alpha=.05;df=1} = 1.32$ with the χ^2 value of the χ^2 distri-

bution, $\chi^2_{\alpha=.05;df=1} = 3.84$, we can see that the observed value is smaller than the theoretical value. Hence, we confirm our H_0, which means that we can conclude that the difference in our sample occurred randomly.

As this example showed, although a difference in the number of errors occurred in our sample, it most probably occurred randomly. This demonstrated the necessity of significance tests.

For more than two levels for our independent variable and/or more than one independent variable, variants of the χ^2 test also exist (ANDERSON AND FINN [AF96]).

In the next section, we introduce the Mann-Whitney-U test, one representative significance test for ordinal scaled data.

Ordinal

For ordinal data, the Mann-Whitney-U test was developed to compare central tendency of our data (BORTZ [Bor00], p. 150). It compares the rankings of two groups according to a characteristic and checks whether they significantly differ.

The application of the Mann-Whitney-U test requires that the data have ordinal scale and that both samples are independent. For dependent samples (e.g., if we recruit couples for our experiment and assign the female to one group and the according male to the other), we need to apply other tests (e.g., the Wilcoxon test (BORTZ [Bor00], p. 153)).

The first step is to order our complete sample according to a variable we measured (e.g., time to finish a task). Then, we compute the sum of ranks for each group by adding the ranks of all subjects per group:

$$T = \sum_{i=1}^{n} r_i$$

where n is the number of subjects in the group and r the rank of a subject.

Then, we compute U according to:

$$U = n_1 \cdot n_2 + \frac{n_1 \cdot (n_1 + 1)}{2} - T_1$$

where n_1 and n_2 denotes the number of subjects per group. Since the U values are symmetrically distributed around their mean, it suffices to use either T_1 or T_2 (BORTZ [Bor00], p. 151). The next step is to determine the U value with which we compare our computed value by using according tables (e.g., BORTZ [Bor00], p. 826). The U value depends on the sizes of both groups, for example, if we have ten subjects per group, the according U value is 23.

Example. As example, assume we collected response times in an experiment. We assume we had ten subjects per level of our independent variable (e.g., AOP and FOP), with the response times shown in Table 2.8. The table contains the response time in seconds, as well as the rank of the response time in our complete sample. For example, 84 seconds is the third fastest

reaction time, after 78 and 81.

AOP		FOP	
Time (s)	Rank	Time (s)	Rank
85	4	96	8
106	13	105	12
118	17	104	11
81	2	108	15
138	20	86	5
90	6	84	3
112	16	99	9
119	18	101	10
107	14	78	1
95	7	124	19
Rank sums:	T_1 = 117		T_2 = 93

Table 2.8: Example response times for Mann-Whitney-U test.

Now, if one group is slower than the other, the sum of ranks must be larger in that group. We can see that for the AOP group the sum of rank is 117 and for the FOP group 93. In order to check if this difference is significant, we compute the test statistic, U:

$$U = n_1 \cdot n_2 + \frac{n_1 \cdot (n_1 + 1)}{2} - T_1 = 10 \cdot 10 + \frac{10 \cdot (10 + 1)}{2} - 117 = 38$$

and compare this value with the theoretical value, which is in our case $U = 23$. Since our computed value $U = 38$ is larger than the expected value $U = 23$, we reject H_0 that the response times do not differ and confirm H_1 that there is a difference. Hence, we can assume that the observed difference did not occur randomly.

Metric

If we want to compare the arithmetic means of metric data, we apply the t test (ANDERSON AND FINN [AF96], p. 468). It tests whether the means of two groups significantly differ. The null hypothesis for the t test is that there is no difference between both means.

The application of the t test requires that our data are normally distributed (except if our sample is larger than 50 subjects) and that both samples are independent. We can check whether our data are normally distributed with several tests, for example a Shapiro-Wilk test (SHAPIRO AND WILK [SW65]). If a normal distribution cannot be confirmed and our sample is too small, we cannot apply the t test, but have to use other tests (e.g., the Mann-Whitney-U test). If we have dependent samples, we need to apply a t test for dependent samples (or a Wilcoxon test, if our sample is too small and the data are not normally distributed).

As for the other significance tests, we compute a test statistics, in this case a t value, and

compare our computed value with the theoretical value of a t distribution. The t value is calculated according to:

$$t = \frac{\overline{x_1} - \overline{x_2}}{\hat{\sigma}_{(\overline{x_1}-\overline{x_2})}}$$

where

$$\hat{\sigma}_{(\overline{x_1}-\overline{x_2})} = \sqrt{\frac{\sum_{i=1}^{n_1}(x_{i1} - \overline{x_1})^2 + \sum_{i=1}^{n_2}(x_{2i} - \overline{x_2})^2}{(n_1 - 1) + (n_2 - 1)}} \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

Like for the χ^2 test, we need to compute the df for our t test, in this case according to $df = (n_1 - 1) + (n_2 - 1)$.

Example. As example, we use the reaction times in Table 2.8. The arithmetic means are $\overline{x}_{AOP} = 105.1$ and $\overline{x}_{FOP} = 98.5$. We assume that the response times are normally distributed, so that we can apply the t test. Now, we can compute $\hat{\sigma}_{(\overline{x_1}-\overline{x_2})}$ according to:

$$\hat{\sigma}_{(\overline{x_1}-\overline{x_2})} = \sqrt{\frac{\sum_{i=1}^{n_1}(x_{i1} - \overline{x_1})^2 + \sum_{i=1}^{n_2}(x_{2i} - \overline{x_2})^2}{(n_1 - 1) + (n_2 - 1)}} \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}} =$$

$$\sqrt{\frac{(85 - 105.1)^2 + (106 - 105.1)^2 + ... + (96 - 98.5)^2 + (105 - 98.5)^2 + ...}{(10 - 1) + (10 - 1)}}$$

$$\cdot\sqrt{\frac{1}{10} + \frac{1}{10}} = 7.05$$

The t value is :

$$t = \frac{\overline{x_1} - \overline{x_2}}{\hat{\sigma}_{(\overline{x_1}-\overline{x_2})}} = \frac{105.1 - 98.5}{7.05} = 0,94$$

Now, we compare our computed value with the according value of the t-distribution, $t_{\alpha=.05,df=18} = 1.73$. Since our computed value is smaller, we confirm our null hypothesis that there are no differences in response times.

This result may sound surprising, because with the Mann-Whitney-U test, we confirmed a difference on the same data. However, since we did not confirm that our response times are normally distributed, we should not have applied the t test in the first place. We did it anyway, because we could show the necessity of confirming the requirements for the application of a significance test. Furthermore, the underlying mechanism of the t test should be demonstrated, for which it is not necessary to test the requirements.

The three significance tests we introduced (i.e., χ^2, Mann-Whitney-U, and t test) usually suffice to reject or confirm most hypotheses. In more complex design or hypotheses, however,

they cannot be applied, but more sophisticated techniques have to be used. In the next section, we give an overview of some further tests.

More sophisticated techniques

We shortly discuss three further groups of tests to evaluate hypotheses, to give a more complete, but still not exhaustive overview.

One-way ANOVA. What if we want to compare the means of an independent variable with more than two levels? Of course, we could compare the levels pairwise with a t test. However, the type one error cumulates if we conduct more than one test (BORTZ [Bor00], p. 127). Hence, we have to use other methods, for example an ANOVA (analysis of variances), which splits the variances of our sample into several components and uses them for computing the test statistics (ANDERSON AND FINN [AF96]).

Two-way ANOVA. If we have two independent variables in our experiment, we have to use a two-way ANOVA to analyze our data. We can check whether *main effects* occur, i.e., whether each of our independent variables has an effect. In addition, we can look for an *interaction* between our independent variables, i.e., whether a certain combination of levels affects our dependent variable (BORTZ [Bor00], p. 289).

Factorial & cluster analysis. Factorial and cluster analysis are explorative, which means that they are used to look for relationships in a large number of variables. The factorial analysis tries to find latent factors that explain correlations between several variables. The cluster analysis also looks for correlations, but groups experimental units, not variables. Hence, if want to explore relationships between different variables, we can use one of those methods (BORTZ [Bor00]).

In Figure 2.11, we present an overview of analysis methods and when they can be applied.[8]

2.3.4 Interpretation

Since for interpreting results, no psychological terms are necessary, the discussion is rather short. Applying inference tests to evaluate statistical hypotheses is an instrument to analyze the data. Once we have confirmed or rejected our statistical hypotheses, we have to draw conclusions for our hypotheses stated during objective definition. Furthermore, if we have obtained unexpected results, we have to search for possible explanations. In this case, exploring our data set for supporting our explanations is a valid approach. Additionally, searching for similar results in the literature could help explain our outcome. In any case, those unexpected results must be elaborated further, depending on their importance for future experiments.

[8] http://www.vislab.ch/Lehre/EST/est.html

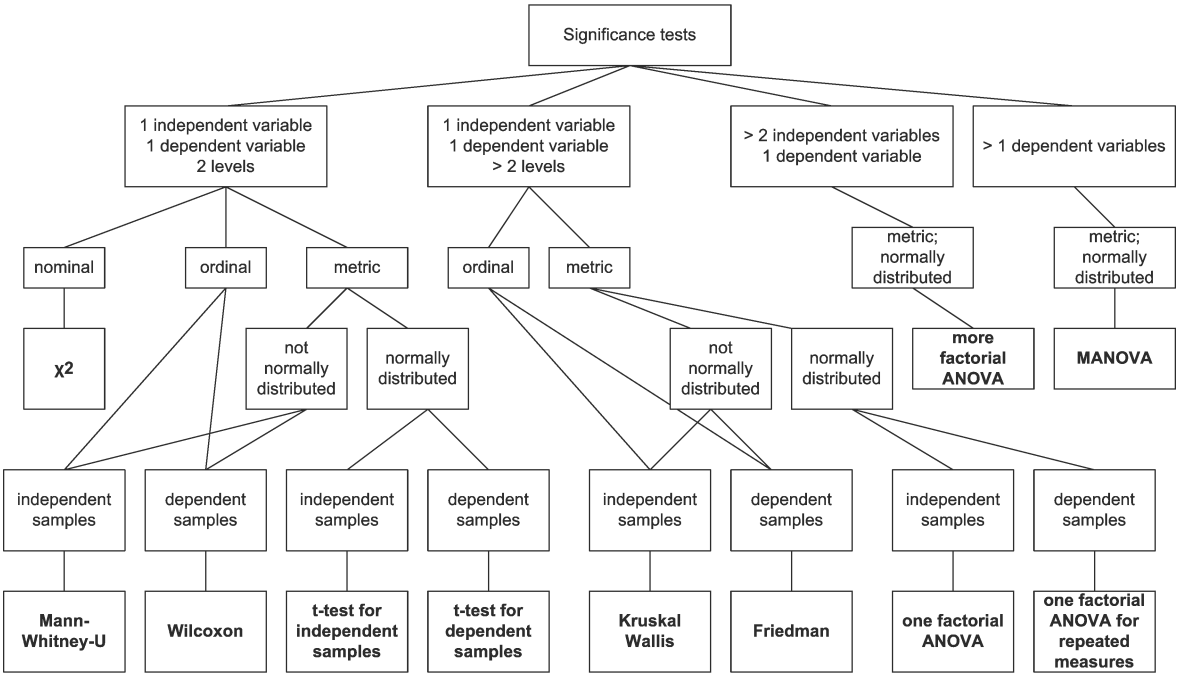


Figure 2.11: Overview of significance tests.

Usually, reporting results is strictly separated from interpreting them (AMERICAN PSYCHOLOGICAL ASSOCIATION [Ass09]). This way, results can be presented objectively, which gives the reader the chance to understand our interpretations and consequences we draw from our results.

2.3.5 Summary

In this section, we explained five stages of conducting experiments: objective definition, design, execution, analysis, and interpretation. For every stage, we introduced relevant terms and methods, which are necessary for creating unbiased results and drawing sound conclusion. In Table 2.9, we present an overview of those terms and methods, ordered according to experimental stages. In the column *Stage*, the experimental stage is denoted, in column *Term*, a term relevant for that stage, and in column *Meaning* the explanation of this term is shown. We omit the stages execution and interpretation, because we did not need to introduce relevant terms.

In the next chapter, we discuss confounding parameters on program comprehension. We do this in an extra chapter, because the number of confounding parameters is large and controlling their influence crucial for designing experiments from which we can draw sound conclusion.

Stage	Term	Meaning
objective definition	independent variable	intentionally varied by the experimenter; influences the outcome of an experiment; has at least two *levels* or *alternatives*
	dependent variable	outcome of an experiment; depends on variations of the independent variable
	hypothesis	describes the expected relationship between independent and dependent variable; must be falsifiable
design	internal validity	degree to which the value of the dependent variable can be assigned to the manipulation of the independent variable
	external validity	degree to which the results gained in one experiment can be generalized to other subjects and settings
	confounding variable	influences the depending variable besides variations of an independent variable; poses threat to internal validity; must be controlled
analysis	descriptives	describe the data, e.g., arithmetic mean, standard deviation
	inference tests	evaluate hypotheses, e.g., χ^2-, Mann-Whitney-U, and t test

Table 2.9: Summary of relevant terms for experiments.

Chapter 3

Confounding Variables for Program Comprehension

In order to properly design experiments and be able to draw sound conclusions from results, it is necessary to control the influence of confounding parameters. Otherwise, we would obtain biased results, such that we cannot explain why we observed a certain outcome (cf. Section 2.3.2.2). Controlling the influence of confounding parameters constitutes the main effort in designing experiments. Hence, in order to be able to evaluate the feasibility of experiments that assess the effect of different FOSD approaches on program comprehension, we have to identify all confounding parameters and discuss how we can control their influence.

For illustrating the influence of confounding parameters on program comprehension, we use a scenario, which we introduce in Section 3.1. After introducing our scenario, we discuss how we identified confounding parameters in Section 3.2. This is an important step, because we only can control the influence of those parameters that we identified.

In order to structure our discussion and get a better overview, we classify confounding parameters in four groups: personal, environmental, task-related, and programming-language-related parameters, which we discuss in Section 3.3, 3.4, 3.5, and 3.6, respectively. In Section 3.7, we summarize the results of this chapter.

3.1 Example scenario

In order to clarify how confounding parameters affect program comprehension, we use a scenario in the remainder of this chapter, which we explain in this section. Our scenario is:

> *A small company releases an* Operating System (OS). *Shortly thereafter, a severe bug of the paging module is found, which leads to a deadlock under certain circumstances and requires a reboot. Unfortunately, the programmer who implemented the paging source code, Howard, is on vacation. Hence, the task to fix the bug has to be delegated to persons that are available: Penny, Sheldon, and Leonard.*

Penny has programmed GUIs *for about 10 years, visited a technical school, studied computer science, and has an above average IQ. Sheldon has also 10 years of programming experience, however in the domain of OSs. He visited a normal school, studied business informatics, and has a normal IQ. The third programmer, Leonard, just finished his education as programmer and has no practical programming experience. He visited a technical school, like Penny.*

None of the three programmers is familiar with the program. Thus, all have to understand the paging source code, before they can fix the bug.

Our scenario is summarized in Table 3.1. All three programmers differ in some of the personal parameters we identified. Each parameter influences program comprehension and the probability with which a programmer succeeds first in fixing the bug. Does Penny fix the bug before Sheldon, because she is more intelligent? Is Leonard faster than Penny, because he visited a technical school? Or is Sheldon successful because of his experience with OSs?

Parameter	Penny	Sheldon	Leonard
Programming experience	10 years	10 years	none
Domain knowledge	GUI	OS	none
Education	Waldorf high school, computer science	normal high school, business informatics	technical high school, education as programmer
Intelligence	above normal	normal	above normal
Gender	female	male	male

Table 3.1: Characteristics of programmers in scenario.

In the following sections, we explain how those parameters can influence program comprehension. We pick each one of the parameters and compare two of our programmers. We discuss how the according parameter could influence the understanding of the source code.

Having introduced our scenario, we start explaining how we identified confounding parameters and how they can affect program comprehension.

3.2 Selection of the variables

In this section, we explain how we identified confounding variables on program comprehension. The process of identification is important, because unidentified parameters cannot be controlled, which would lead to biased results (Chapter 2.3). In order to discover all confounding parameters on program comprehension, we proceed in two steps: review of the literature and consulting experts.

3.2.1 Review of the literature

In order to get an overview of the current state of the art, our first step is to review the literature of experiments measuring program comprehension. The results can be summarized as follows:

1. A large number of confounding parameters exists.
2. Some parameters are more often taken into account than others, which counts especially for programming experience. Other important parameters include experience with a certain programming language, domain knowledge, and intelligence.

We present a summary of confounding parameters of all our reviewed papers in Table 3.2. In column *Parameter*, the confounding parameter is stated. In column *Reference*, the papers reporting the parameter are presented in chronological order. Since we only have a limited amount of resources for this thesis, we could select only few papers. In order to have a representative selection despite our limitation, we selected papers from different conferences, journals, and years. This way, we intend to include different views on program comprehension and thus have wide focus on program comprehension and confounding parameters. If we had focused on one conference, journal, or small time interval, our focus might have been too small and thus we might have neglected relevant parameters. In order to extend our literature review for confounding parameters on program comprehension, a sound meta analysis with a larger set of papers describing an experiment on program comprehension is necessary. However, due to our limited resources, we have to postpone a meta analysis for future work.

Extracting confounding parameters from the papers was not straightforward, since there is no common agreement on how to report them. In addition, current guidelines (JEDLITSCHKA ET AL. [JCP08], AMERICAN PSYCHOLOGICAL ASSOCIATION [Ass09]) do not advice where to describe confounding parameters in a report. Hence, we thoroughly examined the sections that most likely contain the information of confounding parameters, i.e., the description of the experiment, analysis of the data, and interpretation of the data. If we found no conclusive information in those sections, we also considered the remaining sections.

In order to confirm the parameters and identify further, we conducted an expert survey, which we explain in the next section.

3.2.2 Consultation of experts

Our second step was to consult programming experts about their opinions on what affects program comprehension. This way, we confirm the relevance of confounding parameters found in the literature. In addition, we can identify further parameters that we did not encountered so far.

3.2.2.1 Development of expert questionnaire

In order to asses the opinions of experts, we developed a questionnaire with closed questions, despite their shortcomings (e.g., that subjects are restricted in their answers (LARZARSFELD

Parameter	Reference
Programming experience	BOYSEN [Boy80], SOLOWAY AND EHRLICH [SE84], HENRY ET AL. [HHL90], KOENEMANN AND ROBERTSON [KR91], SHAFT AND VESSEY [SV95], BASILI ET AL. [BSL99], DUNSMORE AND ROPER [DR00], PRECHELT ET AL. [PUPT02], KO AND UTTL [KU03], HUTTON AND WELLAND [HW07]
Experience with a certain programming language	KOENEMANN AND ROBERTSON [KR91], SHAFT AND VESSEY [SV95], HUTTON AND WELLAND [HW07]
Domain knowledge	SHAFT AND VESSEY [SV95], KO AND UTTL [KU03], HUTTON AND WELLAND [HW07]
Education	DALY ET AL. [DBM+95], SHAFT AND VESSEY [SV95]
Intelligence	MAYRHAUSER AND VANS [vMV93], PRECHELT ET AL. [PUPT02], KO AND UTTL [KU03]
Gender	SHAFT AND VESSEY [SV95], KO AND UTTL [KU03]
Training of subjects	PRECHELT ET AL. [PUPT02]
Motivation of subjects	BASILI ET AL. [BSL99]
Attitude to experiment	KO AND UTTL [KU03]
Ordering effect	PRECHELT ET AL. [PUPT02]
IDE vs. text editor	HUTTON AND WELLAND [HW07]
Test effects	DALY ET AL. [DBM+95], HUTTON AND WELLAND [HW07]
Drop out	HUTTON AND WELLAND [HW07], DALY ET AL. [DBM+95]
Structure of source code	PRECHELT ET AL. [PUPT02], SHAFT AND VESSEY [SV95], FLEMING ET AL. [FSD07]
Coding conventions	SOLOWAY AND EHRLICH [SE84]

Table 3.2: Confounding parameters in literature.

[Lar44], SCHUMAN AND PRESSER [SP96])). We decided against open questions, because preliminary tests with programming experts revealed that they were not sure how to answer them, despite several attempts to clarify our concern.

For our questionnaire, we used all parameters found in the literature (cf. Table 3.2). In order to assess the relevance of each parameter, we used a three point Likert scale with the levels 'no', 'little', and 'considerable', for each parameter (LIKERT [Lik32]). The experts were asked to state 'no', if they think that the according parameter does not influence program comprehension. If a parameter has little influence, experts should select 'little'. Last, if a parameter has strong influence, experts should select 'considerable'. For a better overview, we grouped the confounding parameters into three categories: personal, environmental, and task-related parameters.

A standard procedure to assure that subjects think about questions and answer them honestly is to use *distractors* (LIENERT AND RAATZ [LR98]). In our expert survey, we included some distractors, which have no obvious influence on program comprehension, e.g., size of shoes, font of the source code, and OS of the computer that is used. The expected answers for those questions are 'no'. If few experts selected one of the other two choices, the genuineness of all answers needs to be checked. However, if most of the experts chose 'little' or 'considerable', it can be assumed that the according parameter indeed is considered by our experts as confounding.

In addition to the closed questions, experts were given the chance to enter further parameters that we did not mention, but in their opinion affect program comprehension. This way, we intend to alleviate disadvantages of closed questions.

In order to give an impression of the questionnaire, we show a sample page in Figure 3.1, which assesses personal parameters. The pages for task-related and experimental parameters were designed similarly with the according parameters. At the beginning of the questionnaire, we introduced the experts to the purpose of our survey.

3.2.2.2 Criteria for being a programming expert

After developing the questionnaire, we invited experts to complete our survey. Hence, we had to decide which persons are regarded as experts and which not. We selected programming experts according to three criteria:

1. Education in computer science.
2. Five or more years of practical programming experience.
3. At least one project with 40,000 LOC or more.

We chose those three criteria, because they were most commonly used when measuring programming experience and distinguishing experts from novices (e.g., SHAFT AND VESSEY [SV95], HUTTON AND WELLAND [HW07]). Persons who rated themselves as experts according to these criteria were included in our expert survey. Besides those three criteria, it

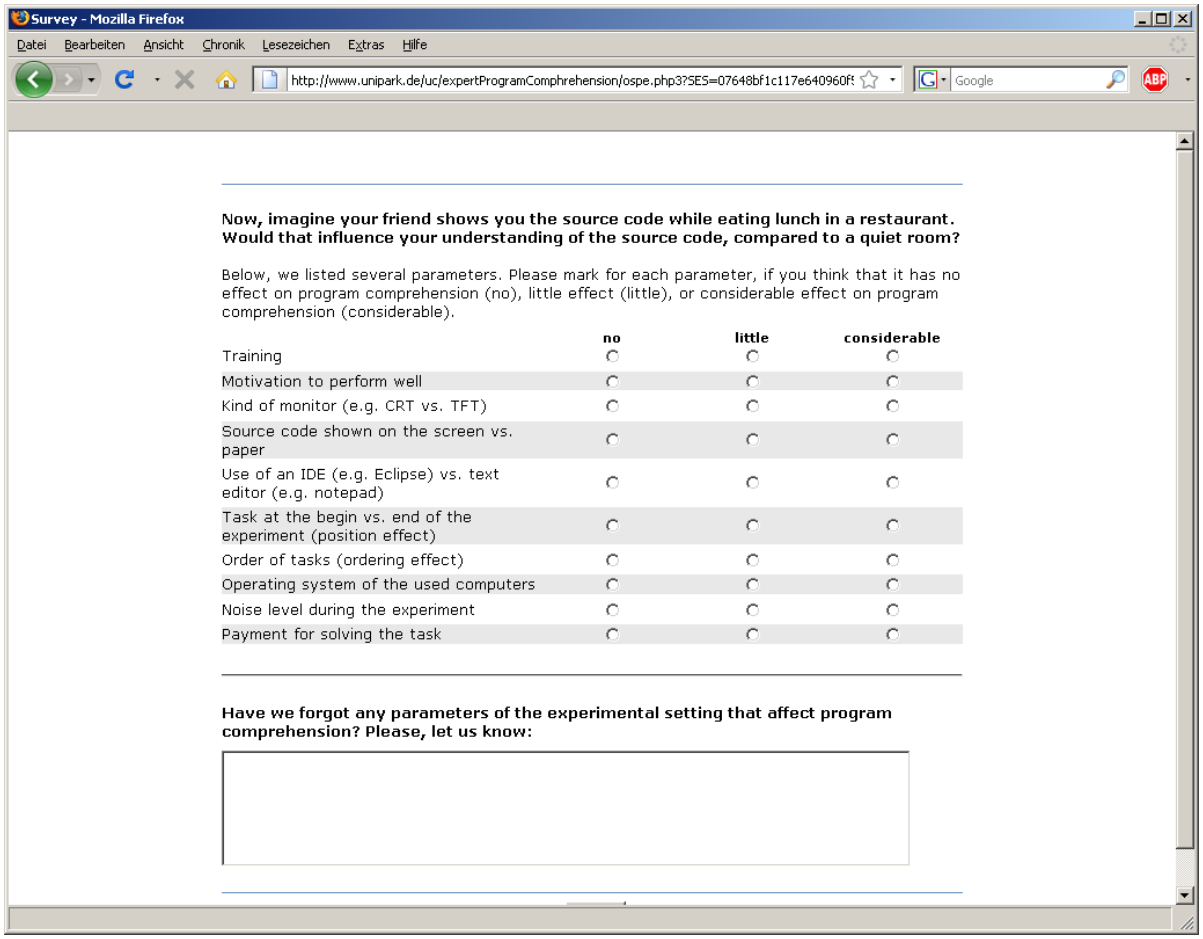


Figure 3.1: Sample page of the expert survey.

was not necessary to collect any further personal information from the experts, since we do not analyze any relation between personal parameters and responses. We contacted several software development companies and freelance developers to recruit our experts, resulting in 18 experts that completed our questionnaire.

3.2.2.3 Results of expert consultation

The high level results of our survey can be described as follows:

- All of the parameters we encountered in the literature are confirmed by our experts.
- Some of our distractors are rated with 'considerable' (e.g., dexterity), whereas others revealed the expected rating (e.g., athleticism).
- Several additional parameters were mentioned by our experts (e.g., state of subjects)

The detailed analysis of the questionnaire can be found in Table 3.3. The column *Parameter* contains the parameter the experts should rate. The numbers in the table indicate the

number of experts who state that a parameter has no influence (column *No*), little influence (column *Little*), or considerable influence (column *Consid.*). In some cases, the number does not sum up to 18, because one or two experts did not estimate the influence of a parameter. The columns *Mean* and *Std* contain the mean and standard deviation of the parameters, respectively. In order to compute both values, we coded 'no influence' with one, 'little influence' with two, and 'considerable influence' with three. Hence, a value close to three indicates a considerable influence of the parameter.

The results confirm the parameters we found in the literature. Furthermore, several parameters were additionally mentioned by one or two experts:

- state of the subjects (e.g., stressed, burnt out, happy, relaxed)
- opinion of the subjects (e.g., convinced of FOP, but not of AOP)
- workplace environment (comfort, distractions)
- degree of domain representation as reflected in the programming language
- usage of standardized algorithm/data structure patterns or libraries that provide them

We include those parameters in our discussion and start with personal parameters in the next section.

3.3 Personal parameters

In this section, we discuss personal parameters, i.e., variables related to the individual, which we obtained by our review of the literature and consultation of experts. In order to explain how a confounding factor can influence the understanding of a program, we pick a certain aspect of our scenario (cf. Section 3.1). In order to keep our discussion simple, we consider only the influence of one parameter per section and neglect the influence of all other parameters in this section.

After explaining the influence of a parameter in our scenario, we generalize our discussion. After stating the problem, we discuss possible solutions and their benefits and shortcomings.

We follow the parameters identified in Section 3.2 top down:

Programming experience (Section 3.3.1)

Domain knowledge (Section 3.3.2)

Intelligence (Section 3.3.3)

Education (Section 3.3.4)

Gender (Section 3.3.5)

Group	Parameter	No	Little	Consid.	Mean	Std
Personal	Programming experience	0	2	16	2.89	0.32
	Domain knowledge	1	6	11	2.56	0.62
	Intelligence	0	9	8	2.47	0.51
	Education	0	10	8	2.44	0.51
	Gender	18	0	0	1.00	0.00
	Dexterity	10	3	5	1.72	0.90
	Size of shoe	15	2	0	1.12	0.33
	Athleticism	17	1	0	1.11	0.47
Environ-	Training of the subjects	1	5	12	2.61	0.61
mental	Noise during experiment	2	3	13	2.61	0.70
	Motivation of the subjects	2	5	10	2.47	0.72
	IDE vs. text editor	2	6	10	2.44	0.71
	Ordering effect	10	6	16	2.38	0.50
	Position effect	1	10	6	2.29	0.59
	Code on screen vs. paper	5	11	2	1.83	0.62
	Payment for solved tasks	6	11	0	1.65	0.49
	Kind of monitor	11	5	2	1.50	0.71
	Operating system	11	5	2	1.50	0.71
Task-	Structure	0	0	18	3.00	0.00
related	Coding conventions	0	3	15	2.83	0.39
	Difficulty	0	6	12	2.67	0.49
	Comments	1	7	10	2.50	0.62
	Syntax highlighting	0	9	9	2.50	0.51
	Documentation	3	8	7	2.22	0.73
	Font of the source code	7	8	3	1.78	0.73
	Color of syntax highlighting	8	9	1	1.61	0.61

Coding: No = 1, Little = 2, Considerable = 3

Table 3.3: Number of experts that rated a parameter as no, little, & considerable influence.

```
1  private LinkedList<IPagingErrorListener> pagingErrorListeners =
2    new LinkedList<IPagingErrorListener>();
3
4  public void addPagingErrorListener(IPagingErrorListener listener) {
5    if (!pagingErrorListeners.contains(listener))
6      pagingErrorListeners.add(listener);
7  }
8
9  public void removePagingErrorListener(IPagingErrorListener listener) {
10   pagingErrorListeners.remove(listener);
11 }
12
13 public void firePagingErrorOccured(PagingErrorException e1) {
14   for (IPagingErrorListener listener : pagingErrorListeners)
15     listener.pagingErrorOccured(e1);
16 }
```

Figure 3.2: Sample implementation of observer pattern.

3.3.1 Programming experience

The first confounding variable of category personal parameters is *programming experience*. Nearly all experiments we reviewed measured programming experience of the subjects (e.g., SOLOWAY AND EHRLICH [SE84] and BOYSEN [Boy80], cf. Table 3.2). In addition, our experts confirmed the influence of programming experience. Hence, its influence cannot be neglected. First, we explain how programming experience affects program comprehension, then we discuss possible solutions. Since it is the first parameter, we explain the problem and possible solutions thoroughly in this section. In the next sections, the discussions are briefer, because most of the knowledge we present here can be easily applied to the other parameters. Furthermore, as we mentioned above, we consider only the influence of programming experience in this section, while all other parameters, e.g., education or intelligence, are neglected.

Problem statement

How could the programming experience of Penny and Leonard influence their understanding the source code? Penny worked for ten years in this company, whereas Leonard just started. Hence, Penny knows the coding conventions of her company, how to debug a program, sources of information to help her understand a program, etc., because she dealt with source code for ten years. Leonard, on the other hand, has no practical experience. Maybe he implemented and maintained some small programs during his education, but it is the first time for him to understand and debug a larger program.

*Due to her experience, Penny has an advantage compared to Leonard. For example, she encounters a code fragment like the one in Figure 3.2, which is part of the module that handles paging errors. Without having to examine the methods, she immediately knows that they implement the observer pattern (*GAMMA ET AL. *[GHJV95], p. 289ff). Leonard, at best, heard of design patterns, but never actually implemented one. Hence, he has to examine every method and needs to understand what happens statement by statement.*

From those explanations it is clear how programming experience helps to understand a program. In general, knowledge that is acquired during programming can be applied for future programming tasks. The longer a person has programmed, the greater his knowledge base is. Hence, the greater the chance is that a person has experienced a problem similar to a current one.

What are the consequences for experiments? The chance that an expert programmer has dealt with a program similar to a task defined for an experiment is considerably higher than for a novice programmer. In the case an expert knows the kind of problem, but a novice does not, the cognitive processes for understanding the source code of the task are not the same. While the expert *uses* his knowledge to solve a task, the novice *acquires* knowledge. Both, expert and novice, may succeed in solving the task, however the cognitive processes are not the same. This means that different aspects are measured, leading to results that cannot be compared. Hence, our results would be biased and we would not measure that one program is more understandable than another, but rather the effect of programming experience on program comprehension, which is not our intention. Consequently, controlling the influence of programming experience is crucial for obtaining unbiased results.

Solutions

How do we deal with the influence of programming experience? In Section 2.3, we described five techniques to control the influence of confounding parameters:

- Use confounding parameter as independent variable
- Randomization
- Pseudo randomization
- Keep confounding parameter constant
- Analyze influence of confounding parameter on result

Firstly, we could declare programming experience as independent variable in our experimental design (2.3). Hence, we would have groups with and without programming experience, and groups that test our two different programming languages. Thus, we would have two independent variables and four groups. For clarification, we show the resulting design in Table 3.4. In addition to a more complex design, the analysis of the data would also get more complicated: With programming language and programming experience as independent variables, we need to conduct a two-factorial ANOVA (ANDERSON AND FINN [AF96]), considering two main effects and one interaction.

Secondly, using randomization, we simply assign subjects randomly to groups and assume that programming experience is equally distributed.

Thirdly, we can use matching, if our sample is too small. In this case, randomization cannot be applied, because the probability that groups are homogeneous according to programming

	FOSD approach 1	FOSD approach 2
With programming experience	group 1	group 2
Without programming experience	group 3	group 4

Table 3.4: Two-factorial design.

experience is too small. In this case, we can use matching, which means that we have to measure programming experience of our subjects in advance. Then, subjects are assigned to two groups according depending on their programming experience. This assures homogeneous groups regarding programming experience.

Fourthly, in order to keep programming experience constant, we have to select subjects with a certain level of programming experience. This also requires measuring programming experience in advance, so that all persons not matching this criterion can be excluded beforehand.

Fifthly, we can analyze the effect of programming experience on our results. This is advisable when we have to deal with a fixed sample, because we cannot exclude subjects, maybe even not assign our subjects to experimental groups. In this case, programming experience can be measured and its influence on program comprehension can be analyzed afterwards.

Except when using randomization as controlling technique, we have to *measure* programming experience. In order to reliably measure programming experience, a standardized or questionnaire is a good choice. However, we did not encounter any such measurement instrument in our review of the literature, but the opposite is the case: Researchers use different indicators for programming experience, e.g., number of programming courses (HUTTON AND WELLAND [HW07]) or years of practical programming experience (SHAFT AND VESSEY [SV95]). The problem with different measures is that they are not comparable: A subject having programmed for twenty years has different programming experience than a subject having heard a couple of courses at the university. Hence, both subjects differ in their cognitive processes when understanding a program. Thus, reliably measuring programming experience requires considerable effort at the current state of the art. However, once a reliable questionnaire exists, measuring programming experience is relatively easy.

Which controlling technique should we use? The optimal solution depends on the kind of questions the experiment should answer and on the circumstances. For example, if we easily can recruit subjects (e.g., we use students from an introductory course, which are usually attended by few hundred students), then randomization is a good choice, since our sample is large enough for most probably creating homogeneous groups. On the other hand, if assessing programming experience can be done by simply administering a five minutes questionnaire, then matching or keeping programming experience constant is more advisable, because it is assured that the influence is controlled.

Thus, the decision to use one of the techniques depends on the circumstances of our experiment, for example financial and human resources, availability of test instruments, or population from which subjects can be drawn.

In our experiment (cf. Chapter 5), we use a carefully designed questionnaire to assess programming experience of our subjects and create homogeneous groups by matching. We elaborate this decision (and all others for controlling confounding parameters) more detailed in Chapter 5, because we only want to give an impression in the current chapter.

In this section, we explained the influence of programming experience and solutions to deal with it in detail. We described how the five techniques for controlling the influence of a parameter we introduced in Section 2.3 can be applied to programming experience. Since for the other confounding parameters the application of those methods is straight forward, we do not repeat this detailed description. Instead, we focus on how to measure a confounding parameter, because for all techniques (except randomization), a sound measurement is necessary to control the influence of a confounding parameter. Furthermore, for nearly all solutions, a measurement of the parameter is necessary. Even for randomization we could measure a parameter in order to confirm that we have indeed created homogeneous samples.

3.3.2 Domain knowledge

Another parameter influencing program comprehension is domain knowledge, as our literature review and expert survey revealed (cf. Tables 3.2 and 3.3). A domain is a particular market segment, e.g., satellite ground control system or software for mobile phones (cf. Section 2.1.1.2). We first show how knowledge of a program's domain can influence comprehension, then explain how the influence can be dealt with.

Problem statement

*We compare the domain knowledge of Penny and Sheldon. Penny has experience with GUIs for ten years, whereas Sheldon implemented source code for several different OSs. Hence, Sheldon most certainly has seen some source code for handling paging. Since the bug is a deadlock in the paging algorithm in an OS, Sheldon can benefit from his experience: He knows how different strategies work, e.g. FIFO or Round Robin (*TANENBAUM *[Tan01]). Furthermore, he knows their typical implementation and potential reasons for a deadlock.*

In contrast to Sheldon, Penny never had contact with paging strategies or deadlocks. She needs to analyze every statement and examine the control flow in order to understand the program and locate the bug. This takes much more time than comparing a typical implementation of, e.g., FIFO, with the current implementation, which produces an error.

The advantage of Sheldon compared to Penny can be explained with the models of program comprehension we presented in Chapter 2.3. If a subject has domain knowledge of a program's domain, then he uses a top-down approach to understand a program. Without domain knowledge, a program has to be analyzed bottom up, which is much more inefficient (MAYRHAUSER AND VANS [vMV93]).

If we do not consider the influence of domain knowledge in our experiment, we could be accidentally measuring that subjects with domain knowledge understand a program better than subjects without domain knowledge, when our original intent was to assess the under-

standability of two different programs, programming languages, or FOSD approaches. Hence, our results would be biased by the existence or non-existence of domain knowledge in our subjects.

Solutions

Ways to deal with the influence of domain knowledge are: use domain knowledge as independent variable, randomization, matching, keep domain knowledge constant, and analyze influence of domain knowledge on result. Except for randomization, we need to measure domain knowledge to control it. So, how can we measure domain knowledge? Fortunately, we could simply ask persons whether or not they are familiar with a certain domain. In this case, we have to make sure that participants know what a domain is.

If we have a fixed sample, e.g., from a certain company, we could use other ways to control domain knowledge. Firstly, we could choose a domain for which we know that our sample is familiar or unfamiliar with. For example, if we have subjects from a company providing software for mobile devices, we could use programs of this domain, if a top-down approach for program comprehension should be measured, or chose an unfamiliar domain, e.g. satellite ground control systems, if we want to measure bottom-up program comprehension.

Secondly, we could enforce a bottom-up approach by excluding all information of a domain from a program, e.g., naming classes and methods not according to their purpose, but `Class 1` to `Class n` and `method 1` to `method m`. Without assessing the domain of our subjects, it is assured that all subjects use a bottom-up approach, because no domain knowledge can be applied.

Last, we could enforce a bottom-up approach for program comprehension, if we recruit subjects that have no programming experience at all. This also assures that subjects are not familiar with a domain and thus use a bottom-up approach. In this case, we have to make sure that subjects are able to solve a task, e.g., by giving them a short introduction or keeping the task rather simple.

Which way is the best? Unfortunately, as for programming experience, there is no optimal way for dealing with domain knowledge. Depending on available resources and the hypotheses of the experiment, one of our proposed ways can be chosen. Since this counts for all remaining confounding parameters, we do not mentioned this anymore, but only present several ways to deal with the according parameter. In our experiment described in Section 5, we enforce our subjects to use a bottom-up approach by choosing a program of an unfamiliar domain, which we assessed by a preliminary questionnaire.

3.3.3 Intelligence

Intelligence is known to be related to success in school, university, and the job (e.g., HUNTER AND HUNTER [HH84], RIDGELL AND LOUNSBURY [RL04]). Hence, an influence on program comprehension can be assumed, as well, as our literature review and experts confirmed (cf. Tables 3.2 and 3.3).

Problem statement

We compare the intelligence of Penny and Sheldon. Penny's intelligence being above average could mean that she has better analyzing skills or a better memory than Sheldon. Thus, she can better analyze the program and fix the bug. A better memory helps her to memorize relevant or irrelevant code pieces. Thus, she does not have to search the code base as often as Sheldon, because her memory capacity is larger.

Hence, the more intelligent a person is, the more advantage he can have in understanding a program. Better analyzing skills help to understand an algorithm, or better memory could mean that a person does not have to look up so much information. In an experiment, the influence of intelligence could be problematic, because the more intelligent a subject is, the higher the probability is that he understands a program better. However, in this case, we would not only measure the understandability of a program, but also the intelligence of subjects. Hence, our results could be biased, so that we do not solely measure program comprehension, but a mixture of program comprehension and intelligence, rendering our results useless.

Solutions

Controlling the influence of intelligence means in most cases that we have to measure it: using intelligence as independent variable, matching, keeping intelligence constant, and analyzing influence of intelligence on results all require a measurement. The problem with measuring intelligence is that the understanding and, consequently, tests measuring intelligence are very diverse. Hence, to answer how intelligence can be measured, we first have to take a closer look at the meaning of intelligence.

For over hundred years now, researchers try to measure intelligence (BINET [Bin03]). As a result, there are many different tests measuring intelligence today, e.g., the Berlin Intelligence Structure Test (JÄGER ET AL. [JSB97]), the Wechsler Adult Intelligence Scale (WECHLSER [Wec50]), or Raven's Progressive Matrices (RAVEN [Rav36]). Every test has a different understanding about the essence of intelligence: Depending on the test, higher intelligence can mean better analyzing skills, memory, or creativity (e.g., Berlin Intelligence Structure Test), better practical or verbal intelligence (e.g., Wechsler Adult Intelligence Scale), or better pictorial intelligence (e.g., Raven's Progressive Matrices). In addition to the diversity, measuring intelligence is usually time consuming (conducting and analyzing take up to several hours per subject) and costly (purchasing a test can cost several hundred dollars).

The right test has to be chosen depending on the hypotheses of the experiment. For example, high verbal intelligence helps in understanding comments and documentation. If the effect of different comment styles on program comprehension should be measured (PRECHELT ET AL. [PUPT02]), then a test for verbal intelligence is a good choice. On the other hand, if the amount of source code memorized by subjects is used as indicator for comprehensibility (e.g., SHNEIDERMAN [Shn77], PENNINGTON [Pen87]), then the memory capacity of subjects needs to be taken into account. An example for analyzing the influence of verbal intelligence on program comprehension can be found in KO AND UTTL [KU03].

These explanations show that controlling the influence of intelligence is tedious: choosing

the appropriate test, purchasing the test, administer it and evaluate the results. In order to avoid this effort, other, more easily accessible measures could be used. As we mentioned earlier, academic or job-related success correlates with intelligence. Hence, we could use the average grade of students or the position in the hierarchy of a company as indicator for intelligence. Of course, this makes our sampling less accurate, because there are other factors that influence academic success, e.g., motivation. If no surrogate measure is available, we could still randomize our sample. Hence, cost and benefit have to be carefully considered.

In our experiment (cf. Section 5), we had no resources to measure intelligence and we found no suitable surrogate measure, which is why we use randomization as control technique.

3.3.4 Education

In this section, we discuss the influence of education on program comprehension, because our literature review and expert consultation confirms and effect of education on program comprehension (cf. Tables 3.2 and 3.3). How can the effect described?

Problem statement

Sheldon visited a Waldorf high school and then studied business informatics. Hence, he had first contact with a programming language during his education at the university. This even does not necessarily mean he can implement a program, because he could have teamed up with a fellow student who did most of the implementing work.

Leonard, on the other hand, visited a technical oriented high school and finished an education as programmer. Hence, for Leonard, programming was part of his education since he was young. Thus, he formally learned how to debug a program, had experience with several programming languages, and knows how to implement some algorithms. He extended his knowledge during his education as programmer. Due to the different education, Leonard has a head start compared to Sheldon in fixing the deadlock in the OS.

Hence, persons visiting different schools have different experiences. In a technical oriented high school, the curriculum focuses mainly on topics like mathematics, physics, and computer science. In contrast, a Waldorf high school aims at thriving and prospering the individual as a whole and encourages pupils to explore themselves (EASTON [Eas97]). As a consequence, persons with an according education know better how to understand a program and fix a bug. They learned the kind of thinking necessary for programming since they were little and thus are accustomed to it. Those persons benefit from their education, in contrast to persons that never were taught how to program.

Thus, the effect of education cannot be neglected, because that could lead to biased results. Instead of assessing understandability of a program, we would also measure the influence of education. However, this would bias our results, such that we do not know to what extend program comprehension was influenced by the understandability of a program and/or the influence of education.

Solutions

If we chose to control the influence of education by using education as independent variable, matching, keeping education constant, or analyzing influence of education on result, we need to measure education of our subjects. Like for domain knowledge, we could ask our subjects about their education, e.g., which school they visited or which college courses they were enrolled. However, the measurement of education is not as easy at it might seem on first sight, because different schools or courses have different curriculums. Even the contents of the same course at the same school can significantly differ, if held by different teachers.

Hence, we need to examine the education of our subjects carefully. However, it is tedious to assess all information relevant information, for example, curriculum during education, contents, or final grade of subjects as indicator for what they have learned. For decreasing our effort, other ways are possible, e.g., include only subjects from schools with certain profiles or certain kind of finished school/university courses, assess the curriculum of subjects and analyze their influence on program comprehension afterwards, or use professionals that have programmed for most of their lives, so that the influence of education becomes negligible.

In our experiment (cf. Section 5), we use college students that are enrolled in the same course, which required the students to have finished certain basic programming courses. Since all students have completed the same basic courses at the same University, we assume to control the effect of education.

3.3.5 Miscellaneous

In this section, we shortly discuss further personal parameters that affect program comprehension, i.e., gender and cultural background.

Gender

The influence of gender on program comprehension may not be obvious. However, there are several studies analyzing the effect of gender on numerous parameters, e.g., mental rotation abilities QUAISER-POHL ET AL. [QPGL05] or academic success (ZOHAR [Zoh03]). We could not find any studies showing an effect of gender on program comprehension. In addition, none of our experts think that gender affects program comprehension. Despite those findings, we cannot neglect gender as confounding parameter, because differences in gender regarding several cognitive abilities are proven (e.g., MACCOBY AND JACKLIN [MJ74], KIMURA [Kim92], or HALPERN [Hal86]). Hence, gender could directly or indirectly influence program comprehension.

Fortunately, the gender of individuals can be easily assessed by asking them or deciding on the basis of their first names. In our experiment (cf. Section 5), we asked the subjects about their gender and created homogeneous groups.

Cultural differences

A further confounding parameter describes cultural differences. The effect of cultural differences is well known in intelligence research, which is why special tests exist to assess intelligence (e.g. RAVEN [Rav36], CATTELL ET AL. [CFS41]). Those tests exclude verbal and numerical aspects of tasks, so that only logical skills are assessed. Since implementing a program is connected with verbal information, cultural differences in programming skills and program comprehension can be assumed.

Like for gender, assessing the cultural background can be done easily, for example with a simple questionnaire. In order to control the influence of cultural differences in our experiment (cf. Section 5), we took our complete sample from the University of Passau of an advanced programming course, which required certain other courses to be completed. Since the enrolled students studied for several semesters in Germany, the influence of cultural differences should be decreased.

In the next section, we discuss the influence of parameters that are related to the experimental setting.

3.4 Environmental parameters

In this section, we consider the influence of environmental parameters, i.e., parameters of the experimental setting. For every environmental parameter, we explain with our scenario and sample experiment, how it can influence the comprehension of a program. Then, we explain how the influence can be controlled.

In contrast to personal parameters, parameters of this category can be manipulated by the experimenter, because they are known in advance. Hence, controlling these parameters can be included in the planning phase. Personal parameters, however, can only be controlled when we start to recruit subjects, since we usually do not know the attributes of our subjects in advance.

We explain the parameters identified in Section 3.2 top down:

- Training of the subjects (Section 3.4.1)
- Motivation of the subjects (Section 3.4.2)
- Tool support (Section 3.4.3)
- Position and ordering effects (Section 3.4.4)
- Effects due to the experimenter (Section 3.4.5)
- Hawthorne effect (Section 3.4.6)
- Test effects (Section 3.4.7)
- Noise level (Section 3.4.8)

As in the previous section, we consider only the influence of one parameter at a time, while neglecting all other parameters.

3.4.1 Training of the subjects

In this section, we explain how training of subjects can influence their program comprehension. Training in the context of experiments means the preparation of subjects before they start to work on a task (MARKGRAF ET AL. [MMW+01]). The effect of training can be compared with the training of a professional long distance runner, who shows better performance on a ten kilometer race than an untrained person, despite the same potential both runners have. In the same way, trained subjects have an advantage in program comprehension compared to untrained subjects.

Problem statement

If Howard has explained to Penny how he implemented the paging algorithm before he left, Penny already has an understanding of how the program works. Thus, when Penny and Leonard start to work on fixing the bug, Penny has a head start. She knows the purpose of several classes and methods and thus can use a top-down approach on several source code fragments, whereas Leonard has to use a bottom-up approach all the time.

Hence, we have to make sure that all subjects receive the same training before an experiment. Otherwise, some of our subjects have a head start in completing our task. As a consequence, some subjects might be better able to deal with a task than other subjects, but not because the task is simpler or the program more understandable, but because they were prepared better. Hence, we need to control the influence of training, so that our results are not biased by differences in the training.

Solutions

In order to avoid differences in training, we have to make sure that both groups get the same training. There several ways to do so, e.g., instruct the experimenters on how to train subjects, use the same experimenter for all trials, or conduct one training session for all subjects at once. We explain those methods next.

Firstly, we can train our experimenters. If we do not instruct our experimenters how to conduct the training, we cannot be sure to have the same training effect. Thus, we have to give detailed instructions to both experimenters and make sure they use the instructions properly.

Secondly, we could use the same experimenter for all trials. This reduces variations due to the person of the experimenter. In order to further minimize that subjects receive different trainings, we should train our one experimenter on how to use standardized instructions.

Thirdly, we can simply omit training. Thus, we do not have to deal with assuring the same training effect for all subjects.

Last, we could give one training session for all subjects in one room. In this case, we have to make sure that the time difference between the training and when subjects start to work on a task is comparable. Hence, we cannot conduct a single training session, when part of the subjects have the experiment right after the training session, while others wait one week for the experiment.

Depending on the resources and hypotheses of the experiment, one of these ways has to be chosen. For example, if we have to test all groups at the same time, we need to use several experimenters and make sure that they give the same instructions. Of course, this is only possible we have the according financial and human resources. In our experiment (cf. Section 5), we give our introduction in a single room for our complete sample.

3.4.2 Motivation of the subjects

In Section 3.3.3, we mentioned that motivation is an important parameter for academic success. Persons that are motivated to achieve a goal work hard to succeed, in contrast to persons who are forced to do something or do not care about the outcome of their work. The same counts for the performance in experiments.

Problem statement

Sheldon works in the same department as Howard. Hence, he does not want that his department looses its good reputation. Thus, he is motivated to fix the bug in the paging algorithm and thus works overtime. As a consequence, he spends much time and effort in understanding Howard's program. Penny is assigned to the task, because she is one of the persons with the most programming experience in the company. She does not really care whether she gets to fix the bug, or whether Sheldon succeeds. Hence, she just spends some of her working hours on the problem. Finally, Leonard dislikes Howard, because Howard voted against Leonard's employment in the company. Thus, he might even wish that Howard gets fired for his mistake and thus claims – without trying – that he cannot understand the source code due to bad coding and commenting style.

Since Sheldon is the most motivated of the three, he fixes the bug first. In contrast to Sheldon, Leonard rather sabotages the attempt to fix the bug and does not succeed. Penny, whose attitude towards fixing the bug can be described as neutral, eventually fixes the bug, but needs more time than Sheldon.

Generally, motivation can affect our behavior and accomplishments (MOOK [Moo96]). If we are highly motivated to achieve a goal, we work hard to succeed. On the other hand, if we do not care for the outcome of our work, we spent only as much time on a project as we can spare. The same counts for subjects in experiments: Highly motivated subjects tend to perform better than subjects that are not motivated. Hence, we have to control the influence of motivation, otherwise, we would not measure understandability, but a combination of understandability and motivation. This would produce useless results.

Solutions

From our scenario it is clear that a low motivation diminishes the effort of subjects. In addition, motivation to sabotage the experiment produces unusable results. Hence, all subjects must be motivated at least a little to perform well. In order to increase motivation, several ways are possible: pay subjects for participation or reward subjects for good performance.

Firstly, all subjects could get paid for participation in the experiment. However, this assumes that there are enough financial resources. A further problem is that payment does not necessarily increase motivation to perform well, but can even diminish the effort of subjects (FESTINGER AND CARLSMITH [FC59]).

Secondly, subjects could be rewarded for good performance, e.g., one dollar for each correct response, paying the first subject to finish, or subtract one dollar for each false response. If students are used, the performance in the experiments can be considered for the final grade of their course. With limited financial resources, rewards could be raffled (e.g., tickets for theater), whereas subjects with good performance could have a higher chance of receiving the reward.

In our experiment (cf. Chapter 5), subjects were required to take part to finish their course. In addition, we raffled a gift certificate. In order to check the influence of motivation on our results, we asked subjects for each task how motivated they were to solve it.

3.4.3 Tool support

Different tools provide different support for creating and analyzing source code, which can have different effects on program comprehension. Our experts confirmed that tool support is a confounding parameter. In this section, we explain how this can influence program comprehension.

Problem statement

Penny uses usually uses Eclipse[1] *for implementing source code, because it provides several useful features, like outline view, code assist, and code folding (*CLAYBERG AND RUBEL *[CR06]). Sheldon and Howard also like the comfort provided by* Integrated Development Environments (IDEs), *but prefer Microsoft Visual Studio*[2]*, because the support for C++ is more sophisticated (*NAYYERI *[Nay08]).*

Unfortunately, Penny's computer broke down, so she has to use Howard's computer with Microsoft Visual Studio instead. In addition to understanding Howard's program, Penny now has to get used to a new IDE. Hence, Sheldon has a head start compared to Penny and thus most likely succeeds first.

Since Leonard is new to the company, he has not decided for an IDE yet and works with a text editor. Besides syntax highlighting, no other functionality is provided. Compared to

[1] http://www.eclipse.org/

[2] http://msdn.microsoft.com/

Sheldon and Penny, he has no tool supporting him in navigating through the source code, e.g., open the definition of a method when the call is encountered somewhere in the source code.

There are further tools that help to analyze source code beyond the support integrated in a standard IDE, for example, FEAT (ROBILLARD AND MURPHY [RM03]), StackAnalyzer (EVSTIOUGOV-BABAEV [EB01]), or SeeSoft (EICK ET AL. [ESEES92]). The use of such a tool would further influence how a program is understood.

In general, tools can support program comprehension. Even after an equal amount of time, persons can use different sets of features of the same IDE. Hence, letting persons use the same IDE does not control the influence of tool support sufficiently: either some subjects need to familiarize with it, or do not know how to use a certain functionality, etc. Letting subjects use their preferred IDE prevents that subjects have to familiarize with an unknown tool, but introduces variations in tool support. Hence, in order to obtain unbiased results in an experiment, the influence of tool support needs to be carefully considered.

Solutions

In order to control the influence of tool support, several ways are possible: define one tool all subjects are familiar with, let subjects use their preferred tool, or let all subjects work with the same editor.

Firstly, all subjects could be restricted to use the same tool, e.g., Eclipse. In this case, we have to make sure that all subjects are familiar with Eclipse in a comparable degree. As further restriction to control tool support, the allowed features that subjects can use could be defined. This way, we know exactly which support subjects have.

Secondly, we could let the subjects decide about the tool they use. This way, every subject can use the preferred tool. This approach is advisable, if it should be measured how well students understand a new programming language after a certain course, because they are familiar with it.

Thirdly, we let all subjects work with the same text editor. Furthermore, we could only include those subjects that are used to implement source code with an IDE, so that none of our subjects is familiar with a text editor. This way, we do not give any subject a head start and we have comparable results, because the provided support is the same.

Which way to choose depends on the focus of the experiment: If the comprehensibility of two languages should be compared, tool support would confound the result, because not the comprehensibility of the language alone, but also how the language is supported by the tool is measured. On the other hand, if skills of subjects should be measured, then letting them use their preferred tool is more advisable, because they do not have to familiarize with a new one. One example of controlling the influence of tool support can be found in HUTTON AND WELLAND [HW07]. In their experiments, subjects were allowed to use their preferred text editor, so that tool support was eliminated, but subjects could use a familiar editor. However, before features of a tool can be used, persons need to familiarize with it. In our experiment (cf. Chapter 5), we eliminate all tool support by displaying the source code as HTML files and forbid the search feature of the browser.

3.4.4 Position and ordering effect

Position effect refers to the influence of position of the task relative to the beginning of an experiment. Usually, subjects are more tired at the end of the experiment than at the beginning. Ordering effect means that the order in which tasks are applied influences the result. Hence, the result differs depending on the order of the applied tasks. Since both effects are very similar, we discuss only the position effect in detail and make few statements for the ordering effect.

Problem statement

Penny starts to work on the source code after several hours of work. Since she already feels fatigue when starting to fix the bug, she finds it harder to concentrate and focus on the work. Hence, she uses only a fraction of her cognitive resources, which negatively affects the success of her work.

Leonard, on the other hand, starts to work on the bug first thing in the morning. Since he is rested, he does not feel fatigue, can focus on his task, and thus make more progress than Penny in the same time.

Sheldon finishes implementing a different program, which takes him half an hour, before he starts to work on Howard's program. Thus, compared to Leonard, Sheldon is already warmed up and does not need to get used to the way of thinking that is necessary for understanding and maintaining a program.

The same problems can be observed in experiments. First, subjects have to get used to the experimental context, e.g., the room, the tasks, and the material. After getting used to the situation, the actual performance of subjects can be assessed. The longer the experiment lasts, the more fatigue subjects get, and performance decreases again. Hence, the performance of subjects depends to some extent on the position of a task in the experiment. If this effect is not considered, results could be biased.

The ordering effect is very similar. It describes the influence due to the ordering of the task. For example, if subjects that are familiar with Java first work with a program implemented in C, they have to familiarize with the syntax *and* with the program. If subjects then work with a Java version of the same program, they already are familiar with both the syntax *and* the problem. If we switched the order of tasks, then subjects would first familiarize with the problem (Java version), then familiarize with the C syntax (C version). Consequently, we would obtain different results for each task depending on the order in which we apply them. Hence, we have to carefully decide on the order of the task.

Solutions

In order to control for the position and ordering effect, several ways are possible: use a within-subject design, randomize the order of the tasks, or assess subject's opinion.

Firstly, a within-subject design can be used if we have only two or three tasks (Table 3.5). We have two groups, A and B, which both complete both tasks, but in a different order. The

Group	Trial 1	Trial 2
A	task 1	task 2
B	task 2	task 1

Table 3.5: Design for controlling position effect.

time interval between both trials can be five minutes or several days, depending on how long each trial is. The effect of position or order can then be analyzed, e.g., by comparing the results of both groups in each trial, the overall performance, or the performance for each task.

Secondly, if the number of tasks is too large, a within-subject design is not feasible, because every permutation of tasks needs to be included. For example, for three tasks we need six groups, because there are six permutations. If we have four tasks, we already need 24 groups. In this case, we can randomize the order of the tasks for every subject, so that position and ordering effects vanish.

Thirdly, we could ask subjects at certain stages of the experiment whether they feel fatigue, stressed out, or they think if their performance would have differed if the task were applied in a different order. The correlation of the answers with the performance can be used to analyze position and ordering effects. In our experiment (cf. Chapter 5), we used a warming up task and task with approximately ascending levels difficulty.

3.4.5 Effects due to experimenter

The experimenter can influence the results of the experiments due to his behavior, intentionally or unintentionally. Since this is a problem of all experiments involving subjects, we did not ask our experts. Examples for effects due to the experimenter can be found in GOODWIN [Goo99]. We explain in this section, how the experimenter can affect program comprehension.

Problem statement

Sheldon, Leonard, and Penny manage to contact Howard on his vacation. Howard and Sheldon work in the same department and both know each other. Since Howard wants that the bug is fixed by his department, he helps Sheldon to understand the program.

Howard voted against Leonard's employment, because he thinks that Leonard is not skilled enough for the company. In addition, Howard developed an antipathy against Leonard, because Leonard got the job, anyway. Thus, the willingness to support Leonard in fixing the bug is rather low, leaving Leonard on his own.

In an experiment, the experimenter can introduce a bias to our results. This bias can occur during conducting, analyzing, or interpreting the results. The reasons are diverse, e.g., he likes one group of subjects more, he noted in a former trial that with several tips subjects are happier or perform better, he is attracted to one of the subjects, he prefers a certain outcome, etc. The experimenter can even unintentionally behave different in different trials, because he favors

a certain hypothesis or simply has a better day. Unintentional differences in experimenter behavior that lead to different performances of subjects are referred to as Rosenthal effect (ROSENTHAL AND JACOBSON [RJ66]).

Solutions

Effects due to the experimenter can be controlled in several ways: clear or standardized instructions for the experimenter, training of the experimenter, or single blind trials.

Firstly, we could give the experimenters clear instructions on what they should or should not say. In order to nearly delete the influence, experimenters could get a standardized script and just read it to the subjects. Questions of the subjects occurring during the task could be answered with a standardized reply. However, this leaves experimenter with no way to react to unexpected questions or behavior of the subjects.

Secondly, in order to allow experimenters to flexibly react to subjects during the experiment without biasing the result, experimenters could be trained. Of course, this negatively affects time and financial resource of our experiment. Hence, we have to compare cost and benefit of well trained experimenters versus the bias on our result.

Thirdly, especially to control the Rosenthal effect, we could let the experimenter unclear about the intentions and hypotheses of our experiment. This prevents subconscious shift of the behavior, since the experimenter does not know which the preferred outcome is. One example of standardization during analysis of data can be found in SHAFT AND VESSEY [SV95], where a coder's manual to objectively classify recorded statements from think aloud protocols. In our experiment (cf. Chapter 5), we carefully designed the instructions, answered only questions concerning the tasks, and cross checked our analysis of the answers of subjects.

3.4.6 Hawthorne effect

The Hawthorne effect describes that subjects do not show their usual behavior, because they take part in an experiment (ROETHLISBERGER [Roe39], LANDSBERGER [Lan58]).

Problem statement

Since Leonard is new to the company, he is on probation for the first month. Hence, if his performance is not good, he will be let go. Thus, Leonard is highly motivated to fix the bug, because a good performance in the first month is beneficial for his job. Penny, on the other hand, has worked for ten years in this company. Hence, a bad performance for fixing the bug does not endanger her job. Due to the pressure, Leonard tries harder to succeed with his assignment, while Penny can begin her task more relaxed.

The Hawthorne effect was discovered during experiments in the Hawthorne company, in which the influence of the light level on working efficiency was assessed (ROETHLISBERGER [Roe39]). It was discovered that subjects performed better, simply because they knew they were observed. Hence, the effect is especially problematic in social sciences, where natural

behavior should be assessed (HEWSTONE AND STROEBE [HSJ07]). In experiments that measure behavioral aspects of program comprehension, e.g., efficiency of solving a task, behavior during a maintenance task, the Hawthorne effect could bias the results. In addition, if attitudes towards FOSD approaches should be assessed, subjects could hide their real opinion, rendering our results useless. Hence, the Hawthorne effect needs to be taken into account when planning an experiment in order to obtain useful results.

Solutions

In order to control the influence of the Hawthorne effect, there are three ways: either let all subjects know that they participate in an experiment, hide the real intentions of the experiment, or let all subjects unaware that they take part in an experiment.

Firstly, all subjects could be made aware that they take part in an experiment. This solution can be applied, if, e.g., two groups should be compared regarding their performance, because an increase in performance due to the Hawthorne effect occurs in both groups.

Secondly, we could hide the real intentions of our experiment. This approach is usually applied in social sciences (HEWSTONE AND STROEBE [HSJ07]). In this case, the Hawthorne effect occurs, however the behavior that should be measured remains unaffected.

Thirdly, we could let subjects unaware that they take part in an experiment. The second and third way are referred to as single-blind trials. In both cases, it is imperative that the experiments are ethically approved, because we cannot get the subjects consent before the experiment. In Germany, ethics committees have developed, which can be consulted for determining whether subjects are exploited or planned interventions are unreasonable (WIESING [Wie03]).

An extension to single-blind trials are double-blind studies, which means that both, the subjects and the experimenter, are 'blinded' regarding the experiment or its intention. This way, both the Hawthorne and Rosenthal effect can be controlled simultaneously. In our experiment (cf. Chapter 5), we let all subjects know that they participated in an experiment.

3.4.7 Test effects

Test effects occur usually with repeated measurement and similar tasks (SHADISH ET AL. [SCC02]). In this case, subjects learn information in the first trial that affect performing in following trials.

Problem statement

Since Sheldon works in the same department as Howard, he had to deal with Howard's source code a couple of times. Hence, Sheldon is familiar with Howard's coding style. On the other hand, Leonard has first contact with Howard's coding style, which means that he has a harder time understanding the source code.

For experiments, this problem occurs when having a design with repeated measures, i.e., using the same subjects again for another trial. If the trials are very similar, then subjects profit from the first trial. This would confound the results of the second trial, because they also depend on the first trial. Hence, we cannot reliably compare the results of the first with the second trial. Thus, carefully considering test effects is essential for useful results.

Solutions

In order to avoid test effects, we could let a large time interval pass between two trials or administer different tests.

Firstly, we could wait a considerable amount of time between to trials, e.g., one year. However, we then have to deal with mortality, i.e., that subjects drop out of our experiment for whatever reason (SHADISH ET AL. [SCC02]). Test effects diminish with a larger time interval, whereas mortality increases with the time interval.

Secondly, for avoiding mortality, we can administer different tests. This way, we do not need to let a large time interval pass. However, we have to make sure that both tests are comparable in their difficulty. This could be done by letting equally skilled subjects conduct both tests. If our tests have an equal level of difficulty, then the performance of all subjects should be nearly the same. As another way, we could ask subjects about their perception of the difficulty of the conducted tests.

In our experiment (cf. Chapter 5), we avoid repeated measures by creating two homogeneous groups that worked on different versions of the source code.

3.4.8 Miscellaneous

In this section, we discuss some further environmental parameters. We explain them altogether in this section, since their influence and ways to control can be shortly explained.

Parameters related to the computer system, for example, CRT vs. TFT, keyboard, mouse, or OSs, should be kept constant or varied according to the preference of the subjects (cf. Section 3.4.3). Especially in the context of measuring program comprehension, these parameters need to be considered. In our experiment (cf. Chapter 5), we used the same configuration of the working stations for all subjects.

The same noise level during the complete experiment should be maintained. Otherwise, if one group hears the lawn-mower outside the whole time, while the other group gets to work in silence, the results would be biased. As for light, it is not advisable to let one group of subjects work in daylight, while the other group has to work in darkness, just having the lights from their monitors. We conduct our experiment (cf. Chapter 5) in one room, so that all subjects have the same noise and light level.

In the next section, we explain the influence of those parameters that are related to the task itself, i.e., task-related parameters.

3.5 Task-related parameters

In this section, we discuss the influence of variables that are related to the tasks subjects should perform. For experiments measuring program comprehension, those parameters are mostly related to source code. Programming language can be understood as task-related parameter, however we describe its influence in an additional section, because there are many ways in which it can influence program comprehension. Hence, this section would become too large or the importance of programming language neglected. Like for the environmental parameters, the influence of those parameters can be included in the planning phase of the experiment.

We discuss parameters in the following order:

- Structure of the source code (Section 3.5.1)
- Coding conventions (Section 3.5.2)
- Difficulty of the task (Section 3.5.3)
- Comments and documentation (Section 3.5.4)

In Section 3.7, we summarize all parameters, we discussed, including personal and environmental parameters.

3.5.1 Structure of the source code

In Section 2.1.1, we discussed the necessity for SoC and compared two differently structured implementation of our stack (cf. Figure2.1 on Page 6). In this section, we show how the structure affects program comprehension.

Problem statement

Howard has implemented several programs before. Hence, he knows that well structured source code increases its readability. Thus, he divided his source code into several modules, all of the methods fit on the screen without scrolling, and none of his classes exceeds five hundred LOC. Due to this structuring, he helps Penny, Sheldon, and Leonard to understand his program.

In contrast, if Howard implemented his source code in just one class, the class would be very complex. Hence, it would be rather difficult to understand the program.

If we are not careful in our sample experiment when designing the different versions of our program, then we could create one well structured and one badly structured version. This would lead to an advantage for the group that analyzes the well structured source code. Thus, we would not measure that one programming language is more understandable, but that one implementation is better structured than the other.

Solutions

How can we assure that different implementations have comparable structure?

One way is to let programming experts review the different versions. Since experts have programmed for several years, they know different ways to structure source code. They have a feeling for which different versions of source code are comparable regarding their structuring. If we do not have access to experts, we could ask subjects afterwards how they perceived the structure of the source code and if two versions are comparable.

A further way is to conduct a pretest, in which we compare two versions. If we assume that two version do not differ in their structuring (and any other aspects), then we can expect the same response from our pretest subjects. However, if we obtain a significant difference in the performance, then we must assume that the two versions are not comparable. A similar solution is discussed in Section 3.4.7, in which we explained how test effects can be controlled.

In our experiment (cf. Section 5), we use a program that is code reviewed by experts. This way, we control all task-related parameters except difficulty of a task. Hence, we only mention how we control difficulty in our experiment (Section 3.5.3). All other task-related parameters are controlled by using a code reviewed program.

3.5.2 Coding conventions

Coding conventions were developed to standardize coding styles of different programmers. Coding conventions suggest, e.g., how text should be indented or how methods, variables, and classes should be named.

Problem statement

Howard likes to have as much source code on the screen as possible. Hence, his statements often exceed the suggested eighty columns. Furthermore, since text indentation takes up place, all of his statements begin in the first column. In addition, he does not like to name methods according their purpose, but after Star Trek characters. Usually, that is not a problem, because Microsoft Visual Studio has an autoformat feature, which formats the text according to specified conventions, e.g., statements shorter than eighty columns and two white space as indentation. With the refactor feature, method names can be changed, so that every method is named according to its purpose. However, Howard forgot to restructure his source code, so his colleagues have to work with a source code in which most of the coding conventions are violated.

The first thing that Penny and Sheldon do is to use the auto format feature of their IDE, so that the source code is formatted as they are used to. What about method names? Naming a method according to its purpose, without knowing its purpose, is impossible. Hence, Penny and Sheldon have to find out the purpose of methods by analyzing what they are doing, which increases the effort in understanding the program.

An important result on the influence of coding conventions on experts and novices was

found by SOLOWAY AND EHRLICH [SE84]: The performance of experts was identical with the performance of novices, if the coding conventions were violated. Otherwise, the experts clearly outperformed novices. Hence, the influence of coding conventions needs to be carefully controlled, otherwise our results would be biased and useless.

Solutions

We can control this parameter the same way we can control the influence of structure of the source code: consult experts, conduct pretests, or assess the opinion of subjects. In addition, we could disrespect coding conventions intentionally.

3.5.3 Difficulty of the task

The difficulty of the task does not directly affect program comprehension. However, since we use tasks to measure how well programs are understood, the difficulty of the task influences our result and thus needs to be controlled.

Problem statement

What if the bug in Howard's program was not a deadlock, but a wrong background color in several conditions? Changing the color of an output is easier than resolving a deadlock. Supposedly, Sheldon would be the only programmer assigned to fix the bug and would probably be finished within half an hour, because displaying the right color is rather easy.

If we do not design the tasks for our sample example careful enough, we could end up with easy tasks for one group, and difficult tasks for the other group. Although both groups understand the program well, the group with the difficult tasks shows worse performance. However, we would assume that the programming language of group two is less comprehensible.

Solutions

Hence, we have to make sure that the tasks we use to measure program comprehension have comparable levels of difficulty. Like for controlling the influence of coding conventions, we could consult experts, ask our subjects' opinion, or do not use tasks at all to measure program comprehension.

In the latter case, we have to use other indicators for program comprehension, e.g. ask the subjects how well they have understood a program. However, relying on a subjective measure solely is not advisable, because the reliability of this procedure is rather low (DUNSMORE AND ROPER [DR00]). Potential reasons are that subjects could be wrong in their estimate, or simply lie, since they do not want to admit having understood less than they expected or because of social desirability, i.e., behaving in a way that is viewed as favorably by others (CROWNE AND MARLOWE [CM60]). Hence, subjective rating should only be used in addition to other measures, e.g. maintenance tasks.

In our experiment (cf. Section 5), we use the same tasks for different source code versions to assess program comprehension of subjects.

3.5.4 Comments and documentation

On the one hand, useful comments and good documentation can help to understand a program (MCCONNELL [McC04]). On the other hand, badly commented source code or chaotic documentation has a negative effect on program comprehension.

Problem statement

Howard is usually too lazy to write comments or documentation and since nobody in his company forces him, Howard's source code is usually uncommented and without documentation. This makes the job harder for Penny, Sheldon, and Leonard. If we consider the source code implementing the observer pattern (cf. Figure 3.2 on page 53), then comments that actually describe this code fragment as observer pattern helps understanding it, being familiar or unfamiliar with the observer pattern. Penny and Sheldon can simply check whether everything is implemented correctly, and Leonard can look up the observer pattern.

One experiment comparing different commenting styles was conducted by PRECHELT ET AL. [PUPT02]: They developed one commented version of source code, which implements several design patterns. The second version was identical to the first, but had additional comments that reveal when code fragments implemented a certain design pattern. Due to the high degree of similarity between both versions, the observed differences in performance of subjects in favor of the version that contained design pattern comments can only be attributed to the additional comments.

An interesting observation regarding documentation was made by KOENEMANN AND ROBERTSON [KR91]: Their subjects should maintain an unknown program and were recorded using think aloud protocols. Subjects looked at documentation of source code only as last resort. This unwillingness to use documentation should be taken into to account when designing an experiment, in which the documentation of source code plays an important role.

Hence, if for an experiment different versions of a program exist, it has to be made sure that all versions are commented and documented in a comparable way. Otherwise, we would not only measure the understandability of a source code, but also the quality of comments and documentation, leaving us with biased results.

Solutions

For controlling the influence of comments or documentation, we could use the same means as for controlling the influence of the structure of source code: consult experts, conduct pretests, or assess the opinion of subjects. In addition, we could simply omit all comments and documentation. Then, neither can influence program comprehension.

This concludes our discussion on task-related parameters. Next, we discuss how the underlying programming language of source code can influence program comprehension.

3.6 Programming language

In this section, we explain how understanding a program can be influenced by its programming language. We do not discuss, however, the effect of different subjects knowing different programming languages on program comprehension, because we consider this to be part of programming experience.

One might argue that programming language is not a confounding parameter on program comprehension, but rather an independent variable. However, in our thesis we want to assess whether we can measure the understandability of different FOSD approaches, not different programming languages. Since one FOSD approach can include several programming languages, they can be considered as independent variable depending on the hypotheses of an experiment. For example, CPP can not only be applied to C or C++, but to any other programming languages, like Java. Hence, for assessing the understandability of CPP, other programming languages need to be considered as well.

Due to its importance, we discuss programming language in a separate section and not as part of task-related parameters. Furthermore, the influence of a programming language itself can be split into several facets. We consider the underlying programming paradigm and syntax as the most influential facets, because they cause differences between programs of different programming languages. Additionally, programming languages can influence task-related parameters.

Problem statement

We start with the underlying programming paradigm and syntax of a programming language. Then, we discuss the influence on task-related parameters.

Programming paradigm. Programming languages can look very different depending on the programming paradigm they represent. Well known paradigms (and according languages) include logical (Prolog), functional (Haskell (HUDAK ET AL. [HPJW+92])), and object-oriented (Java) programming paradigms. All paradigms have different underlying concepts. Logical programming is based on mathematical logic and represents information declaratively. The functional paradigm is based on mathematical functions. OOP uses objects that have a modifiable state for expressing information.

In order to demonstrate the difference between different programming paradigms, we show two implementations of the quicksort algorithm in Figure 3.3. One is implemented in Haskell, the other in Java. Although both implement the same algorithm, both programs differ considerably due to the underlying programming paradigm: The Haskell implementation needs few lines (Figure 3.3a), whereas the Java implementation needs much more statements (Fig-

ure 3.3b). The Haskell version works recursively, the Java version has recursive and imperative elements. The '=' sign in Haskell is used to define functions, in Java to assign values. The definition of functions/methods differs. All these differences result from the underlying programming paradigm, which shows that different programming paradigms can make a feasible comparison hard. Nevertheless, we have to consider this difference, because otherwise we would not measure the effect of different FOSD approaches on program comprehension, but also the effect of different underlying programming paradigms.

Syntax. Different programming languages even in the same paradigm have a different syntax. If we do not consider the influence of different syntax, our results would be biased, such that we not only measure the influence of FOSD approaches on program comprehension, but also differences in syntax. For example, in the C++ example, the address of the variable `arr` is used (Line 2), whereas in the Java example, the reference of the according variable (Line 1). The according operator in the C++ example, `&`, is not only used to get the address of a parameter, but also as *bitwise and* operator. Hence, it is *overloaded*, which is not possible in Java. Besides overloading operators and using the reference or address of a parameter as needed, there are further differences between C++ and Java, for example destructors, explicit memory allocation and deallocation, multiple inheritance in C++, but not in Java. Hence, although C++ and Java look similar, they have considerable differences on a closer look. Those differences have to be taken into account because otherwise, we do not know what differences in syntax caused differences in program comprehension. An interesting approach is proposed by APEL ET AL. [ALMK08], who developed an algebra that describes the composition of features. The algebra abstracts from programming languages and thus eliminates the effect of syntax.

Influence on task-related parameters. Besides different programming paradigms and syntax, programming languages can induce differences in task-related parameters. For example, for different languages, different conventions for naming variables or structuring source code exist. For example, in C++, private class variables should have an underscore suffix, but not in Java. Now, when comparing a C++ and Java program, this produces a conflict, because either we can name our variables accordingly and thus introduce a difference in our program, or we can decide for one convention and thus violate the other. Hence, we have to carefully consider the influence of programming languages on task-related parameters, because otherwise, our results would be biased.

An example of comparing maintainability of an object-oriented with a procedural version of a program was conducted by (HENRY ET AL. [HHL90]). They used C and Objective-C as programming languages, which helps making the programs comparable, since Objective-C is an extension of C, allowing object-oriented programming. Due to the similarity of both programming languages, differences in maintainability can be explained by the different underlying programming paradigms.

```haskell
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort less ++ [x] ++ qsort greater
               where
                 less  = [y | y <- xs, y < x]
                 greater = [y | y <- xs, y >= x]
```

(a) Haskell

```java
  public void qsort (int[] arr, int left, int right) {
    int low = left, high = right;
    if (high > low) {
      int mid = arr[(low + high) / 2];
      while (low <= high) {
        while (low < right && arr[low] < mid)
          ++low;
        while (high > left && arr[high] > mid)
          --high;
        if (low <= high) {
          swap(arr, low, high);
          ++low;
          --high;
        }
      }
      if (left < high)
        qsort (arr, left, high);
      if (low < right)
        qsort( arr, low, right);
    }
  }
  public void quickSort (int[] array) {
    qsort (array, 0, array.length - 1);
  }
```

(b) Java

```cpp
template <class T>
void qsort(array<T>& arr, int left, int right) {
    if (left < right) {
        int left_aux(left), right_aux(right);
        T aux = arr[left];
        while (left_aux != right_aux) {
            while (arr[left_aux] < aux && right_aux > left_aux) ++left_aux;
            while (aux < arr[right_aux] && left_aux < right_aux) --right_aux;
            swap(arr[left_aux], arr[right_aux]);
        }
        qsort(arr, left, left_aux - 1);
        qsort(arr, left_aux + 1, right);
    }
}
template <class T>
void quicksort(array<T>& v) {
    qsort(v, 0, v.size() - 1);
}
```

(c) C++

Figure 3.3: Implementation of quicksort algorithm in Haskell, Java, and C++.

Solutions

How can we make sure that programs in different programming languages differ only to a small degree, such that observed differences can only be explained by different underlying FOSD approaches?

In order to deal with this problem, we can use the same methods as for most of the task-related parameters: consult experts, conduct pretests, or assess the opinion of the subjects (cf. Section 3.5). However, it is still difficult to assure that different versions of the same algorithm are comparable, which the Haskell and Java comparison demonstrated.

Hence, comparing different programming languages is a very tedious endeavor, because there are so many aspects that need to be controlled. Furthermore, interpreting results and drawing conclusions is limited and has to be done very carefully. This problem gets larger, the more different the according programming languages are, e.g. Haskell vs. Java. The more programming languages have in common, the easier it is to create comparable programs.

In order to feasibly design experiments that control the influence of programming language, we have to start with small comparisons, for example compare a quicksort implementation in programming languages that have much in common (e.g., C and C++). If sound conclusions can be drawn from the experiment, then more complex programs can be taken into account. This is a useful application for stepwise development: A simple program can be developed and after sound tests and results extended (WIRTH [Wir71]). This allows to test complex programs and draw sound conclusions, because simpler versions of the same programs already lead to reasonable results. A similar idea has be proposed by FLEMING ET AL. [FSD07], who suggest letting subjects implement program families in several experiments, such that with increasing number of experiments, the program families get more complex.

A further useful technique to control the influence of different programming languages are think-aloud protocols, because they can help to interpret our results. Using think-aloud protocols, the way subjects think during understanding a program can be analyzed. This can reveal, how different programming languages are perceived and dealt with by different subjects. However, think aloud protocols are very time consuming and costly, since for every subject, several hours for testing, analyzing and interpreting are necessary (cf. Section 2.2).

In our experiment (cf. Section 5), we compared CPP and CIDE based on the same Java program, so that the only difference of the programs was the kind of annotation. A next step would be to replicate the experiment with a different Java program, different programming language, or a different sample in order to confirm our results.

3.7 Summary

Confounding parameters influence the results of experiments. Hence, it is necessary to identify and control them. Otherwise, results are biased, which leads to wrong conclusions. In order to identify confounding parameters on program comprehension, we consulted the literature and programming experts. Using a scenario, we outlined how the parameters can influence program comprehension.

Parameter	How controllable?	How much influence?
Programming experience	depends	2.89
Domain knowledge	easy	2.56
Intelligence	hard	2.47
Education	depends	2.44
Miscellaneous	easy	-
Training	depends	2.61
Motivation	depends	2.47
Tool support	easy	2.44
Ordering	depends	2.38
Position	depends	2.29
Experimenter	depends	-
Hawthorne	depends	-
Test effects	depends	-
Miscellaneous	easy	-
Structure	depends	3.00
Coding Conventions	depends	2.83
Difficulty	depends	2.67
Comments	depends	2.50
Documentation	depends	2.22
Programming language	hard	-

How controllable: estimate about effort for controlling a parameter, How much influence: arithmetic mean of the expert survey (cf. Table 3.3).

Table 3.6: Overview of confounding parameters.

After showing the need to control the according parameters, we presented several ways to deal with the parameters. There is no optimal way in general, because it depends on several factors, for example, human and financial resources as well as goals and hypotheses of the experiment. In Table 3.6, we show all confounding parameters with a subjective estimate on the difficulty of controlling (column *How controllable?*) and importance (column *How much influence?*) of each parameter. The estimate of the difficulty of controlling depends on how easy the measurement of a parameter can be accomplished. For example, domain knowledge can be assessed by simply asking subjects (easy), whereas intelligence requires time-consuming tests (hard). Usually, controlling parameters depends on the hypotheses and goals of experiments, which can make it easy or hard, such that we cannot give an estimate (resulting in depends). The column *How much influence?* has some missing values, because we either summarized several parameters in one row (*Miscellaneous*) or did not include them in our expert survey (because the influence was due to experiments in general, not program comprehension). In the next chapter, we apply the results of this section to an evaluation of feasibility of comparing FOSD approaches regarding their effect on program comprehension.

Chapter 4

Feasible Scope of Comparing FOSD Approaches

In this chapter, we evaluate the feasibility of measuring the effect of different FOSD approaches on understandability. This is necessary to give an estimate about how time consuming and costly this is and to create an agenda for continuing our work. As the previous chapter showed, designing experiments measuring the effect of different FOSD approaches on program comprehension can get very complicated because of the number of confounding parameters and diversity of FOSD approaches.

We show that it is nearly impossible to state which of the approaches is the most comprehensible. Firstly, we point out the effort of generating a rank list according to understandability in Section 4.1. Secondly, we show in Section 4.2, that the effort for comparing two approaches, even two programming languages, is lower, yet still not manageable within reasonable time and resources. Eventually, in Section 4.3 we show how small the scale is on that statements regarding understandability can feasibly and soundly be made.

4.1 Comparing four approaches

If we want to create a ranking of four FOSD approaches regarding comprehensibility (cf. Section 2.1), our independent variable has four levels: AOP, FOP, CPP, and CIDE. In order to assess the effect on program comprehension, we have to create one program for each FOSD approach and test the understandability of them. An example discussion on how to empirically compare AOP and FOP can be found in APEL ET AL. [AKT07], in which the authors suggest to let programmers implement a graph product line LOPEZ-HERREJON AND BATORY [LHB01] with one AOP and FOP language. However, this approach is too naive, as we show in this section.

4.1.1 Creating a program for each FOSD approach

How can we create one program for every FOSD approach? We could choose one programming language of every FOSD approach, for example AspectJ for AOP, AHEAD for FOP, C for CPP, and Java for CIDE. However, we now would not consider AOP, FOP, CPP, and CIDE in general, but one specific programming language for every approach. Hence, in order to compare not only one aspect of an FOSD approach, but the complete approach, we have to include all programming languages (or at least a representative set) for each approach. There are numerous programming languages to which each approach can be applied, for example Jak (BATORY ET AL. [BSR04]), Java, JavaCC, Haskell, C, C#, XML APEL ET AL. [AKL09], and C++ (APEL ET AL. [ALRS05]) for FOP or FeatherweightJava, Java, C, C#, or Haskell for CIDE (KÄSTNER [KTA08]).

In this section, we optimistically assume that for each approach five programming languages can be selected as representative. This simplifies our discussion, but still suffices to demonstrate the problems. For five programming languages per FOSD approach, we have to create five $\cdot$ four = twenty programs. The first problem lies in creating twenty programs that are comparable regarding all task-related parameters (e.g., structure, comments, documentation). We have to conduct a lot of pretests and/or expert consultations in order to be sure that all 20 programs have a comparable structure, comparable comments and documentation. Furthermore, we have to consider how we deal with rules of programming discourse, for example different naming conventions of variables between Java and C. Should we stick with conventions for the according programming language or should we decide for one of them and thus violate the other?

4.1.2 Recruiting and managing a sufficient number of subjects

Creating twenty comparable programs is not the only problem. Given that we were able to create twenty different but comparable programs, the second problem is to recruit enough subjects to conduct our experiment. If we choose a simple one-factorial design (cf. Table 2.2 on page 31), we have twenty groups. All groups should be large enough, such that we control the influence of confounding parameters like intelligence or education. Since the number of subjects that defines large enough is not specified, (cf. Section 2.3.2.2), we optimistically assume that ten subjects per group are enough. Hence, we would have to recruit 200 subjects for our experiment. If we use students as subjects, we could recruit them via an introductory lecture to computer science, which can easily be addressed to several hundred students (at least in larger universities like the University of Magdeburg). Recruiting 200 experts requires more effort, because we have to contact companies and most certainly pay them for participation, which is very tedious and costly.

However, creating twenty comparable programs and recruiting 200 subjects are not the only problems. Given that we solved both problems somehow, we have to organize conducting the experiment for 200 subjects. For example, if we conduct the experiment using computers, we have to find one room in which all subjects fit in order to control training effects, noise and light level. Furthermore, we have to train and include enough experimenters to control

that subjects are working properly and that everything is according to our experimental plan. Instead of conducting our complete experiment in one session, we could split our experiment to several sessions, such that we can use a smaller room and need lesser experimenters. However, the chance that we introduce more bias due to different noise levels, training, or some randomly occurring event (e.g., power failure), increases this way.

In addition to the effort in conducting our experiment, we have a lot of effort in analyzing the data. We have to check the responses of every subject, evaluate whether the solutions are correct or incorrect, etc. The more manual effort has to be put into the analysis, the longer it takes to evaluate the data of one subject. However, the higher the degree of automated data collection and analysis is, the lower our flexibility about the tasks of subjects. For example, we could let subjects draw a control flow diagram of a program, but evaluating whether it correctly represents the control flow is very time consuming. Think-aloud protocols may be nice to analyze cognitive processes, however they are not feasible for 200 subjects. Hence, the techniques and measures for program comprehension we can feasibly apply for 200 subjects are rather limited.

4.1.3 Reducing number of required subjects (and introducing test effects)

In order to increase feasibility for our comparison, we could decide to use a design with repeated measures (cf. Table 2.3 on page 31). For every FOSD approach, we create one group, so that every subject is required to work with five different programs. If we assume that for each program, subjects need about half an hour, and additionally some time for introduction and breaks, our subjects would need three hours to complete our experiment. During the complete period, we need to keep our subjects motivated. Furthermore, we need to make sure that all subjects are familiar with all programming languages they have to work with. Nevertheless, we now can complete our experiment with about forty subjects (assuming ten subjects per group). Forty subjects and three hours sounds manageable, because they probably all fit in one room, such that we control several confounding parameters and are able to conduct our experiment with few experimenters, but no we introduced test effects.

One problem with repeated measures is that test effects can occur. For dealing with them, we could define a time period between our trials (e.g., several weeks), but then we have to deal with mortality (cf. Section 3.4). In addition, we have not controlled the influence of position and ordering effects. If we want to do so, we need to introduce different orders of the tasks, which means that we need more subjects again. For example, if we create every possible order of our five programs for every FOSD approach, we have $5! = 120$ order for every FOSD approach, which cannot be tested feasibly. However, for continuing our argumentation, we assume we have controlled test, position, and ordering effects somehow.

4.1.4 Assuring generalizability of our results

Creating twenty comparable programs and design our experiment such that 40 subjects suffice *and* test, ordering, and position effects are controlled somehow, are not the only problems, because we still have not considered external validity in our design. If we keep all confounding parameters constant, our internal validity is maximized, yet our results would not be generalizable. In order to increase our external validity without diminishing our internal validity, we have to include the influence of all confounding parameters in our experimental design (cf. Section 2.3.2.2). We pick some of our confounding parameters identified in the last chapter and show why they are problematic to be included in our design. The remaining parameters and according problems are summarized in Table 4.1.

Firstly, we discuss programming experience. In our evaluation, we have only considered students as subjects. Hence, possible results can only be generalized to students, not to professional programmers. In order to increase our external validity, we have to include professional programmers as subjects as well. Professionals are problematic because they cannot be motivated to participate by giving them a bonus for their final exam (because unlike students, they already passed it), but they usually need to get paid. Furthermore, parameters like education and domain knowledge are hard to control, because we have to find persons that have a certain level of programming experience, visited a specific college or university, and are equally experienced with a certain domain. The more requirements we specify, the lesser the number of persons is that meet those requirements. Hence, finding 40 or even 200 appropriate programming experts and motivating them to participate in our experiment is difficult.

Secondly, we consider domain knowledge. Since different program comprehension models depend on the amount of domain knowledge, we have to include different levels of domain knowledge. Otherwise, our results could only be applied to the model of program comprehension we tested in our experiment (e.g., bottom-up program comprehension). Hence, we have to find subjects with no domain knowledge to test bottom-up program comprehension, and with enough domain knowledge to test top-down program comprehension. Additionally, we should integrate a mediocre level of domain knowledge to test integrated models of program comprehension as well. The problem here is to define the amount of domain knowledge such that we can be sure to measure all three different program comprehension models. Otherwise, we could end up with top-down comprehension, although we intended to assess integrated program comprehension.

Thirdly, we discuss tool support. Since tools can support program comprehension, we have to include them in our experiment. Firstly, we could simply include commonly used IDEs in our experiment, for example Eclipse, Microsoft Visual Studio, or JDeveloper[1]. This way, we would assess the influence of those IDEs on program comprehension. However, the functionality those IDEs provide must not necessarily be the same. Hence, we can restrict the features that subjects are allowed to use for program comprehension, however we have to confirm that only the allowed features are used. Besides IDEs, tools like FEAT provide help for comprehending a program and should also be included. We have to make sure that subjects

[1] http://www.oracle.com/technology/products/jdev/index.html

are familiar with according tools, otherwise the results would be confounded with getting used to a new tool or functionality.

The remaining parameters and their problems are summarized in Table 4.1. The column *Levels* contains a suggestion of which levels of the confounding parameter to include in the experimental design. Results from our experiment could be generalized to those levels.

Based on our assumptions before considering external validity (i.e., we have twenty comparable programs, forty subjects and sufficiently controlled the influence of test, position, and ordering effects), we include levels of confounding parameters in our design. If we include only two levels for each confounding parameter, we suddenly have exponentially more experimental conditions to test. For example, we have to test our FOSD approaches with novices and experts, combined with bottom-up and top-down program comprehension, combined with easy and difficult tasks, and so on. For every confounding variable with two levels we include in our experiment, the possible combinations are doubled. Including all 15 confounding parameters (cf. Table 4.1) and four FOSD approaches (because we use repeated measures), we would end up with $2^{15} \cdot 4 = 131,072$ combinations, which means that we have to recruit and handle 1,310,720 subjects in our experiment. Since over one million subjects it is impossible to handle, we cannot compare four FOSD approaches *and* have generalizable results.

If we consider the naive proposal to compare FOSD approaches (i.e., let programmers implement a graph product line with one language per approach), we now see why it most probably will not work: only one programming language per approach is considered, no confounding parameters are mentioned (e.g., programming experience), only one program is considered, and no work to generalize results is discussed. In order to obtain useful results, we restrict ourselves to comparing two programming languages of two different FOSD approaches, which we discuss in the next section.

4.2 Comparing two programming languages

How can we compare two programming languages regarding understandability? In this section, we show that even this seemingly simple question, compared to comparing four FOSD approaches, has no trivial answer. For illustrating problems with comparing two programming languages, we use AspectJ and AHEAD as example, because they are the most common programming languages for AOP and FOP, respectively (cf. Section 2.1.2). Furthermore, we neglect all other confounding parameters, because they are irrelevant for the discussion in this section. Instead focus on the problem of creating two comparable programs of different programming languages.

Before running our experiment, we need to create two programs that are comparable. In Section 2, we showed one AspectJ and AHEAD example (cf. Figure 2.4 on page 11 and 2.5 on page 13). Those examples showed that AspectJ and AHEAD differ considerably, for example regarding the keywords, structure of source code, composition mechanism, etc. Furthermore, there are numerous ways of implementing the same solution in AspectJ and AHEAD (cf. 2.5, page 13 and 2.6, page 14). We have to consider all of them in order to make sound conclu-

Parameter	Levels	Problems
Programming experience	novices, experts	experts are expensive; personal parameters of experts are harder to control than for students
Domain knowledge	no, little, much	assess domain knowledge such that intended program comprehension models are used
Intelligence	different IQs	soundly measure intelligence
Education	different elementary school, high school or college; different age with which subjects started to program	curriculum during education; knowledge subjects gained during education
Gender	male, female	draw representative sample: percentage of males and females?
Training of subjects	no, little, intense	keep training constant; specify little and intense training
Noise level	quiet, noisy	specify kind of noise
Motivation of subjects	motivated to sabotage, not motivated, highly motivated	ensure intended level of motivation
Tool support	no tool support, use text editor, restricted functionality of different IDEs	define set of IDEs to be used, restrict functionality, familiarize subjects
Position and ordering effect	every possible order of tasks and/or program	define small number of tasks/programs to have manageable number of permutations
Code on screen vs. paper	TFT, paper	ensure that subjects are used to code on paper
Structure	unusual, usual	ensure comparable structure in different programs
Coding conventions	violated, ensured	different coding conventions for different programming languages
Difficulty	easy, difficult	ensure intended levels of difficulty
Comments	confusing, none, helpful	ensure intended nature of comments
Documentation	confusing, no, helpful	ensure intended nature of documentation; make sure subjects use documentation

Table 4.1: Overview of problems with confounding variables.

sions, otherwise we probably picked a badly understandable implementation in AspectJ, while the given AHEAD implementation is easier to comprehend. With another implementation of the same problem, the effect could be reversed. In this case, we would not compare AspectJ and AHEAD, but the kind of implementation, which would render our results useless. Additionally, we only would compare the implementation of *one* problem. Maybe for some other problems, the outcome would be reversed?

Hence, comparing even two programming languages regarding their understandability is not trivial. We have to be sure to have comparable programs and we do not measure the effect of something not intended, for example different structuring, comments, or coding conventions. Creating two comparable programs in different programming languages is harder the more both programming languages differ. Although AspectJ and AHEAD both have Java as underlying programming language and thus much in common, they still differ considerably.

We could implement a problem in the most understandable way for both programming languages. In order to determine which the most understandable implementation is, we can let experts of a programming language implement a problem with the requirement that it should be the most understandable. We can compare the implementations of all experts and choose the most frequent one. We could also let experts rate several implementations of one problem regarding understandability. Instead of letting experts create or rate a program regarding understandability, we can test other criteria, for example that an implementation should be the most simple one in experts' opinion or shortest in terms of LOC. In any case, we know that both programs have something in common (understandable, simple, or shortest), which is specific for each programming language. Hence, observed differences in program comprehension cannot be attributed to those criteria, but to other differences of the programs. However, we cannot isolate which of the numerous other differences caused a difference in program comprehension.

Having demonstrated the comparing even two programming languages is difficult, we discuss feasible comparisons regarding FOSD approaches.

4.3 Realistic comparison

The previous explanations showed that a high degree of external validity requires a large number of subjects. Hence, in realistic comparisons, we have to abandon the goal of creating generalizable experiments. Only if we are sure how several parameters influence program comprehension, we can create experiments with higher external validity. At the current time, however, the influences of most parameters in relation to FOSD approaches on program comprehension are unknown, so that we have to start from scratch. Thus, we need focus on internal validity for creating our experiments.

First of all, we restrict our experimental design to one independent variable with two levels. This reduces the number of subjects we need. Then, we keep all personal parameters constant, which diminishes our external validity, but also the number of subjects we need. Next, we choose two levels of our independent variable that are rather similar. For example,

```
1 public class Stack {
2   /* ... */
3   // #ifdef TOP
4   public Element top() {
5     return elements.getFirst();
6   }
7   // #endif
8 }
```

(b) CPP

```
1 public class Stack {
2   /* ... */
3   Element top() {
4     return elements.getFirst();}
5   }
6 }
```

(a) CIDE

Figure 4.1: CPP and CIDE implementation of the stack example.

CPP and CIDE can be applied to several programming languages, so that we can simply use the same program, but with different annotations when comparing the effect of CPP and CIDE. In Figure 4.1 we show the implementation of the method `top` of the stack example (cf. Section 2.1.3). We can see that both implementations only differ in their kind of annotation, but everything else is identical. Hence, any observed difference in program comprehension can solely be caused by the different kind of annotation.

Of course, we still have to make sure that our results can be generalized (e.g., by testing more than one implementation of the same algorithm or confirm results with larger programs). Nevertheless, being able to use the same programming language helps us to create two versions of source code that differ only in few aspects, such that we know that observed differences in program comprehension are caused by those few aspects. The more different aspects we include in our comparison, the lesser we know why we observed a difference, because the difference could be caused by one, some, all, or a mixture of those aspects.

As another example for similar levels we consider the effect of a few keywords on understandability, for example an inter-type declaration in AspectJ vs. method introductions with class refinements in AHEAD, whereas everything else is kept constant. In Figure 4.2, we show how two such comparable programs can look like with the stack example. The difference is that in AspectJ, the keyword `aspect` is used, whereas in AHEAD, `refines class`. Furthermore, the aspect is called `Top`, while the refinement `Stack`. Additionally, the way of specifying that the method `top` belongs to the class `Stack` differs (AspectJ: `Stack.top`, AHEAD: class designator is `Stack`). Hence, both source code fragments provide the same amount of information. Now, any observed differences in program comprehension could now be attributed to the small differences of both source code fragments. However, we cannot be sure whether this is caused by the different keywords, designators, or specifying the belonging of the method `top`. We could modify the method definition in AHEAD in line two to `public Element Stack.top()`, such that observed differences in program comprehension can only be caused by different keywords and designators.

Now, the scope of our hypotheses is rather small, but we can design experiments without needing a tremendous amount of resources to realize them. Furthermore, due to the high degree of internal validity, we can draw sound conclusions from our results. Those sound

```
1 aspect Top {
2   public Element Stack.top() {
3       return elements.getFirst();
4   }
5 }
```

(b) AspectJ

```
1 refines class Stack {
2   public Element top() {
3     return elements.getFirst();
4   }
5 }
```

(a) AHEAD

Figure 4.2: AspectJ and AHEAD implementation of the stack example.

conclusions can then be used to broaden the scope of our hypotheses and create a body of knowledge, which helps us to understand the relationship between FOSD approaches, confounding parameters, and program comprehension.

4.4 Agenda

The consequence we can draw from our explanations is that the scope of feasible *and* sound comparisons is rather small. Hence, it could take decades until we have gathered enough information to make a sound statement about the effect of different FOSD approaches on program comprehension. In order to considerably reduce the amount of time, it is necessary to establish a research community to combine our resources. Based on the results of experiments with high internal validity, we can generalize our results by conducting experiments with a higher degree of external validity, which allows us to develop a theory about the understandability of FOSD approaches.

In summary, we suggest the following proceeding in order to gather a sound and exhaustive knowledge base about the effect of different FOSD approaches on program comprehension in a reasonable amount of time:

- Establish a research community.
- Design experiments with a small scope and high degree of internal validity.
- Confirm hypotheses of experiments by replication.
- Increase external validity of experiments.
- Integrate results of experiments in a theory about the effect of FOSD approaches on program comprehension.

4.5 Summary

In this chapter, we showed that comparing all FOSD approaches at once is practically impossible due to the large number of confounding parameters. Comparing even two programming languages is very tedious and could take years of empirical research. However, since we have no knowledge about the effect of FOSD approaches on program comprehension, we have to start with experiments that have a high degree of internal validity. This allows us to draw sound conclusions about effects on program comprehension. Having established some basic knowledge about the effect of FOSD approaches on program comprehension, we can design experiments with greater external validity. Since this could take decades with only one researcher group, it is imperative to develop a community, such that we can combine our resources and considerably speed up the process.

In the next chapter, we demonstrate our explanations on an experiment, in which we compare the effect of CPP and CIDE on program comprehension of a Java program.

Chapter 5

Experiment

In this chapter, we present an experiment that illustrates the small scope of feasible comparisons of FOSD approaches we derived in the last chapter. The objective of this experiment is to assess the effect of different annotations on program comprehension. We start this chapter with a summary of our experiment and results to give an overview of the most important aspects. Readers not interested in details can skip the rest of this chapter. After this overview, we describe our experiment according to the experimental stages we introduced in Chapter 2.3: First, we define our independent and dependent variables as well as our hypotheses in Section 5.2. We continue in Section 5.3 with the design stage, in which we explain how we controlled confounding parameters. In Section 5.4, we report how we executed the experiment. We analyze our data in Section 5.5. In Section 5.6, we interpret the meaning of the results for our hypotheses, discuss threats to validity and present some ideas for future work based on our results. We summarize this chapter in Section 5.7.

5.1 The experiment in a nutshell

In this section, we present the important aspects of our experiment. This helps to get a quick overview of the experiment.

Objective. The goal of our experiment is to assess whether the kind of annotation (text, as provided by CPP vs. colors, as provided by CIDE) has an effect on program comprehension. We restrict our comparison to a single programming language in order to design a feasible experiment within the limits of our resources. We choose Java as programming language, because it is common to two both, CPP and CIDE. Furthermore, we keep most of the confounding parameters constant (e.g., we eliminate tool support), which maximizes our internal validity, such that we can draw sound conclusions from our result.

We evaluate the kind of annotation by comparing program comprehension of two experimental groups: one group works with CPP annotations, the other group with CIDE annotations. This allows us to explain observed differences between both groups with the different

kind of annotations.

Subjects. As subjects, we decided to use students of the University of Passau that were enrolled in the course *Modern Programming Paradigms (German: Moderne Programmierparadigmen)*, because students learned about FOSD approaches and how to work with some of them (including CPP (Munge) and CIDE). This assures that subjects are equally skilled regarding FOSD approaches in general.

A few weeks before the actual experiment, we measured programming experience of subjects with a carefully designed questionnaire. We assessed the experience subjects had with several programming languages and asked for familiar domains. Based on the answers subjects provided, we created homogeneous groups regarding programming experience, decided to use Java as programming language (because all subjects were at least moderately familiar with it), and decided to use mobile devices as unfamiliar domain (because none of the subjects were familiar with it).

Source code. Having specified the programming language and domain for our experiment, we looked for existing SPLs (i.e., Java as programming language, mobile devices as domain), so that we can test the effect of different annotations. We decided to use the MobileMedia SPL described in (FIGUEIREDO ET AL. [FCM+08]), because, beside fulfilling our criteria, it is code reviewed by programming experts, uses CPP annotations to realize variability, and is neither too small nor to large (28 classes, 3800 LOC). It handles multi media on mobile devices. For creating a CIDE version of this SPL, we deleted all CPP comments and annotated source code belonging to features with according background colors. Since the CPP and CIDE version are identical except for the kind of annotation, we can explain possible differences in program comprehension with the kind of annotation.

In Figure 5.1, we show one class of (a) the CPP version and the equivalent class of (b) the CIDE version.

Tasks. A further advantage of using identical programs is that we could use the same tasks for measuring program comprehension for both versions. Besides a warming up task, in which subjects should familiarize with the experimental setting, we created two different kinds of tasks. Firstly, subjects should locate feature code, which we assessed by (1) letting them complete a collaboration diagram template of the source code and (2) locate statements that described feature interactions. Secondly, we created four maintenance tasks introducing bugs into the program, which subjects should fix. The bugs in the program were introduced by us and located in feature code (which was necessary for assessing the effect of different annotations, because only feature code is annotated).

Results. We measured the response time of subjects and whether a solution of a task was correct. Three main results can be described. Firstly, for locating feature code, subjects with

```
// #ifdef includeFavourites
public void toggleFavorite() {
  this.favorite = ! favorite;
}

/**
 * @param favorite
 */
public void setFavorite(boolean favorite) {
  this.favorite = favorite;
}

/**
 * @return the favorite
 */
public boolean isFavorite() {
  return favorite;
}
// #endif

// #ifdef includeCountViews
public void increaseNumberOfViews() {

}

/**
 * @return the numberOfViews
 */
public int getNumberOfViews() {
  return numberOfViews;
}

/**
 * @param views
 */
public void setNumberOfViews(int views) {
  this.numberOfViews = views;
}
// #endif
}
```

```
public void toggleFavorite() {
  this.favorite = ! favorite;
}

/**
 * @param favorite
 */
public void setFavorite(boolean favorite) {
  this.favorite = favorite;
}

/**
 * @return the favorite
 */
public boolean isFavorite() {
  return favorite;
}
public void increaseNumberOfViews() {

}

/**
 * @return the numberOfViews
 */
public int getNumberOfViews() {
  return numberOfViews;
}

/**
 * @param views
 */
public void setNumberOfViews(int views) {
  this.numberOfViews = views;
}
}
```

Figure 5.1: Comparison of CPP (left) and CIDE (right) version of SPL in our experiment.

colors were significantly faster than subjects with text based annotations (on average by 30 %), because they could benefit from preattentive perception of colors (cf. Section 2.1.3.2).

Secondly, we found no differences in response time between text-based and color-based annotation for maintenance tasks, except for the last task, in which subjects with the text-based version were significantly faster (by 37 %). Since in maintenance tasks, source code needs to be analyzed on a textual basis regardless of the annotations, it is not surprising that response times did not differ for three of the four maintenance tasks. The problem with the last task was that the according class was entirely annotated with red, which was hard to look at for a certain amount of time (according to some subjects). This could explain the faster response time for subjects that worked with the text based version.

For the number of subjects that provided a correct solution for a task, we found no differences for any tasks between both groups, indicating that the kind of annotation has no effect on correctness of program comprehension. Besides those results, we showed how a feasible experiment can be designed. After this overview, we describe our experiment in detail in the next section. We start with the process of objective definition.

5.2 Objective definition

Before we can start to design our experiment, we need to define our variables of interest and our hypotheses. First, we explain our independent (Section 5.2.1) and dependent (Section 5.2.2) variables. Then, we present our hypotheses in Section 5.2.3.

5.2.1 Independent variable

As we discussed in Chapter 2.3, our independent variable is the FOSD approach. For our experiment, we choose two levels, CPP and CIDE. We choose both levels, because the effect of colors for annotations is controversially discussed and thus would be interesting to evaluate (KASTNER ET AL. [KTA08]). Furthermore, CPP and CIDE are language independent and can be applied to various programming languages. This helps us to create comparable programs, because we can choose one programming language to which both approaches can be applied. However, we do not assess CPP and CIDE in general, but focus only on few aspects. Firstly, we decided to include only one programming language in our experiment, because more would not be feasible. Secondly, we use Antenna as representative for CPP, which constitutes a preprocessor for Java ME[1]. Furthermore, we eliminate tool support, so that we do not test CIDE, but only its annotations (cf. Section 5.3.2). However, for convenience, we use CPP and CIDE to refer to the levels of our independent variable.

The decision of the concrete programming language depends on two aspects: Firstly, of course, CPP and CIDE must be applicable to this programming language. Secondly, our subjects should be familiar with the programming language, so that they can concentrate on the experimental tasks and do not have to learn a new programming language, additionally.

5.2.2 Dependent variable

We decided to measure bottom-up program comprehension, because it is easier to find an unfamiliar domain for all subjects than a domain all subjects are familiar with. This restricts our external validity, but we control the influence of domain knowledge and thus increase our internal validity.

In order to measure program comprehension, we used static and maintenance tasks, because they are reliable measures (cf. Section 2.2). Furthermore, for the maintenance tasks we created bugs that occurred during runtime, with which we intended to force mental simulation of the program. This way, we intend to further increase accuracy of our measurement (cf. Section 2.2). We measured the correctness of answers and time to complete the tasks (referred to as *response time*). We decided against think-aloud protocols, because they are not feasible for our thesis, because they are very time consuming and costly (cf. Section 2.2). A detailed description of the tasks can be found in Section 5.3.

Having defined our variables and how we plan to measure them, we can specify in our hypotheses the expected results.

[1] http://java.sun.com/javame

5.2.3 Hypotheses

In this section, we discuss our hypotheses. We state every hypothesis and explain, why we expect the specified relationship between our variables.

> For static tasks, CIDE annotation speeds up bottom-up program comprehension compared to CPP annotation.

In our static tasks, subjects have to locate feature code (Section 5.3), which is annotated either text based (CPP) or with background colors (CIDE). In order to locate feature code in CPP, subjects have to locate *#ifdef* statements analyze their content. For locating feature code in CIDE, subjects have to locate background colors, which means that every statement that has a different background color than white, is feature code. Since colors constitute basic features, they are processed preattentively and thus faster than the meaning of text (GOLDSTEIN [Gol02], p. 165). Hence, we assume that for the static tasks, CIDE annotation has a positive effect on response time.

> For maintenance tasks, there are no differences in response time between CIDE annotation and CPP annotation with bottom-up program comprehension.

For maintaining software, subjects have to analyze source code on a textual basis. A programmer has to understand why a certain problem occurs and how it can be solved or how he can extend a program with a required functionality. This includes locating code fragments and analyzing source code on a textual basis. In the previous tasks, subjects already located feature code and got an overview of the program. Hence, the main part of a maintenance task is analyzing textual information. Thus, it should be irrelevant how source code is annotated, which is why we assume that there are no differences in response times for maintenance tasks.

> There are no differences in the number of correctly solved tasks between CPP annotated and CIDE annotated source code with bottom-up program comprehension.

Since both kinds of annotation provide information about feature code and the feature to which it belongs, it should make no difference for successfully completing a task. Both kinds of annotation provide the same amount of information, the only difference is how it is represented. Hence, we assume that there is no influence on successfully completing a task, although it might take more time to complete a task.

> Subjects that worked with the CPP version estimate their performance with the CIDE version better than subjects that worked with the CIDE version and estimate their performance with the CPP version.

This hypothesis is based on observations made by HENRY ET AL. [HHL90] and DALY ET AL. [DBM+95]. Although subjects performed better with OOP, they estimated that OOP decreased the performance. We assume a similar effect, because colored annotations provide

different colors for each feature and a better separation of feature code at first sight. In CPP, the annotations have to be analyzed text based before an according feature can be identified. Hence, we assume that both groups differ in their performance with the other source code version.

In the next section, we show how we designed our experiment so that we can test our hypotheses.

5.3 Design

In this section, we explain how we controlled the influence of confounding parameters. We start with personal parameters in Section 5.3.1, continue with environmental parameters in Section 5.3.2 and task-related parameters in Section 5.3.3, and conclude with programming language in Section 5.3.4. Secondly, we describe the tasks we created to assess program comprehension in Section 5.3.5.

5.3.1 Controlling personal parameters

We explain for every parameter, how we plan to control its influence. We traverse the list of personal parameters in the same order as in the previous chapter.

Programming experience. Programming experience influences program comprehension, because the more experience a programmer has, the more he can profit from his experience when understanding a program (cf. Section 3.3.1). The understanding of programming experience is rather diverse in the literature (PRECHELT ET AL. [PUPT02], KO AND UTTL [KU03], HUTTON AND WELLAND [HW07]). We found no commonly used definition or questionnaire to assess it. Hence, we developed and tested a questionnaire that measures programming experience according to aspects we found in the literature (e.g., years of programming, number of large projects). We tested this questionnaire with students of the University of Magdeburg and improved it according to the results. The complete questionnaire we used for our experiment and the coding of questions can be found in Tables A.1 (page 135) and A.2 (page 136) in the appendix. We used a five point Likert scale (LIKERT [Lik32]) to assess the answers of our subjects. We summed up all answers of subjects, such that the larger the sum is, the higher programming experience is.

Although we carefully designed and tested this questionnaire, we do not know for certain that it accurately measures programming experience. However, we are sure that we measured all relevant aspects for our experiment, such that we assume that for our purpose the questionnaire suffices. Nevertheless, it is an interesting issue for future work to create a questionnaire that measures programming experience, so that we do not need to design one ourselves for every experiment we conduct.

We applied this questionnaire six weeks before the actual experiment. In order to assure anonymity of the subjects and to be able to match the programming experience score to the

answers of subjects in our experiment, we let subjects generate a code word. This code word consists of six letters and was created by six personal questions like 'What is the first letter of your mother's first name'. This way, only a subject knows his code word, but not the experimenters. In order to create homogeneous groups according to programming experience, we used matching as control technique according to the score in our questionnaire (cf. Section 2.3.2.2).

Domain knowledge. The amount of domain knowledge influences the process of program comprehension, such that the more domain knowledge a programmer has, the more likely it is that he uses a top-down approach to understand a program, which is more efficient than a bottom-up approach (cf. Section 3.3.2). We decided to measure bottom-up program comprehension, since we can easier determine a domain all subjects are unfamiliar with than a domain with which all subjects are equally familiar with. In order to determine unfamiliar domains, we asked subjects to list all domains they have experience with. We did this at the same time as we measured programming experience. We then selected the domain of mobile devices, because none of our subjects mentioned it.

Intelligence. Intelligence can influence program comprehension, because of a larger memory capacity or better analyzing skills (cf. Section 3.3.3). Unfortunately, we had no resources to measure intelligence and we found no substitute measure we could use. For example, the average grade of the high school diploma as is not suitable, because the time intervals to the high school diploma varied for our subjects. Hence, we used randomization as control technique: We assumed that our sample was large enough, so that with matching according to programming experience our samples are homogeneous according to intelligence and thus this threat to internal validity is eliminated.

Education. Different schools can have different focus of interest in their curriculum, so some subjects may have learned several programming languages, whereas others only learned one on a rudimentary level (cf. Section 3.3.4). Our subjects either studied Computer Science or Internet Computing at the University of Passau. All were enrolled in the course *Modern Programming Paradigms (German: Moderne Programmierparadigmen)*, an advanced programming course. Several other basic programming courses should have been completed to enroll in this course (i.e., Programming I and II, Software Engineering). Hence, we can assume that all subjects have about the same education regarding programming.

Miscellaneous. In our sample, we had four females and one color blind subject. Matching according to programming experience assigned two females to each group and the colorblind subject to the CPP version. Hence, our groups are homogeneous according to gender and we eliminated one threat to internal validity by letting the colorblind subject not work with the CIDE annotation.

In the next section, we show how we controlled the influence of environmental parameters.

5.3.2 Controlling environmental parameters

For explaining how we controlled the influence of environmental parameters, we proceed the same way as in the last section.

Training of subjects. Well prepared subjects can outperform unprepared subjects just because of their preparation (cf. Section 3.4.1). In order to control the effect of training, we refreshed subjects' knowledge of CPP and CIDE with familiar source code examples they learned in the course they were enrolled in. Specifically, we used the class `Edge` of a graph product line (LOPEZ-HERREJON AND BATORY [LHB01]), which was used running example in the lecture. We demonstrated the implementation of a feature *Weighted*, which enables weighted edges, with CPP and CIDE. In the same way, we explained feature interactions, for which we additionally used the feature *ShortestPath*, which implements finding a shortest path between two nodes and requires the feature *Edge*. The introduction was held in one room for all subjects. The slides we used can be found at http://www.fosd.de/exp_cppcide. Furthermore, we can assume that all subjects are equally skilled with FOSD, SPLs, the use of CPP (Munge, a preprocessor[2] for Java that works like any other preprocessor) and CIDE, etc., because they all attended the same programming course.

Noise during experiment. Since the noise level can influence program comprehension, we conducted the experiment in one room, so that all subjects were exposed to the same normal noise level.

Motivation of the subjects. Highly motivated subjects tend to perform better than subjects that are not motivated at all (cf. Section 3.4.2). Subjects were required to participate in our experiment to complete the course they are enrolled in. This could negatively affect motivation. In order to motivate our subjects for participating in our experiment, we raffled an Amazon gift card (30 Euros, sponsored by METOP GmbH). Additionally, we asked subjects for each task how motivated they were on a five point Likert scale. In order to analyze the effect of motivation, we check whether there is a significant difference between CPP annotations and CIDE annotations. If we kept motivation in both groups comparable, we should obtain no significant differences.

Tool support & source code on screen vs. paper. Tool support can benefit program comprehension, such that subjects that are allowed to use an IDE can better comprehend a program than subjects that have to use a text editor (cf. Section 3.4.3). In our experiment, we eliminated tool support to solely compare the kind of annotations. One way to do so would be to present the source code on paper. However, since colors on screen look differently than colors on paper and CIDE uses colors for annotation, we chose to display the source code on screen. Due to the elimination of tool support (and restriction to one programming language), we do not test CPP and CIDE, but only the annotations both approaches use (cf. Section 5.2.1).

[2] http://weblogs.java.net/blog/tball/archive/munge/doc/Munge.html

In order to control the influence of tool support, we created HTML pages of every source code file. Subjects used the Mozilla browser[3] to display the source code and were forbid to use the search features (otherwise, CPP subject would have an advantage, because they could search for feature names). Links to all HTML files were displayed on the left of the screen (similar to the package explorer in Eclipse). This way, we could exclude the influence of tool support, however at the cost of external validity, because we further stripped down CPP and CIDE to the kind of annotation, not to kind of annotation mixed with tool support. An example of the HTML files subjects worked with is shown in Figure 5.2.

Position & ordering effect. Position and order of tasks can influence a subjects' performance, because he has to get used to the experimental setting or becomes fatigue after several hours (cf. Section 3.4.4). We used a warming up task, in which subjects counted the number of features. This way, subjects had to look at every source code file and could familiarize with the experimental setting, so that subjects are warmed up when starting to work at the actual tasks and our measurement would not be biased with subjects getting familiar with the experimental setting. Our sample was too small to create different orders of the tasks. Instead, we presented the tasks in ascending order according to difficulty for both source code versions, which we defined by evaluating our tasks before the experiment with students from the University of Magdeburg. If position or ordering effects occur, they affect both groups similarly.

Effects due to experimenter. The experimenter can influence the behavior of subjects and thus the results of our experiment (cf. Section 3.4.5). In order to control for these effects, we kept our introduction neutral and communication with the subjects to a minimum.

Hawthorne. The Hawthorne effect describes that behavior of subjects is influenced simply by the fact that subjects know that they are observed (3.4.6). Since the Hawthorne effect is especially problematic for measuring behavior, but our focus was measuring performance, we let all subjects know they participated in an experiment. Hence, the performance of all subjects is influenced by their participation in the same way.

Test effect. We chose a simple design (cf. Table 2.2 on page 31). Since we had no repeated measure, no test effects could occur.

Miscellaneous. The experiment was conducted on Linux systems with 19" TFT displays for all subjects. Subjects were instructed to use the Mozilla browser.

5.3.3 Controlling task-related parameters

Since we used one technique for controlling the influence of most of the task-related parameters, we proceed differently than in the last sections. First, we explain how we controlled the

[3] http://www.mozilla.org

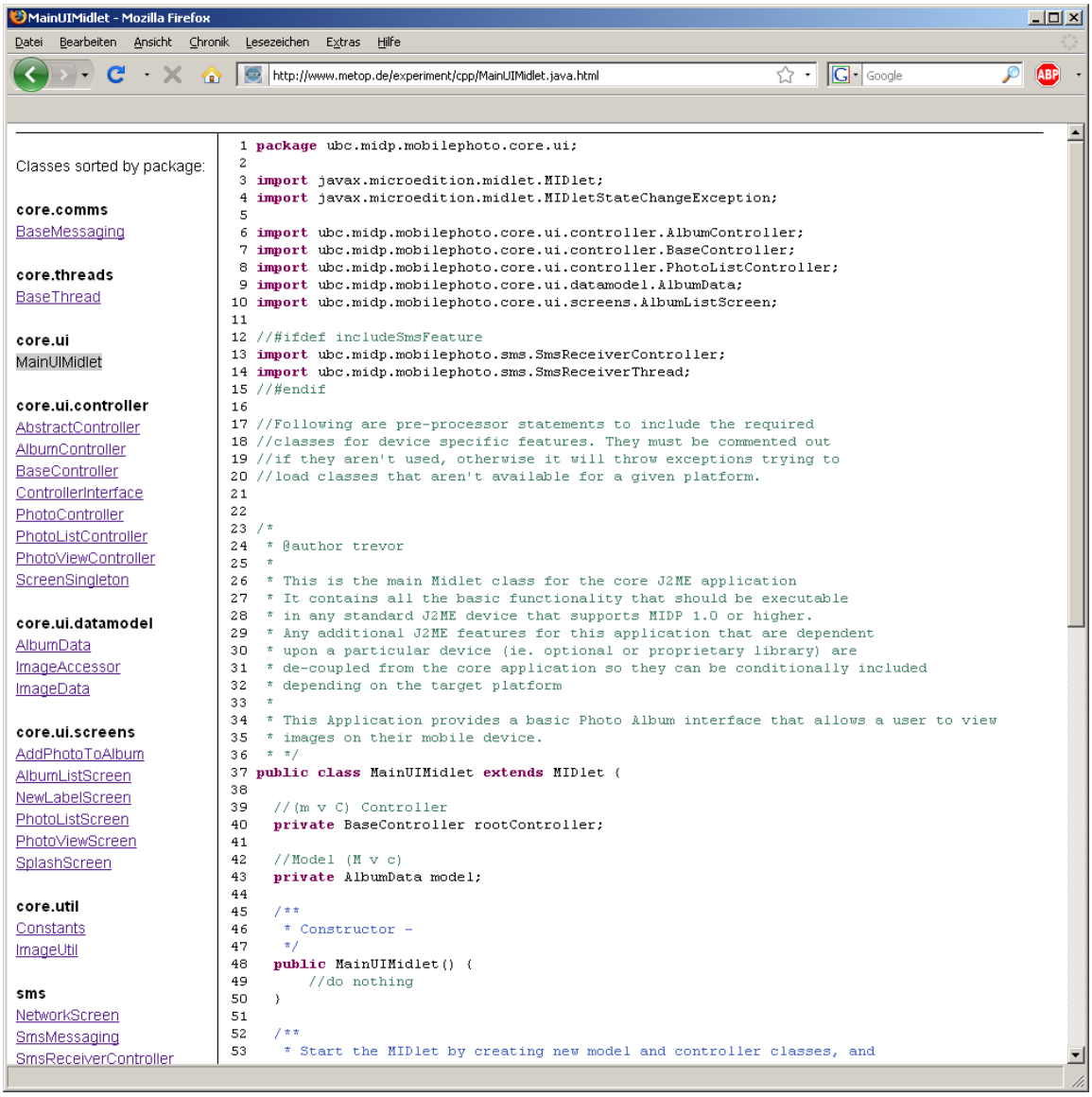


Figure 5.2: Sample HTML file with preprocessor statements in Lines 12 and 15.

influence of structure, coding conventions, difficulty, comments, and documentation. Then, we discuss how we assured that we have comparable tasks.

Structure, coding conventions, difficulty, comments, documentation. In order to control task-related parameters, we used a code reviewed Java SPL for mobile devices (FIGUEIREDO ET AL. [FCM+08]). The authors developed the source code of this SPL with eight scenarios, while with every scenario, more features were added. We used the source code resulting from the sixth scenario. We decided to use this SPL and scenario for several reasons:

- All subjects are unfamiliar with the domain (mobile devices), which ensures bottom-up program comprehension.
- The SPL was implemented in Java, which was the most familiar programming language for all subjects.
- The SPL was implemented with Antenna, a preprocessor developed for wireless Java applications, which is one level of our independent variable.
- The SPL was code reviewed, which helped us to control most of the task-related parameters.
- We decided to use the sixth of eight scenario, because the size of the SPL in terms of LOC and number of features seems suitable, so that subjects neither got lost in numerous files nor understood the complete SPL after completing the first task.

The SPL describes applications that modify photos, music, and video on mobile devices. In the sixth scenario, it consists of four features. For a better overview, we show the feature diagram of this version in Figure 5.3. The *Sms* feature (*Send/Receive Photo via SMS*) allows to send and receive multi media via SMS. The *Favourites* feature (*Favourites*) allows to set multi media as favorites. The *CopyPhoto* feature (*Copy Photos*) handles copying multi media, for example in different albums, and *CountViews* (*Count Views of Photo*) counts the number of views of a media.

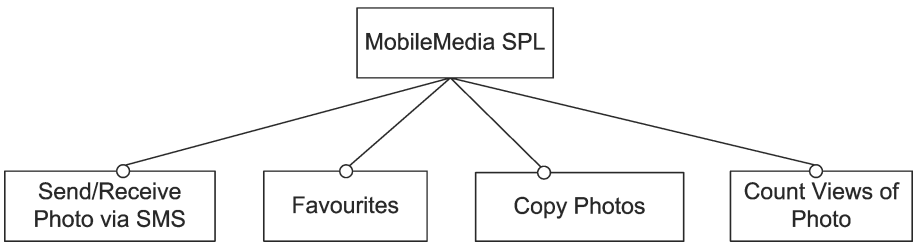


Figure 5.3: Feature diagram of the MobileMedia SPL in the sixth scenario.

By using a code reviewed program, we assured that structure, coding conventions, difficulty, comments, and documentation do not violate rules of programming discourse. Since

the version of the SPL described in FIGUEIREDO ET AL. [FCM+08] was already annotated with CPP statements, we could easily create CIDE annotations by deleting all lines that described CPP statements and annotate all code that was between those statements with colors accordingly[4].

Difficulty of the task. Since we used one program, but with different kinds of annotation, we could use the same tasks for both versions, thus controlling the effect of difficulty. In order to analyze whether subjects perceived difficulty differently, we let subjects estimate the difficulty of each task on a five point Likert scale. Similar to subjects' motivation, an observed difference between both groups could indicate an influence on subjects' program comprehension. Although the tasks are identical, they could be perceived as more difficult in conjunction with one version of the source code, which is why we need to consider this influence.

5.3.4 Programming language

Controlling the influence of programming language posed no problem, because we decided to include only one programming language in our experiment. Hence, the influence of programming language is kept constant. In the next section, we describe how we developed the tasks to measure program comprehension.

5.3.5 Tasks

Since program comprehension is a latent variable, we need to find indicators to assess it. DUNSMORE AND ROPER [DR00] have evaluated the reliability of several measures for program comprehension, on which we base the selection of tasks. In this section, we describe the measures we used for assessing program comprehension.

We created one warming up task, two static tasks (i.e., tasks regarding structural analysis, cf. Section 2.2), and four maintenance tasks (i.e., fixing bugs or improving a program, cf. Section 2.2). We tested the tasks in a preliminary test, in order to assess the difficulty and to estimate the time subjects need to complete the task. We describe the resulting tasks in the same order as we applied them in the experiment: warming up task, static tasks, maintenance tasks.

In the warming up task, subjects should count the number of different features, which required them to look at every source code file. This way, subjects could familiarize with the source code and the experimental setting (e.g., navigating between files). Hence, when starting with the subsequent tasks, subjects could concentrate on the tasks solely, not on getting used to the experimental setting additionally. Since program comprehension is confounded with familiarizing with the experimental setting for the warming up task, we did not include it in our analysis (cf. Section 3.4.4).

[4]The CPP variant can be found at http://www.metop.de/experiment/cpp/MainUIMidlet.java.html, the CIDE variant at http://www.metop.de/experiment/cide/MainUIMidlet.java.html.

Static tasks. In the second task (S1)[5], subjects should identify the roles of classes in each feature and mark them in a template of a collaboration diagram (cf. Section 2.1.2.1), which was printed on paper. We show an excerpt of this template for both annotations in Figure 5.4. The complete collaboration diagram with the correct solution can be found in Figure A.1 on page 137 in the appendix. We analyzed three different measures: (1) the number of correctly identified roles, (2) the number of incorrectly identified roles, and (3) the number of missed roles. We created this task, so that subjects had to deal with annotated source code. This way, we can attribute differences in the number of correct solutions or response times to the kind of annotation. For the same, reason, we created the next task.

In the third task (S2), subjects should locate *feature interactions*, which describe that a source code fragment belongs to more than one feature. They occurred between the two features *CopyPhoto* and *SmsFeature* in three different classes: `PhotoController`, `PhotoViewController`, and `PhotoViewScreen`. In Figure 5.5, we show the feature interaction in class `PhotoController`. For identifying both features correctly, one point was given. For identifying the correct classes, one point per class was given, so that the maximum number of points was four. Subjects naming the correct classes but the wrong features did not receive any points, because we cannot be sure that they indeed identified feature interactions.

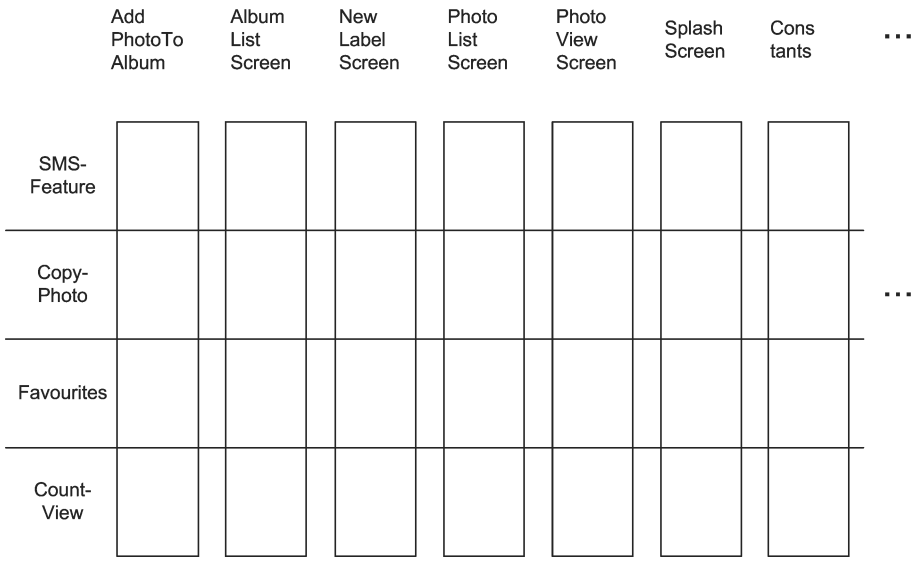
Maintenance tasks. The next four tasks were maintenance tasks, for which we introduced bugs into the program. We tested several bugs with students from the University of Magdeburg and found that certain bugs, for example a deleted statement within a method that consisted of numerous statements or an infinite loop, were very hard for subjects to find. In the end, we created four bugs, which took subjects from our preliminary test about one hour to complete, such that we knew that tasks were not too difficult. All bugs occurred in feature code, so we can test the effect of different annotations. All variants in which the bug occurred were part of the bug description, so that subjects knew the features in which to look. Furthermore, all bugs occurred during runtime, not compile time, which forced subjects to examine the control flow of the program.

The resulting four tasks were used for our experiment, because they fulfill the previously described requirements: (1) the bugs are located in feature code, (2) the bugs occur during runtime, and (3) the tasks are not too difficult for subjects to find in a reasonable amount of time. This assures that we soundly assess program comprehension of our subjects.

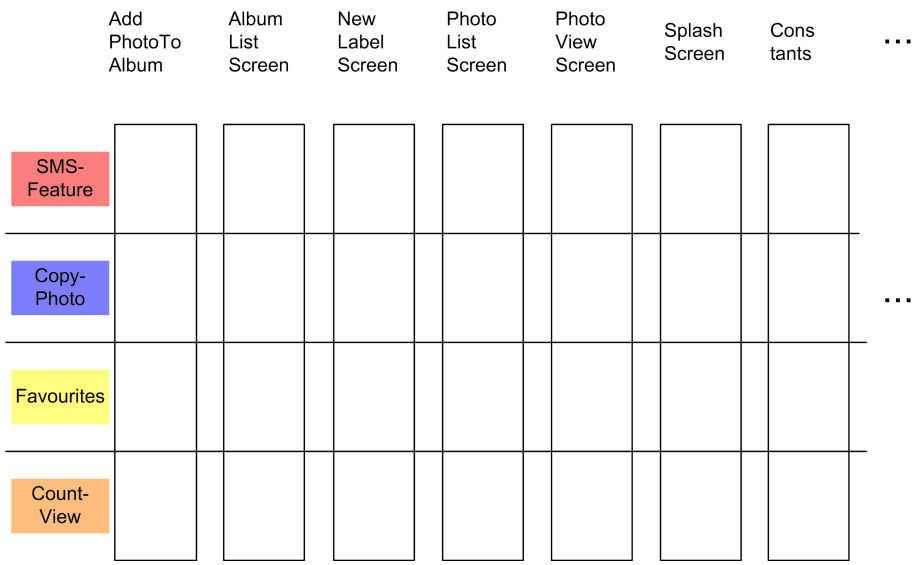
For each task, subjects got a bug description, and should locate the bug (i.e., name class and method), explain why it occurs, and suggest a solution. We used all three pieces of information to rate whether a task was successfully completed or not. This way, we had more information to rely on in case we were not sure whether subjects identified the problem. We analyzed the solutions of each subject and consulted a programming expert, who evaluated whether the answers of the subjects were correct (i.e., fixed the bug).

In the first maintenance task (M1), subjects got the following bug description: ‘If pictures should be sorted by views, they are displayed unsorted anyway. Feature, in which the bug

[5]We use those abbreviations for all further tasks. S means static, M means maintenance.



(a) CPP template



(b) CIDE template

Figure 5.4: Template of collaboration used in our experiment.

```
1  public class PhotoController{
2    /* ... */
3    //#if includeCopyPhoto || includeSmsFeature
4    PhotoViewController controller = new PhotoViewController(midlet, getAlbumData(),
         getAlbumListScreen(), name);
5    controller.setNextController(nextcontroller);
6    canv.setCommandListener(controller);
7    nextcontroller = controller;
8    //#endif
9    /* ... */
10 }
```

Figure 5.5: Feature interaction (shared code) in the class PhotoController.

occurs: CountViews'. The bug was located in the class `PhotoListController`, in which the method `bubbleSort` existed, but was not implemented. We show the bug in Figure 5.6, in which we show the according CPP annotated source code (Lines 6–8).

```
1  public class PhotoListController{
2     /* ... */
3     //#ifdef includeCountViews
4     /* ... */
5     public void bubbleSort(ImageData[] images) {
6       System.out.print("Sorting by BubbleSort...");
7       // TODO implement bubbleSort
8       System.out.println("done.");
9     }
10    //#endif
11 }
```

Figure 5.6: Bug for M1: `bubbleSort` is not implemented.

In the second maintenance task (M2), the bug description was: 'When a picture is displayed, the variable that counts the views is not updated. Feature, in which the bug occurs: CountViews'. The problem was that the method `increaseNumberOfViews` in the class `ImageData` had no content. In Figure 5.7, the according source code is shown to clarify the bug (Line 5).

```
1  public class ImageData{
2     /* ... */
3     //#ifdef includeCountViews
4     public void increaseNumberOfViews() {
5
6     }
7     /* ... */
8     //#endif
9  }
```

Figure 5.7: Bug for M2: `increaseNumberOfViews` is not implemented.

The bug description for the third maintenance task (M3) was: 'Although several pictures are set as favorites, the command to view favorites is not displayed in the menu. However,

the developer claims having implemented the according actions. Feature, in which the bug occurs: Favourites'. The bug was located in the class `PhotoListScreen` in the method `menuItem`. The according command is initialized in the class, but not added to the menu. In Figure 5.8, the according source code is shown (Line 10).

```
1  public class PhotoListScreen{
2     /* ... */
3     //#ifdef includeFavourites
4     public static final Command favoriteCommand = new Command("Set Favorite",Command.ITEM,1);
5     public static final Command viewFavoritesCommand = new Command("View
          Favorites",Command.ITEM,1);
6     // #endif
7     /* ... */
8     public void initMenu() {
9       /* ... */
10      // #ifdef includeFavourites
11      this.addCommand(favoriteCommand);
12
13      //#endif
14      /* ... */
15    }
16 }
```

Figure 5.8: Bug for M3: `viewFavoritesCommand` is not added.

In the fourth maintenance task (M4), the bug description was: 'If during sending a picture the according picture is not found, a NullPointerException is thrown. Feature, in which the bug occurs: SmsFeature'. The problem was that if a picture during sending was not found, an exception was thrown, but after the exception was caught, the program continued. However, the variable holding the picture was not initialized, but a method was called on it, so a NullPointerException occurred. The according method is `handleCommand` in the class `SmsSenderController`. In Figure 5.9, the according source code is shown (Line 11).

```
1  //#if includeSmsFeature
2  public class SmsSenderController{
3     /* ... */
4     public boolean handleCommand(Command c) {
5       /* ... */
6       ImageData ii = null;
7       byte[] imageBytes = null;
8       try {
9         ii = getAlbumData().getImageAccessor().getImageInfo(selectedImageName);
10        imageBytes =
              getAlbumData().getImageAccessor().loadImageBytesFromRMS(ii.getParentAlbumName(),
              ii.getImageLabel(), ii.getForeignRecordId());
11      } catch (ImageNotFoundException e) {/* ... */}
12      /* ... */
13      System.out.println("SmsController::handleCommand - Sending bytes for image
            " + ii.getImageLabel() + " with length: " + imageBytes.length);
14      /* ... */
15    }
16 }
17 //#endif
```

Figure 5.9: Bug for M4: potential null pointer access (Line 11)

Having decided how to control the confounding parameters and developed our tasks, we can conduct our experiment, which we explain in the next section.

5.4 Execution

In this section, we describe how we conducted our experiment. We start with how we collected the data in Section 5.4.1. Then, we describe how we recruited our subjects in Section 5.4.2. Finally, we describe all deviations from our plan that occurred during the experiment in Section 5.4.3.

5.4.1 Data collection

For collecting the data, presenting the tasks and programming experience questionnaire, we used *Globalpark Enterprise Feedback Suite Survey (EFS Survey)*[6]. It is a Web-based software system for organizing, conducting, and analyzing online surveys. We decided to use EFS Survey, because it allowed us to define the format of the data and export the collected data as SPSS data set, which we used to compute the descriptive statistics and inference tests. In addition, it captured the time that each page was displayed.

Unfortunately, EFS Survey did not support displaying HTML files with font colors (which is necessary to display source code files). Hence, we used the METOP server for displaying the HTML files. In order to assess the time subjects spend looking at an HTML source code file, we implemented a PHP script, which saves the absolute timestamp each time a HTML page was requested in a log file. Although we cannot determine whether subjects indeed looked at a HTML page the whole time and how long the last page in our experiment was displayed, we can use those files in order to have an idea what subjects did during our experiments. Since examining the log files is not feasible for our thesis, we postpone this for future work.

5.4.2 Conducting

When we started to plan our experiment, we contacted Sven Apel and Jörg Liebig, held the course *contemporary programming paradigms* at the University of Passau. Both agreed to make participation in our experiment mandatory for the students in order to complete the course. This way, we were sure to have enough subjects for our experiment. The curriculum of the course contained FOSD approaches and their application for developing SPLs. In order to motivate subjects for our experiment, we raffled a 30 Euro Amazon gift card, sponsored by METOP GmbH. Six weeks before the experiment, all students that were enrolled in this course were given a link to the questionnaire that measured programming experience as well as familiar domains and were given one week to complete it. The questionnaire itself took about fifteen minutes to complete (cf. Tables A.1 and A.2 in the appendix).

[6] http://www.globalpark.de/

The experiment was conducted at the end of June 2009 between 10 am and 12 am, instead of a regular exercise session of that course. We started with an introduction to CPP and CIDE, which lasted about fifteen minutes. After our introduction, subjects could ask questions. When all questions were answered, each subject was seated at a computer and started to work on the tasks on his own. Each task had the same structure: firstly, the task was introduced and it was explained what we expect from the subject; secondly, when subjects were clear on the instructions, they displayed the next page with the concrete task. An example of such a task description is shown in Figures A.2 and A.3 on pages 138 and 139 in the appendix.

We handed out the template of the collaboration diagram in the second task, so that subjects could not see the number of features for the warming up task (in which they should count the number of features). Subjects were instructed to request the templates when they were ready to begin with the second task. Since subjects kept the collaboration diagram template, but the roles should only be marked in the second task, we instructed subjects to use a different colored pencil from the third task on. This way, we saw if subjects made corrections to the collaboration diagram after the second task. Subjects had sheets of paper to make notes during working on the tasks and were encouraged to enter comments and critique at the end of the experiment.

We estimated the duration of the experiment to about two hours, whereas ninety minutes were planned for the completion of the tasks, fifteen minutes for the introduction and fifteen minutes for unforeseen events (e.g., later arriving subjects). Subjects were allowed to leave as soon as they had completed all tasks, and none of the subjects stayed longer in the room than the estimated two hours. We had three experimenters, who regularly checked that subjects worked as planned (e.g., did not use the search function or used the right pencil color for each task).

Despite all careful planning, deviations occur, which we discuss in the next section.

5.4.3 Deviations

A few subjects arrived later and got a personal introduction, in which the important points were mentioned again. We could not wait for all subjects to arrive, because we could use the room, in which we conducted our experiment, only for two hours. Furthermore, some subjects were seated in another room, because there were not enough working stations. All experimenters regularly checked how those subjects were doing, however we cannot be sure that they were exposed to the same noise level. For one subject, we had no CIDE template of the collaboration diagram, so he had to use a CPP template. In order not to jeopardize the anonymity of subjects, we decided not to note which subjects were in the other room and which got the wrong collaboration diagram template. We assume that our sample is large enough to compensate for these deviations.

In addition, for assessing the opinion of subjects (motivation, difficulty, and version), we forgot to include the seventh task, because we had only six tasks in our preliminary test. As soon as we noticed that, we asked subjects to evaluate the seventh task on the sheet of paper. Unfortunately, some of the subjects had already left the room at that time, so we did not have

the opinion of all subjects for that task.

We refer to these deviations when we interpret our data in Section 5.6. We mentioned them here because they occurred during this experimental stage. Having conducted the experiment, we can analyze our data, which we explain in the next section.

5.5 Analysis

In this section, we describe our data analysis. We do not interpret our data here, in order to present our results as objectively as possible (cf. Section 2.3). For the analysis, we prepared our data set by excluding subjects who did not complete both, the programming experience questionnaire (four subjects) and the experiment (seven subjects). Furthermore, we excluded one subject that did not answer the programming experience questionnaire genuinely, which we recognized by the comments he left on the questionnaire. That left us with 43 data sets to analyze. We refer to subjects that worked on the CPP version as *CPP subjects*, and subjects that used the CIDE version as *CIDE subjects*.

For analyzing the data, we used SPSS[7], which provides a large set of descriptives, significance tests, and graphs to analyze data sets. First, we describe our sample and data in Section 5.5.1. Then, we test our hypotheses in Section 5.5.2.

5.5.1 Descriptive statistics

In Tables 5.1 and 5.2, we describe our complete sample and both groups. We can see that gender (CPP: 19 males, 2 females, CIDE: 20 males, 2 females) and course of study (CPP: 12 Computer Science, 9 Internet Computing, CIDE: 10 Computer Science, 9 Internet Computing) are about equally distributed in both groups. We can see that the mean of programming experience (metric scale type) is similar in both groups (CPP: 39.29, CIDE: 38.64), which also counts for age (CPP: 24.62, CIDE: 24.23) and the number of years that subjects are studying (years of study, CPP: 3.19, CIDE: 3.71). In order to visualize the data in Table 5.2, we present histograms with plotted normal distributions for the complete sample in Figure 5.10. Since the variables in Table 5.1 have only two levels, we omit a histogram. The data in Table 5.1 have a nominal scale type, while the data in Table 5.2 an interval scale type (which is why we present the information in separate tables).

Next, we describe the answers our subjects gave. In Table 5.3, we show the response times for each task in seconds. EFS Survey saved relative time stamps for each page when a page was submitted, and the time stamp for the beginning of the questionnaire. Since for every task, we had one page that prepared the subjects and a following page with the actual task, the difference of the timestamp of both pages denotes the time subjects needed to complete a task. We can see that the differences in response time are the largest for the first (CPP: 728 seconds, i.e., 12 minutes, CIDE: 424 seconds, i.e., 7 minutes) and last task (CPP: 883 seconds, i.e., 15 minutes, CIDE: 1,404 seconds, i.e., 23 minutes). Furthermore, we can see that the last task

[7] http://www.spss.com/

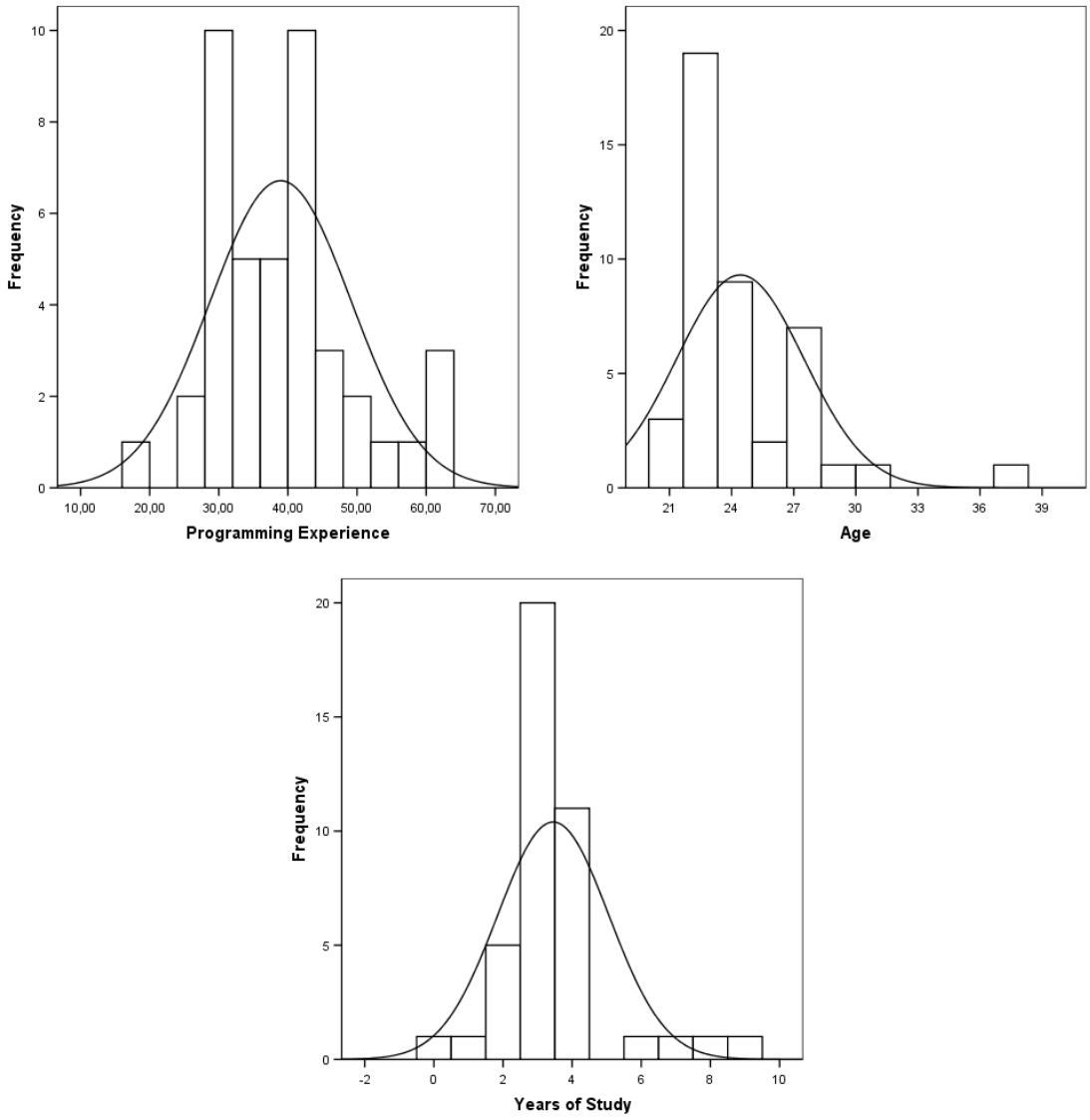


Figure 5.10: Histrograms for programming experience, age, and years since subjects study.

Variable	Level	CPP	CIDE	Total
Gender	Male	19	20	39
	Female	2	2	4
Subject	Computer Science	12	12	24
	Internet Computing	9	10	19

Table 5.1: Descriptives of sample by group for nominal scaled variables.

Variable	Group	N	Min	Max	Mean	Std
Programming Experience	CPP	21	26	63	39.29	10.39
	CIDE	22	19	61	38.64	10.28
	Total	43	19	63	38.95	10.22
Age	CPP	21	21	38	24.62	3.65
	CIDE	22	21	30	24.23	2.47
	Total	43	21	38	25.42	3.07
Years of study	CPP	21	0	6	3.19	1.12
	CIDE	21	1	9	3.71	1.98
	Total	42	0	9	3.45	1.61

N: number of subjects, Min/Max: smallest/largest observed value, Mean: arithmetic mean, Std: standard deviation

Table 5.2: Descriptives of sample by group for metric scaled variables.

took the longest time to complete. In Figure 5.11, we visualize the results for reaction times in box plots.

In Table 5.4, we show the opinion of subjects, which we assessed with a five point Likert scale (LIKERT [Lik32]). For motivation (Table 5.4), we can see that it increased until M1 (CPP: 3.95, CIDE: 4.00), and then decreased again (M4: CPP: 3.63, CIDE: 3.38). For difficulty, we see that for the maintenance tasks, the perceived difficulty increased (M1: CPP: 2.15, CIDE: 2.05, M4: CPP: 3.47, CIDE: 3.47). For the version, we can see that the opinion depending on the group differs more than for motivation and difficulty (M2: CPP: 3.10, CIDE: 2.23).

Having described our sample and data, we can continue our analysis with testing our hypotheses.

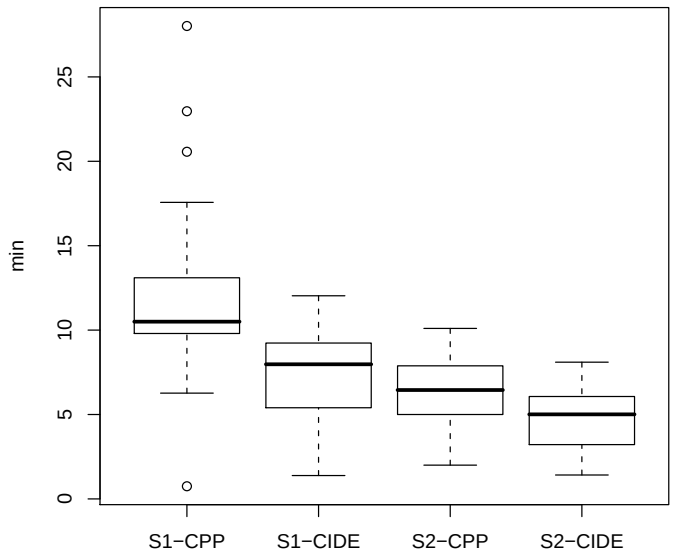
5.5.2 Hypotheses testing

In order to test our hypotheses, we have to transform them into statistical hypotheses. We traverse our hypotheses, explain the statistical hypotheses we tested with an according inference test, and state whether we confirmed or rejected it. In this section, we only report our results, whereas we interpret them in the next section (cf. Section 2.3.4).

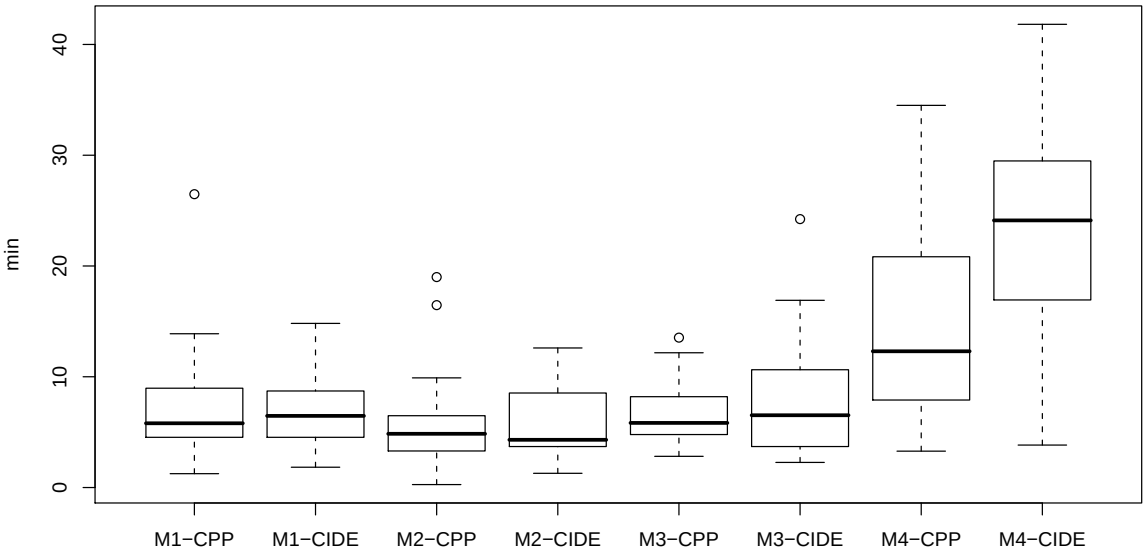
Response time for static tasks. Our first hypothesis is:

> For static tasks, CIDE annotation speeds up bottom-up program comprehension compared to CPP annotation.

Since we have two static tasks, we state for each task a null hypothesis:



(a) S1 – S2



(b) M1 – M4

Figure 5.11: Box plots for response times.

Task	Group	N	Min (s)	Max (s)	Mean (s)	Std	Diff (m)	Diff (%)
S1	CPP	21	45	1,681	738.90	351.76	5.3	43
	CIDE	22	83	722	424.50	210.27		
S2	CPP	21	120	606	369.33	135.70	1.5	24
	CIDE	22	85	486	281.23	112.27		
M1	CPP	21	75	1,589	434.24	326.25	0.4	5
	CIDE	22	110	889	412.36	204.18		
M2	CPP	21	16	1,140	351.10	280.95	0.2	3
	CIDE	22	77	756	341.36	191.87		
M3	CPP	21	169	812	395.86	171.84	1.2	16
	CIDE	22	136	1,454	470.64	320.94		
M4	CPP	21	197	2,070	882.76	529.13	8.7	37
	CIDE	22	230	2,509	1,404.14	575.91		

Task: task designator, Group: experimental group, N: number of subjects, Min/Max (s): largest/smallest observed value in seconds, Mean (s): arithmetic mean in seconds, Std: standard deviation, Diff (m/%): difference between groups in minutes/percent

Table 5.3: Descriptives of response times.

H_{01}: For S1, there is no difference in response times between CPP subjects and CIDE subjects.

H_{02}: For S2, there is no difference in response times between CPP subjects and CIDE subjects.

Since our sample is smaller than fifty subjects and the response times were not normally distributed, as revealed the Shapiro-Wilk test (SHAPIRO AND WILK [SW65]), we conducted Mann-Whitney-U tests to evaluate our hypotheses. In Table 5.5, we show the results of the tests. Since CIDE subjects are faster than CPP subjects (cf. Table 5.3) and the difference is significant (S1: $p < .001$, S2: $p < .027$) for both tasks, we reject our null hypotheses. Hence, we confirmed our hypothesis stated in the objective definition that the kind of annotation benefits the response time of CIDE subjects.

Response time for maintenance tasks. Or next hypothesis describes the effect of different annotations on maintenance tasks:

> For maintenance tasks, there are no differences in response time between CIDE annotation and CPP annotation with bottom-up program comprehension.

Our according statistical null hypotheses are:

		Motivation		Difficulty		Version	
Task	Group	N	Mean	N	Mean	N	Mean
S1	CPP	20	2.79 [1, 5]	20	2.35 [1, 4]	20	4.05 [3, 5]
	CIDE	22	3.45 [1, 5]	22	1.86 [1, 4]	22	2.23 [1, 3]
S2	CPP	20	3.05 [1, 5]	20	2.25 [1, 4]	20	4.05 [2, 5]
	CIDE	22	3.55 [2, 5]	22	2.27 [1, 4]	22	2.59 [1, 5]
M1	CPP	20	3.95 [1, 5]	20	2.15 [1, 5]	20	3.05 [2, 4]
	CIDE	22	4.00 [3, 5]	22	2.05 [1, 3]	22	2.18 [1, 3]
M2	CPP	20	4.15 [1, 5]	20	2.30 [1, 4]	20	3.10 [2, 5]
	CIDE	22	3.64 [2, 5]	22	2.14 [1, 4]	22	2.23 [1, 3]
M3	CPP	20	3.85 [1, 5]	20	2.75 [1, 4]	20	3.05 [2, 5]
	CIDE	21	3.62 [1, 5]	21	2.91 [1, 5]	22	2.23 [1, 3]
M4	CPP	16	3.63 [1, 5]	16	3.47 [2, 5]	13	3.08 [2, 4]
	CIDE	16	3.38 [1, 5]	16	3.69 [2, 5]	16	2.44 [1, 3]

Group: group of the subjects, N: number of subjects, Mean: mean [min, max] of subjects' opinion; Coding: Motivation: 1: very unmotivated, 2: unmotivated, 3: neither, 4: motivated 5: very motivated; Difficulty: 1: very easy, 2: easy, 3: neither, 4: difficult, 5: very difficult; Version: 1: clearly worse, 2: worse, 3: no difference, 4: better, 5: clearly better

Table 5.4: Descriptives of subjects' opinion.

H_{03}: For M1, there is no difference in response times between CPP subjects and CIDE subjects.

H_{04}: For M2, there is no difference in response times between CPP subjects and CIDE subjects.

H_{05}: For M3, there is no difference in response times between CPP subjects and CIDE subjects.

H_{06}: For M4, there is no difference in response times between CPP subjects and CIDE subjects.

In Table 5.6, we show the results of the Mann-Whitney-U test, which we applied because of our sample size and non normal distribution of our data (cf. Section 2.3.3.3).

The results mean that for tasks M1-M3, the observed differences in response time are random and we confirm $H_{03} - H_{05}$. For M4, the p value is smaller than .05, which means that the observed difference in response time is significant. Hence, we reject this null hypothesis (H_{06}). Consequently, we have to reject our hypothesis stated during objective definition (i.e., that there are no differences in response time for maintenance tasks).

Task	Group	N	Mean Rank	Sum of Ranks	U	p
S1	CPP	21	28.76	604.0	89.0	.001
	CIDE	22	15.55	342.0		
S2	CPP	21	26.33	553.0	140.0	.027
	CIDE	22	17.86	393.0		

Mean Rank/Sum of Ranks: statistical values for the significance test, U: test statistic, p: significance level

Table 5.5: Mann-Whitney-U test for response times of S1 and S2.

Task	Group	N	Mean Rank	Sum of Ranks	U	p
M1	CPP	21	21.52	452.0	221.0	.808
	CIDE	22	22.45	494.0		
M2	CPP	21	21.55	452.5	221.5	.817
	CIDE	22	22.43	493.5		
M3	CPP	21	21.26	446.5	215.5	.706
	CIDE	22	22.70	499.5		
M4	CPP	21	16.67	350.0	119.0	.007
	CIDE	22	27.09	596.0		

Mean Rank/Sum of Ranks: values for the significance test, U: test statistic, p: significance level

Table 5.6: Mann-Whitney-U test for response times of M1 - M4.

Number of completed tasks. Our next hypothesis concerns the number of completed tasks:

> There are no differences in the number of correctly solved tasks between CPP annotated and CIDE annotated source code with bottom-up program comprehension.

The according statistical null hypotheses we evaluate are:

> H_{07}: For S1, there is no difference in the number of false positives between CPP subjects and CIDE subjects.
>
> H_{08}: For S1, there is no difference in the number of false negatives between CPP subjects and CIDE subjects.
>
> H_{09}: For S1, there is no difference in the number of right positives between CPP subjects and CIDE subjects.
>
> H_{010}: For S2, there is no difference in number of successfully correctly solved tasks between CPP subjects and CIDE subjects.
>
> H_{011}: For M1, there is no difference in number of successfully correctly solved tasks between CPP subjects and CIDE subjects.
>
> H_{012}: For M2, there is no difference in number of successfully correctly solved tasks between CPP subjects and CIDE subjects.
>
> H_{013}: For M3, there is no difference in number of successfully correctly solved tasks between CPP subjects and CIDE subjects.
>
> H_{014}: For M4, there is no difference in number of successfully correctly solved tasks between CPP subjects and CIDE subjects.

Since we want to compare frequencies with these hypotheses, we conduct a χ^2 test (cf. Section 2.3.3.3). The result is shown in Table 5.7. We can see that for all tasks, there is no significant difference in the number of completed tasks. Hence, we confirm our null hypotheses and thus our hypothesis stated during objective definition.

Opinions of subjects. Our last hypothesis stated that subjects of different groups estimate their performance with the other source code version differently:

> Subjects that worked with the CPP version estimate their performance with the CIDE version better than subjects that worked with the CIDE version and estimate their performance with the CPP version.

The according null hypotheses are:

Task	χ^2	df	p
FP	7.66	7	.36
FN	8.18	9	.52
RP	7.60	9	.58
S2	1.61	4	.81
M1	.41	1	.52
M2	.41	1	.52
M3	.00	1	.95
M4	.56	1	.45

χ^2: test statistic, df: degrees of freedom, p: significance level

Table 5.7: χ^2 test for number of correct tasks.

H_{015}: For S1, there is no difference in estimation of performance between CPP subjects and CIDE subjects.

H_{016}: For S2, there is no difference in estimation of performance between CPP subjects and CIDE subjects.

H_{017}: For M1, there is no difference in estimation of performance between CPP subjects and CIDE subjects.

H_{018}: For M2, there is no difference in estimation of performance between CPP subjects and CIDE subjects.

H_{019}: For M3, there is no difference in estimation of performance between CPP subjects and CIDE subjects.

H_{020}: For M4, there is no difference in estimation of performance between CPP subjects and CIDE subjects.

Since those data have ordinal scale type, we conduct a Mann-Whitney-U test for evaluating these hypotheses. The results, presented in Table 5.8, show the difference of this estimation is very significant for all tasks. Hence, CPP subjects thought they would have performed better with the CIDE version, and CIDE subjects thought they would have performed worse with the CPP version. Thus, we reject our null hypotheses and confirm our hypotheses stated during objective definition (i.e., there is a difference in the estimation of the performance with the other source code version).

Analyzing influence of motivation and difficulty. Next, we compare the opinion of subjects according to motivation and difficulty. Although we did not state any hypotheses during objective definition, we have to check whether they differ significantly, because we assessed

Task	Group	N	Mean Rank	Sum of Ranks	U	p
S1	CPP	19	31.50	630.0	20.0	.00
	CIDE	22	12.41	273.0		
S2	CPP	20	29.00	580.0	70	.00
	CIDE	22	14.68	323.0		
M1	CPP	20	28.88	577.5	72.5	.00
	CIDE	22	14.80	325.5		
M2	CPP	20	28.43	568.5	81.5	.00
	CIDE	22	15.20	334.5		
M3	CPP	20	27.78	555.5	94.5	.00
	CIDE	21	15.80	347.5		
M4	CPP	16	18.65	242.5	56.5	.01
	CIDE	16	12.03	192.5		

Mean Rank/Sum of Ranks: values for the significance test, U: test statistic, p: significance level

Table 5.8: Mann-Whitney-U test for version.

these opinions in order to analyze the influence of motivation. In order to be sure that motivation and perceived difficulty influenced both groups in the same way, we should obtain no significant difference. Since our according null hypotheses are similar to the null hypotheses for the estimation of version and to save space, we do not state the according null hypotheses explicitly.

Since those data are ordinal, we conduct a Mann-Whitney-U test. The results are shown in Tables 5.9 and 5.10. Except for M2, both groups were motivated to the same amount for the tasks. Subjects estimated the difficulty of both tasks equally for both groups. Hence, we can be sure, that we controlled the influence of motivation (except for M2) and difficulty.

Having evaluated our statistical hypotheses, we have to discuss what this means for our hypotheses we derived during the objective definition. Furthermore, next steps can be derived based on our results.

5.6 Interpretation

Based on the data and our rejected and confirmed statistical hypotheses, we need to draw conclusions from our results for future research on understandability of FOSD approaches. In order to do so, we first evaluate what our results mean for our hypotheses in Section 5.6.1. Then, we discuss threats to validity of our experiment in Section 5.6.2, which could have biased our results. Eventually, in Section 5.6.3 we give some ideas for future work that can be

Task	Group	N	Mean Rank	Sum of Ranks	U	p
S1	CPP	19	17.42	331.0	141	.06
	CIDE	22	24.09	530.0		
S2	CPP	20	19.40	388.0	178	.27
	CIDE	22	23.41	515.0		
M1	CPP	20	21.75	435.0	215	.89
	CIDE	22	21.27	468.0		
M2	CPP	20	25.23	504.5	145.5	.04
	CIDE	22	18.11	398.5		
M3	CPP	20	22.10	442.0	188	.54
	CIDE	21	19.95	419.0		
M4	CPP	16	17.63	282.0	110	.67
	CIDE	16	15.38	246.0		

Mean Rank/Sum of Ranks: values for the significance test, U: test statistic, p: significance level

Table 5.9: Mann-Whitney-U test for motivation.

Task	Group	N	Mean Rank	Sum of Ranks	U	p
S1	CPP	20	24.23	484.5	165.5	.15
	CIDE	22	19.02	418.5		
S2	CPP	20	21.33	426.5	216.5	.92
	CIDE	22	21.66	476.5		
M1	CPP	20	21.70	434.0	216	.91
	CIDE	22	21.32	469.0		
M2	CPP	20	22.43	448.5	201.5	.63
	CIDE	22	20.66	454.5		
M3	CPP	20	21.03	420.5	210.5	.80
	CIDE	22	21.93	482.5		
M4	CPP	17	16.09	273.5	120.5	.56
	CIDE	16	17.97	287.5		

Mean Rank/Sum of Ranks: values for the significance test, U: test statistic, p: significance level

Table 5.10: Mann-Whitney-U test for difficulty.

derived from our results.

5.6.1 Evaluation of results and implications

Having tested our statistical hypotheses, we need to interpret what this means for our hypotheses stated during objective definition. We traverse the hypotheses as we stated them in objective definition and discuss the implications of our according statistical hypotheses for each of them.

Response time for static tasks. Our hypothesis is:

> For static tasks, CIDE annotation speeds up bottom-up program comprehension compared to CPP annotation.

Since the observed difference is significant according to the Mann-Whitney-U test, we have confirmed this hypothesis. We can explain this difference by the preattentive color perception, compared to attentive text perception (5.2.3). The consequence we can draw from this result is that colors can help a programmer to understand a program, when he does static tasks like locating feature code. Hence, we could support program comprehension by using background colors to annotate source code. However, before we can generalize this result, we should confirm it and increase external validity, for example by using programming experts as subjects, testing top-down program comprehension, other programming languages, or programs with more LOC or feature to check scalability of our results.

Response time for maintenance tasks. Our hypothesis is:

> For maintenance tasks, there are no differences in response time between CIDE annotation and CPP annotation with bottom-up program comprehension.

Since this did not hold for the last maintenance task (M4), we cannot confirm this hypothesis. Now, we have to look for an explanation for the difference in the last task. In this task, CIDE subjects were significantly slower than CPP subjects. Hence, the preattentive color perception cannot be responsible for the difference. Furthermore, subjects already identified the roles of all classes and marked them in the collaboration diagram template in S1, so locating according feature code should not have been the main part of this task. Hence, we take a closer look at the location of the bug of M4: the class `SmsSenderController`. Since it completely belongs to the SmsFeature, it is entirely annotated with red in the CIDE version. This could be hard for the eyes to look at when searching for the bug. In order to explore this explanation, we look through the comments subjects were encouraged to enter at the end of our experiment. We found indeed some comments in which subjects criticized the annotation with red, which is one evidence for our explanation. In future experiments, we can try to confirm this explanation.

The consequence we can draw from the difference in response time for both static and maintenance tasks is that we could enable programmers to adjust the intensity of background color to their needs. For example, when searching for feature code, intense colors pose no problem, but can reduce the time it takes to locate feature code. On the other hand, if a class is analyzed for a bug, colors are not helpful or even can disturb a programmer. For this case, the background color should be faded out. We can test whether our assumption holds in further experiments.

Number of completed tasks. Our hypothesis concerning the number of completed tasks is:

> There are no differences in the number of correctly solved tasks between CPP annotated and CIDE annotated source code with bottom-up program comprehension.

We confirmed this hypothesis with a χ^2 test. Since both kind of annotations provide information about feature code and the feature to which it belongs, subjects are enabled to correctly solve our tasks, independently of the kind of annotation. Hence, we showed that the kind of annotation has no effect on a solution of a task. Only the response time of subjects is influenced. Thus, future experiments could estimate the magnitude of response time benefit for subjects. This way, we would be able to estimate whether CIDE is worth the effort to use compared to CPP, or whether the performance benefit is so small that the costs for learning CIDE are higher than our benefit.

Opinions of subjects. Our last hypothesis stated that subjects of different groups estimate their performance with the other source code version differently:

> Subjects that worked with the CPP version estimate their performance with the CIDE version better than subjects that worked with the CIDE version and estimate their performance with the CPP version.

For the estimation of performance with the other version of the source code, we found a very significant difference for all tasks. When looking through the comments of subjects, we find that some CIDE subjects were happy to get to work with the CIDE version, whereas some CPP subjects wished they had worked with the CIDE version. This could explain the difference in estimating the performance, because some subjects liked the CIDE version better, which they reflected to their performance and thus estimated that they would have performed better with the CIDE version or worse with the CPP version. One might argue that this also could affect motivation, yet we did not find any differences in motivation between both groups (except for M2, in which CPP subjects were more motivated). Hence, despite no differences in correct solutions, subjects thought that they perform differently with a different annotation. Consequently, we could use colors more often in programming, because it could make subjects happier.

Subjects opinion. We assessed the opinion of subjects in order to control the influence of motivation and difficulty of tasks. For difficulty, we observed no significant differences between CPP and CIDE subjects. The same counts for motivation, except for M2. Hence, we can assume that we sufficiently controlled the effect of difficulty and motivation.

However, for M2, we have to take a closer look at subjects' motivation. For finding an explanation, we looked at subjects' comments, but did not find any clues that could be responsible for the difference in especially this task. If the position or ordering effect influenced subjects' performance, both groups should be affected the same way. Maybe the experimenters influenced subjects especially in this task, because more subjects communicated for this task with one of the experimenters than for other tasks. Or, CIDE subjects noticed that the advantage in response time for S1 and S2 did not hold anymore, which influenced their motivation. However, those are only speculations, and we found no information that could explain why this difference occurred. Depending on the importance of this difference, we can either explore it further in future experiments or accept that it occurred.

Having interpreted our results, we discuss threats to validity.

5.6.2 Threats to validity

Threats to validity are important to mention, because they can render our results useless. Furthermore, we can learn from these threats and avoid them in subsequent experiments. First, we discuss threats to internal validity, then threats to external validity.

Threats to internal validity

A first problem is that, despite all our effort, we could have not been careful enough for controlling confounding parameters. For example, some subjects could have forgot to mention that they are experienced with the domain of mobile devices, one group could be more intelligent than the other or some could have visited a technical high school, whereas others a Waldorf high school. We could have unintentionally influenced our subjects or not motivated them well enough. Problems like this could always occur despite all effort. However, we are sure to have managed those problems due to our careful preparation.

Further threats occur due to the deviations we described. Firstly, some subjects sat in another room and had an individual introduction (because they were late). This could introduce a different noise level, lighting conditions, and training for subjects. Therefore, we tried to keep those parameters constant, so that we can assume that the influence of those parameters is negligible. Secondly, for one subject, we had no CIDE template of the collaboration diagram, so he had to use a CPP template. This could have affected his speed, because he had to map from colors to textual annotation for completing the task. However, to assure anonymity, we did not evaluate whether it had an effect. Thirdly, since for assessing the opinion of subjects we forgot the seventh task, several opinions were missing. We do not know how this could have affected the results.

Furthermore, the source code we used could have influenced the performance of subjects:

Although it was reviewed according to conventions, some unusual formatting styles and indentations occurred, in addition to some confusing comments. However, we decided to leave the source code as it was for two reasons: Firstly, it helps replicating our results, because the versions we created can be unambiguously reproduced (by deleting scenario-related comments and `#ifdef` statements for the CIDE version). Secondly, besides those problems, the source code is not too small or too big (3,800 LOC) and implemented an unfamiliar domain. Creating such a program of our own or further searching for such a program would not have been feasible for us.

Finally, one threat emerges from our programming experience questionnaire. In order to check whether our questionnaire measures programming experience, we analyzed the relationship of the programming experience with the response time and number of correctly solved task, because more experienced programmers should be faster and more correct in fixing a bug. Thus, a relationship between our programming experience value and response time/number of correctly solved tasks would indicate that we indeed measured programming experience with our questionnaire.

In Figure 5.12, we show the relationship between programming experience and response time for M4 as example to illustrate our analysis. The vertical axis denotes the response time in seconds, the horizontal axis the programming experience value. The line in this figure denotes the relationship between programming experience and response time. The less the data points deviate from this line, the stronger is the relationship.

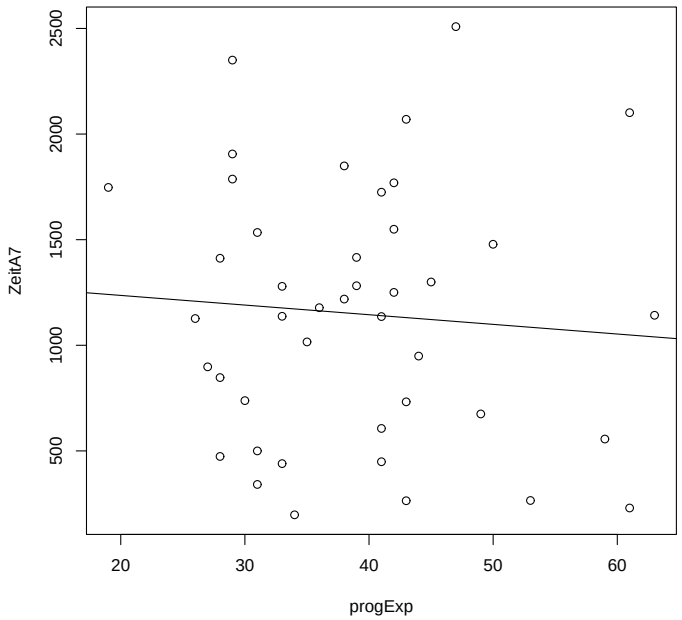


Figure 5.12: Relationship between programming experience and response time for M4.

Since all data points are widely scattered, the relationship between response time and pro-

gramming experience is rather low for M4. This counts for all other tasks and relationship with correctly solved tasks as well. Hence, we could not validate that we measured programming experience with our questionnaire. This does not necessarily mean that our programming experience questionnaire is useless, but several other reasons could be responsible for this result. For example, the differences in programming experience of our subjects could be too small or programming experience may have nothing to do with response time or number of correctly solved tasks in our experiment. In the worst case, this result could mean that we did not control the effect of programming experience at all and that our results are biased by programming experience and useless. We plan to explore this result in future experiments.

Threats to external validity

In our experiment, we have maximized our internal validity in order to feasibly and soundly measure the effect of different annotations on program comprehension. Thus, we have neglected external validity. For example, we used students as subjects (no expert programmers), tested only bottom-up program comprehension (not top-down or both), had only one programming language for CPP and CIDE, and excluded tool support. Hence, our results are only applicable to the subjects and setting we used in our experiment. For generalizing our results, we need to include those parameters we kept constant, for example expert programmers, top-down program comprehension, further programming languages, different source code, more features and feature interactions, or different granularity of annotations. This is an important step for future work based on our experiment, which we discuss in the next section.

5.6.3 Next steps

Obvious next steps are to replicate our experiment to confirm our results and increase external validity in following experiments by using other programming languages (e.g., C++), testing top-down program comprehension, expert programmers, etc.

As example, we discuss how we can carefully include tool support in our evaluation. In CIDE, tool tips show the feature names for annotated source code. We did not used tool tips in our experiment, because otherwise, CIDE subjects had the information of color in addition to the textual information. This would have provided them with more information than our CPP subjects and thus would have biased our results. Based on the observation that for most maintenance task there is no difference in response time between CPP and CIDE version, we could test whether there is a difference between textual information (CPP) and color plus textual information (CIDE).

One interesting result is the differences in response times. For static tasks, in which it suffices to scroll through the code, colors provided a performance benefit. On the other hand, in maintenance tasks, in which the code needed to be analyzed text based, our results indicate that colors have – at best – no effect (or a very small effect). An idea for future work could be to enable programmers to adjust the intensity of background color to their needs. For example, when searching for feature code, intense colors pose no problem, but can reduce the time it

takes to locate feature code. On the other hand, if a class is analyzed for a bug, colors are not helpful or even can disturb a programmer. For this case, the programmer could be enabled to adjust the intensity of the background color or to switch between text and color annotations. Before extending CIDE by this feature, we could test in an experiment if adjusting the intensity of colors really benefits program comprehension.

Several subjects mentioned that they found it hard to identify the `#ifdef` statements, because they were highlighted like other ordinary comments and not at the beginning of a line. We could investigate these issues by comparing the CPP version we used with one in which we use a different color for the `#ifdef` statements and/or putting them at the beginning of a line. A significant difference in response time would prove subjects' statements. In addition, we could compare the newly created CPP version with the CIDE version, where no significant differences in response times would prove subjects' comments.

Another way to continue our work is to analyze our programming experience questionnaire further. Since we found no relationship between the results of our questionniare and our experiment, it could mean that we did not measure programming experience with our questionnaire. For exploring why we found no relationship, we can examine the log files for the HTML files of the source code and check whether single items in our programming experience questionnaire have more influence than others or whether some have a reverse effect for programming experience.

Hence, there are several interesting steps for future work that can be derived from our results and which can be chosen depending on the focus of interest. In the next section, we summarize this chapter.

5.7 Summary

In this chapter, we described our experiment, in which we compared the effect of CPP and CIDE on program comprehension. We showed how we controlled confounding parameters, which we explained in Chapter 3. We revealed differences in response times in favor of CIDE, if we have static tasks. For maintenance tasks, in which subjects have to analyze the source code on a textual basis, we obtained either no differences in response times or differences in favor of CPP. A further difference was found in subjects' opinion regarding the source code version: CIDE subjects estimated the effect of colors on their performance more positive as CPP subjects the effect of `#ifdef` statements on their version.

With this experiment, we demonstrated the small scope of hypotheses that feasibly measure the effect of different FOSD approaches on program comprehension: We choose Java as programming language, because it is common to two FOSD approaches, CPP and CIDE. We kept a lot of confounding parameters constant, which maximized our internal validity, but reduced our external validity. However, due to the high degree of internal validity, we can draw sound conclusions from our experiments, on which we can base future research.

In the next section, we discuss relate our thesis to the work of other researchers.

Chapter 6

Related Work

In this section, we related work. Firstly, we discuss work on reading comprehension, which is similar but more general than program comprehension, because in both cases, text has to be processed. The functionality that such tools provide can be used and adapted for FOSD approaches. Secondly, we present objective measures to assess program comprehension, because they show another way to assess program comprehension. Thirdly, we discuss some reviews of the literature of empirical software engineering, because they provide us with an overview of the status of empirical software engineering in general. Finally, we discuss approaches that aim at supporting program comprehension beyond programming paradigms.

Reading comprehension

Program comprehension is related to reading comprehension, since for both processes, verbal information needs to be perceived and understood. Numerous studies are concerned with reading comprehension and its measurement, for example MCKEOWN ET AL. [MBOP83], MEZYNSKI [Mez83], or WATERS [Wat96]. There are several studies that evaluated the validity of reading comprehension measures, for example FUCHS ET AL. [FFM88] or ANDERSON ET AL. [ABPC91]. Based on those numerous studies, several attempts are made to efficiently teach children how to read (summarized in PRESSLEY [Pre98]). It would be interesting to see if some of those ideas can also be applied to program comprehension or teaching programmers how to implement an algorithm so that other programmers can understand it better.

For example, GOUGH [Gou65] and SLOBIN [Slo66] found that reading comprehension depends on how a sentence is stated. Specifically, they examined the effect of voice (active vs. passive), truth (true vs. false), and affirmation (affirmative vs. negative). Results are that active, true, and/or affirmative stated sentences were processed significantly faster than according passive, false, and/or negative sentences, respectively. For example, the sentence 'Leonard hit Howard' is processed faster than 'Howard was hit by Leonard', although they have the same content. For program comprehension, the result regarding affirmation is interesting, because they can be applied to programming, in the sense that affirmative statements are processed faster than negative statements. For example, instead of `while(!(i==42 &&`

`j==23))`, the statement `while(i<42 && j<23)` could be processed faster and should thus be preferred to increase program comprehension.

Another experiment is described in BOYSEN [Boy77] and assessed the effect of logical connectives of sentences, for example *and* and *or*. The results indicated that sentences connected with *and* are better understood than sentences connected with *or*. This can be applied to statements in source code, in the sense that statements connected with *and* should be preferred to statements connected with *or*, because programmers can understand them better.

Furthermore, several tests exist that measure reading comprehension, for example the GAP (MCLEOD [McL89]). Similar, a test measuring program comprehension could be an interesting way to reduce the effort of designing experiments measuring program comprehension, because we would not have to worry about finding the right technique and measure for program comprehension, but we could simply use an existing test. For example, we could create a program and let subjects solve predefined tasks. This way, we would not have to worry about how reliable our tasks measure program comprehension and whether they are comparable, but we know that they are because they were carefully designed and exhaustively evaluated during the development of the test.

Objective measures for program comprehension

Since measuring program comprehension is very tedious, other measures have been suggested that can simply be derived from the source code without consulting subjects. The first primitive measures, based on the physical size of a software project (e.g., LOC or number of statements), are not sufficient as complexity measures, because they neglect other relevant facets of source code, for example, the number of control paths (MCCABE [McC76]). Hence, McCabe developed a measure for program complexity based on the number of independent control paths, which he called cyclomatic complexity (MCCABE [McC76]). The number of independent path are based on the program control graph, which is a directed graph that represents blocks of source code with sequential control flow as nodes and branches as edges. The smaller the number of independent path in the program control graph is, the better the understandability of a program should be.

Another example is the measure developed by GORDON [Gor77], based on Halstead's software science measures HALSTEAD [Hal77]. It describes the mental effort to understand the program and depends on the total number of operators and operands as well as the unique operators and operands of a program.

The benefit of those measures is that they can be computed based on properties of the source code without consulting subjects. However, as BOYSEN [Boy80] showed, those measures are not sufficient as comprehension measure, because they focus only on one facet of a program. Hence, they alone cannot be used as program comprehension measures, but rather in conjunction with empirical assessment of program comprehension.

Review of literature

Numerous studies were conducted to provide an overview of empirical software engineering in general, for example TICHY ET AL. [TLPH95], ZELKOWITZ AND WALLACE [ZW97], GLASS ET AL. [GVR02], SJOBERG [SHH+05], or KAMPENES ET AL. [KDHS09]. The results are that the status is improving, however experiments are too often conducted with students as subjects and not generalized with experts. This poses a serious threat to external validity (cf. Section 2.3.2.1), which we discussed in our work and included in our agenda (cf. Chapter 4). Furthermore, researchers are often restricted to their discipline and do not relate their work to other areas, for example cognitive psychology or behavioral sciences (GLASS ET AL. [GVR02]). However, those disciplines faced and solved similar problems, for example how to measure internal cognitive processes or behavior, so that empirical software engineering could profit from this experience. For example, we discussed think-aloud protocols as technique to measure program comprehension, which were developed in cognitive psychology (WUNDT [Wun74]).

To the best of our knowledge, there are no literature reviews focusing on *program comprehension*. However, conducting a sound review or meta analysis regarding experiments that measure program comprehension is an interesting approach for future work (cf. Section 3.2). One way to support future meta analyses is to provide guidelines for reporting experiments in the area of software engineering. There are several suggestions and evaluations, for example by JEDLITSCHKA ET AL. [JP05] or JEDLITSCHKA AND PFAHL [JCP08]. However, according to KITCHENHAM ET AL. [KAKB+08], they are not accurate enough (e.g., ambiguous section names or contents) and thus need to be improved before being widely accepted.

Enhancing program comprehension beyond programming paradigms

The focus of our work is understandability of programming paradigms within the scope of FOSD. Besides that, several other approaches exist to support program comprehension. IDEs like Eclipse, Microsoft Visual Studio, or FeatureIDE, which was developed to meet specific requirements of FOSD (KÄSTNER ET AL. [KTS+09]), support the programmer in understanding and developing a program by syntax highlighting, call hierarchies, outline views, etc.

Several tools exist for the visualization of the contents of a project, for example SeeSoft (SEESOFT [ESEES92]), which represents files as rectangles and source code lines as colored pixel lines, whereas the color is an indicator of the age of the according source code line. SeeSoft is interest for program comprehension, because it abstracts from the statement level of source code and thus helps to get an overview of a project.

Other approaches focus on supporting program comprehension during the maintenance process. We discuss two of them as example. One interesting approach aims at finding the right developer for a maintenance task, based on commits of software systems' version control repositories (KAGDI ET AL. [KHM08]). Since a developer familiar with a software system is identified, the comprehension process can be speeded up and the cost reduced.

A further idea is to automatically update documentation on source code (HAMMAD ET

AL. [HCM09]). Based on the change history of current code, design documents are adjusted to the current version of the code. An according tool, HippoDraw, outperforms manual update. This saves a lot of time and costs, because fewer errors are produced during maintenance and software developers can focus on updating source code without having to manually adjust according design documents. This way, program comprehension is enhanced, because design documents, one source of information for understanding a program (KOENEMANN AND ROBERTSON [KR91]), are automatically updated. This could be adapted for FOSD approaches.

For all approaches, it would be interesting to test their effect on program comprehension in conjunction with different FOSD approaches. Results could help to reduce the semantic gap between human and computerized way of thinking.

This concludes our discussion of related work. In the next section, we summarize our work and present some starting points for future work.

Chapter 7

Conclusion

The difference between human way of thinking and computerized way of thinking, referred to as *semantic gap*, impairs program comprehension. This is problematic, because program comprehension is necessary for software maintenance, a critical factor for costs of software development. In order to keep maintenance costs low, it is imperative to minimize the semantic gap, for which paradigms like OOP and FOSD were developed. In this thesis, we have set out the effect of different FOSD approaches on program comprehension. Specifically, we defined three goals for our work:

- Evaluate the feasibility of comparing FOSD approaches regarding their effect on program comprehension.
- Create an agenda for evaluating FOSD approaches.
- Demonstrate our results with an experiment.

Firstly, measuring program comprehension has to be done empirically, because it is an internal problem solving process. Since this can be a very tedious endeavor, we need to evaluate the feasibility in order to know how to proceed. For evaluating feasibility, we identified confounding variables for program comprehension by literature review and expert consultation and explained how we can control their influence. We showed that due to the number of confounding parameters, it impossible to conduct a feasible *and* sound experiment in which all FOSD approaches are compared. By stepwise reducing the complexity of the comparison, we showed the small scope of sound and feasible experiments.

Based on our analysis, we developed an agenda for evaluating which FOSD approach provides the most benefit on program comprehension under which circumstances: design experiments with high degree of internal validity and stepwise increase complexity of comparison to increase external validity. This proceeding produces sound results, however it may take decades until we have an exhaustive body of knowledge about the effect of different FOSD approaches on program comprehension. Hence, it is necessary to build a research community, such that we have more resources to answer our question. We hope to have encouraged other researchers to join us.

In order to demonstrate our explanations, we presented an experiment that compared the effect of different annotations (text based á la CPP vs. color based á la CIDE) on a Java program. The results showed that annotating colors benefits identifying relevant code, but has at best no effect on fixing bugs, for which source code needs to be examined on a textual basis. Furthermore, it demonstrated the small scope of feasible experiments measuring the effect of different FOSD approaches on program comprehension. So, how can our work can be continued?

Future work

A first starting point to continue our work is based on the results of our experiment. We found that colors, compared to text based annotation, had a positive effect on response time, when subjects had to locate annotated source code. For maintaining tasks, in which source code had to be examined on a textual basis, we found no effect or a negative effect of colors on response time, respectively. This lead to the idea of adjustable intensity of background color to a programmer's needs. Thus, when a programmer tries to locate source code fragments or get an overview of a program, intense background colors can support him to succeed. On the other hand, if a programmer tries to understand an algorithm, background colors are irrelevant, because the programmer has already located relevant source code and now has to process it on a textual basis. For this case, he can fade the background color to a lower level. A further idea is to combine textual and color based annotations, such that we can benefit from the combination of textual and color based annotations. In future experiments, we plan to test the effect of adjustable intensity of the background color in an experiment and providing textual *and* color information about feature source code.

Second, in our review of the literature for identifying confounding parameters on program comprehension, we had a rather small selection of papers regarding experimental research on program comprehension. However, a larger number of papers was not feasible for our work. Furthermore, there is no consensus on where to describe confounding parameters, which means that we thoroughly had to examine every section. Firstly, it would be interesting to conduct a sound meta analysis with a representative selection of papers. This way, we can get a better overview about empirical research on program comprehension and refine our list of confounding parameters. Secondly, we can evaluate and extend the guidelines of how to report experiments provided for example by JEDLITSCHKA AND PFAHL [JCP08]. For this process, we could include the guidelines provided by the American Psychological Association, because they are constantly evaluated and improved since 1957 (AMERICAN PSYCHOLOGICAL ASSOCIATION [Ass57]) and currently available in the sixth edition AMERICAN PSYCHOLOGICAL ASSOCIATION [Ass09]. Hence, we can profit from over fifty years of experience.

Based on the results of a sound meta analysis, we can adjust our programming experience questionnaire we developed for our experiment (cf. Section 5.3.1) to a valid and reliable test instrument. This is important for two reasons: Firstly, we cannot be sure whether our questionnaire indeed measures programming experience, because we found no relationship to response time or number of correctly solved task for our experiment (cf. Section 5.6.2). Secondly, a

valid and reliable test instrument would allow us to simply administer a questionnaire which we know measures programming experience, instead of carefully designing it for every experiment we conduct. Steps for developing a sound questionnaire include thorough research of literature concerning programming experience, recruiting a large sample of subjects to test different versions of our questionnaire, and find measures with which we can compare the answers of our questionnaire (MOOSBRUGGER AND KELAVA [MK07]). The last step is necessary in order to be sure to measure programming experience and not something else. For example, the results of our questionnaire should not have much in common with a test measuring personality traits (referred to as *discriminant validity*), but rather with response times or correctness in a programming task (referred to as *convergent validity*, MOOSBRUGGER AND KELAVA [MK07]).

In Section 3.4.3, we pointed out the difficulty of controlling the influence of tool support properly, because different IDEs provide different functionalities that can be differently used. One way to deal with this problem is to implement a tool that can be adapted as necessary for an experiment. This way, we can simply switch on or off functionalities we want to include or exclude in an experiment. Furthermore, all subjects would be familiar with this tool to an equal level, because we only use it for our experiment. In training sessions, we can familiarize subjects with functionalities we intend to use in our experiment. Hence, we could reduce the effort of controlling the influence of tool support.

A further interesting for future work is based on results of comparing OOP with procedural programming languages. Some studies showed a positive effect on performance in favor of OOP, yet subjects were not satisfied with OOP compared to procedural programming languages (e.g., HENRY ET AL. [HHL90] or DALY ET AL. [DBM+95]). An interesting idea would be to check how subjects' neurological response depends on FOSD approach he is working with. For this case, *functional magnetic resonance imaging* is an interesting approach (BUXTON [Bux01]). It measures the extent to which brain regions are supplied with blood. Regions that are more active than other regions are supplied with more blood. This allows us to identify active regions compared to inactive regions. Based on this, we could check whether regions are more active depending on the program and task at hand. Since specific brain regions are responsible for specific reactions, we could test, for example, whether colored source code produces more positive emotion than text based annotations or whether AspectJ – due to its complexity – requires brain regions to be more active than AHEAD. These are interesting questions, because if a programmer is happier or needs fewer resources to solve a task, his performance can also be positively influenced.

Appendix A

Additional Material

In this appendix, we present some details that are relevant for replicating our experiment, but not necessary to understand our thesis. We show the items of our programming experience questionnaire and explain how we coded them in order to calculate one value indicating programming experience (with metric scale type). Then, we show the correct solution for the collaboration diagram for one of our experimental tasks. Last, show the task description for one task.

We applied our programming experience questionnaire six weeks before our experiment (cf. Section 5.3.1). In Tables A.1 and A.2, we show all items of our questionnaire. Table A.1 contains parameters we did not include in the programming experience score, but we assessed for other reasons (column *Purpose*). For example, the code word was necessary in order to assure anonymity for our subjects and to match the answers of the programming experience questionnaire to responses in our experiment.

Variable	Scaletype	Purpose
Code word	nominal	assure anonymity and match responses in questionnaire and experiment
Age	metric	describing our sample
Gender	nominal	creating homogeneous groups according to gender
Course of study	nominal	creating homogeneous groups according to study
Color blindness	nominal	assign colorblind subjects to CPP
Familiar domains	nominal	determine domains all subjects are unfamiliar with

Table A.1: Questions of programming experience questionnaire used for diverse purposes.

Table A.2 contains all variables that were assessed to measure programming experience. The column *Coding* describes how we coded the answers of our subjects in order to obtain a value for programming experience. The sum of all coded values yields the value we used to match our experimental groups regarding programming experience.

Variable	Scaletype	Coding
Enrollment year	metric	number of years until 2009
Years of programming	metric	number of years
Number of programming courses	ordinal	number of courses
Java	ordinal (Likert scale from 1-5)	value of the checked level
C	ordinal (Likert scale from 1-5)	1, if checked level is $>= 3$
Haskell	ordinal (Likert scale from 1-5)	1, if checked level is $>= 3$
Prolog	ordinal (Likert scale from 1-5)	1, if checked level is $>= 3$
Further programming languages (with experience $>= 3$)	metric	number of further programming languages
Functional paradigm	ordinal (Likert scale from 1-5)	1, if checked level is $>= 3$
Imperative paradigm	ordinal (Likert scale from 1-5)	1, if checked level is $>= 3$
Object-oriented paradigm	ordinal (Likert scale from 1-5)	1, if checked level is $>= 3$
Worked in a company	ordinal (yes or no)	no: 1, yes: two
Number of years in a company	metric	number of years
Size of projects (LOC)	ordinal ($<$ 900; 900-40,000; $>$ 40,000)	$<$ 900: 1; 900-40,000: 2; $>$ 40,000: 3
Programming experience compared to students of this course	ordinal (Likert scale from 1-5)	value of the checked level
Programming experience compared to programming experts	ordinal (Likert scale from 1-5)	value of the checked level

Meaning of Likert scale levels for programming languages and paradigms: 1: very inexperienced 2: inexperienced 3: mediocre 4: experienced 5: very experienced; Meaning of Likert scale levels for comparison with students/programming experts: 1: clearly worse 2: worse 3: identical 4: better 5: clearly better

Table A.2: Questions and their coding of programming experience questionnaire.

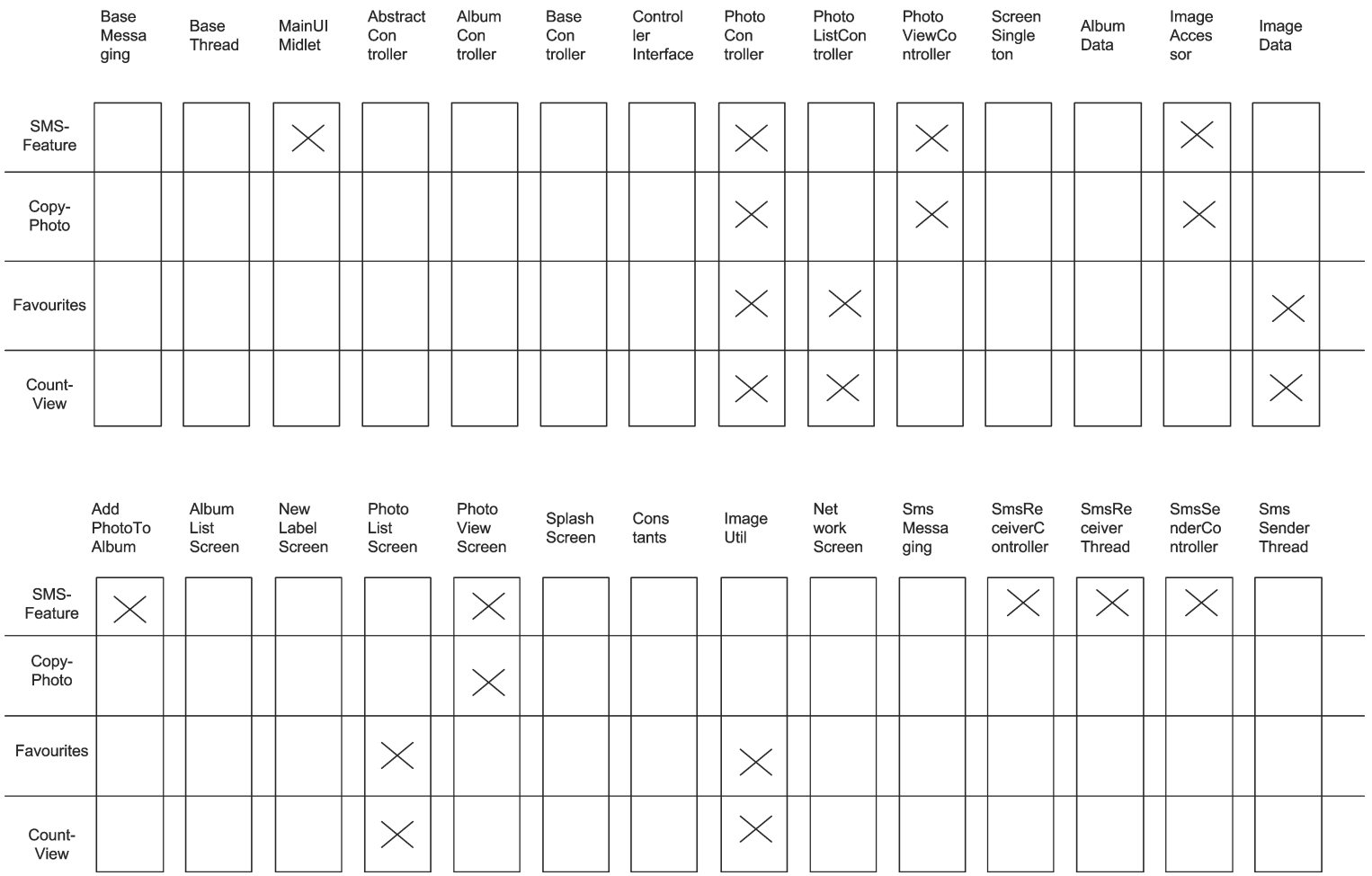
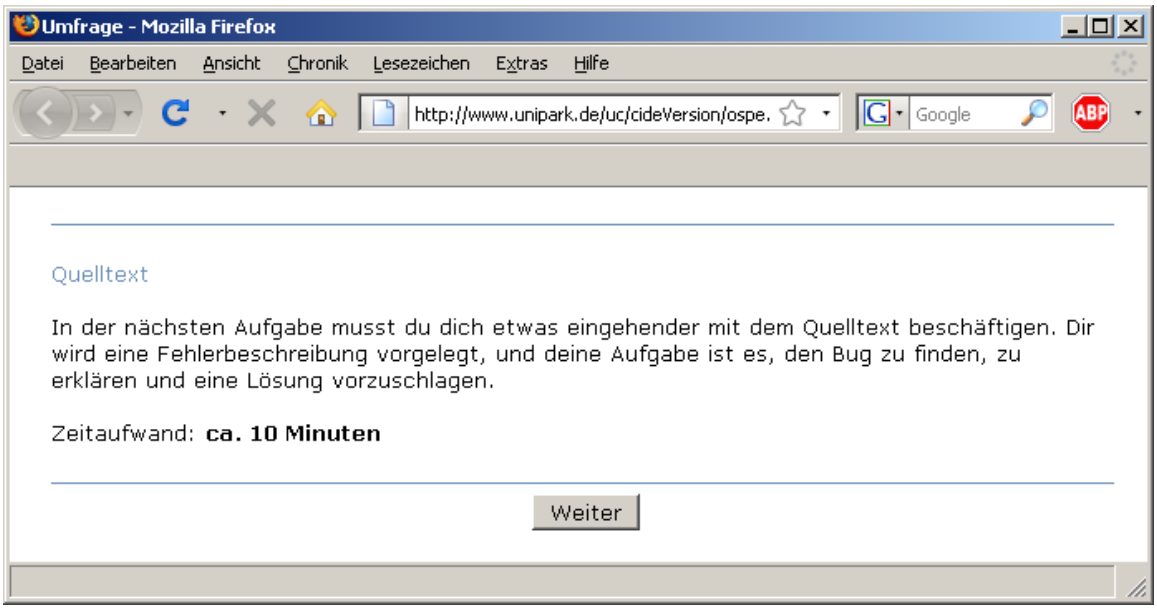


Figure A.1: Correctly marked collaboration diagram template.

Next, we show the correct solution for the collaboration diagram subjects had to create in one the first experimental task in Figure A.1 (S2, cf. Section 5.3.5). The roles of each class in a collaboration are marked with an 'x'.

Finally, we show the task description for the first maintenance task (M1, cf. Section 5.3.5) in Figures A.2 and A.3. In Figure A.2, the introduction to the task is depicted, which prepares subjects about what to expect. The task itself is described in Figure A.3, which shows the bug description and the three subtasks subjects had to complete. Additionally, the feature diagram and explanation of features of the MobileMedia SPL is depicted. For CIDE subjects, the explanation of features additionally contained the according background colors of the features in the source code. Furthermore, in the left upper screen, a link to the source code is displayed. The task descriptions are German, because our subjects were enrolled at a German university and course, such that we can be sure that all are experienced with the German language to understand our tasks. A translation for the text in Figure A.3 can be found in Section 5.3.5.



Translation: In the next task you have to work with the source code closely. You get a bug description and should locate the bug, explain, why it occurs and suggest a solution.
Time to solve this task: **about 10 Minutes**.

Figure A.2: Introduction to task M2.

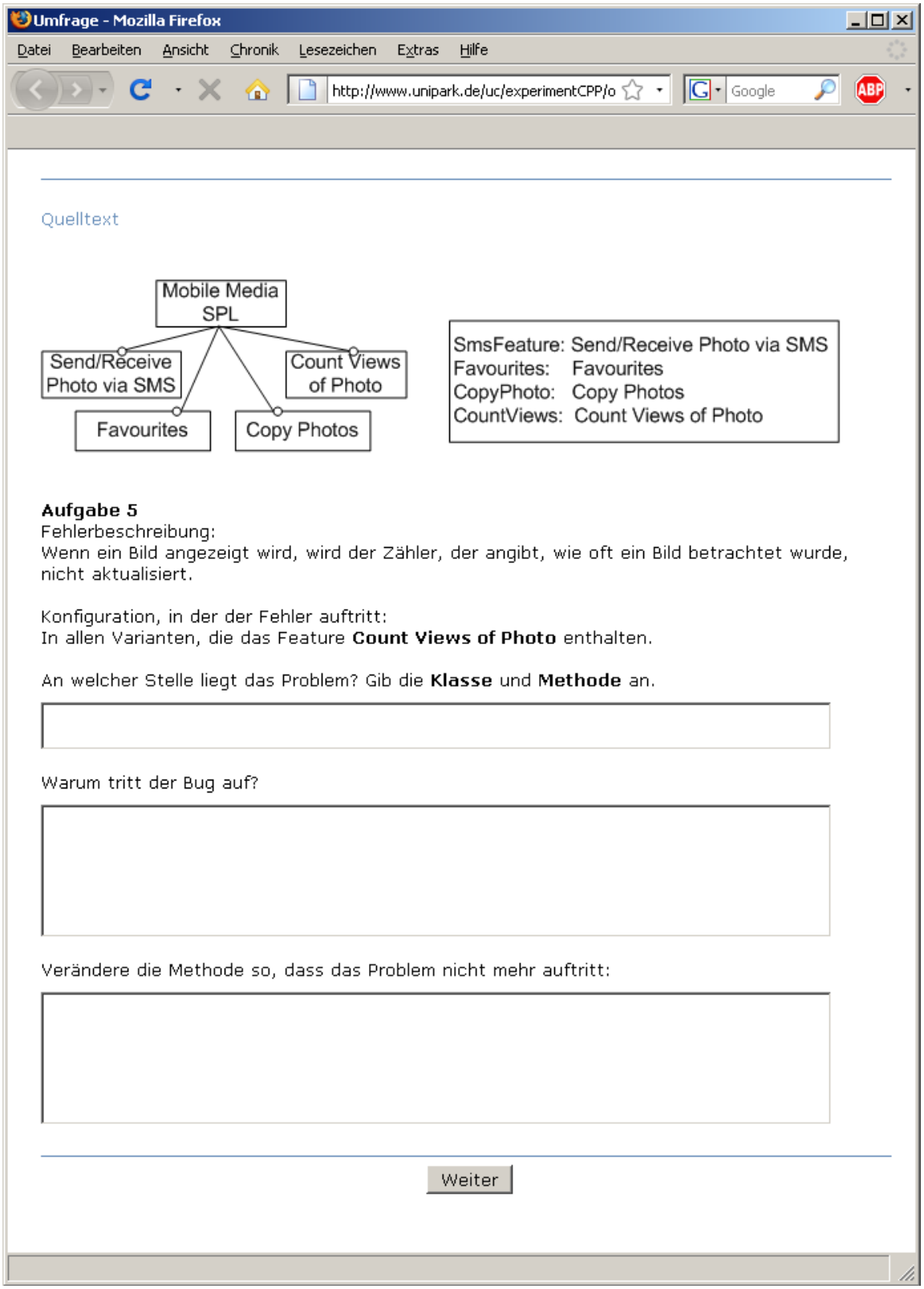


Figure A.3: Task description for M2.

Bibliography

[AB06] Sven Apel and Don Batory. When to Use Features and Aspects? A Case Study. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 59–68. ACM, 2006.

[ABPC91] Neil J. Anderson, Lyle Bachman, Kyle Perkins, and Andrew Cohen. An Exploratory Study into the Construct Validity of a Reading Comprehension Test: Triangulation of Data Sources. *Language Testing*, 8(1):41–66, 1991.

[AF96] Theodore W. Anderson and Jeremy D. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, pages 135–173. Springer, 2006.

[AK09] Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(4):1–36, 2009.

[AKGL09] Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Feature (De)composition in Functional Programming. In *SC '09: Proceedings of the 8th International Conference on Software Composition*, pages 9–26. Springer, 2009.

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[AKT07] Sven Apel, Christian Kästner, and Salvador Trujillo. On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns. In *ACoM '07: Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques*, pages 1–7. IEEE Computer Society, 2007.

[AL08] Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *SC '08: Proceedings of the 7th International Symposium on Software Composition*, pages 20–35. Springer, 2008.

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An Algebra for Features and Feature Composition. In *AMAST '08: Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, pages 36–50. Springer, 2008.

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE '05: Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[ALS08] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[AMGS05] Giuliano Antoniol, Ettore Merlo, Yann-Gaël Guéhéneuc, and Houari Sahraoui. On Feature Traceability in Object-Oriented Programs. In *TEFSE '05: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 73–78. ACM, 2005.

[Ape07] Sven Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, School of Computer Science, University of Magdeburg, 2007.

[Ass57] American Psychological Association. *Publication Manual of the American Psychological Association*. American Psychological Association, 1957.

[Ass09] American Psychological Association. *Publication Manual of the American Psychological Association*. American Psychological Association, sixth edition, 2009.

[Bar99] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly & Associates, Inc., 1999.

[BBB+57] John W. Backus, Robert J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, H. Stern, Irving Ziller, R. A. Hughes, and Roy Nutt. The FORTRAN Automatic Coding System. In *IRE-AIEE-ACM '57: Western Joint Computer Conference: Techniques for Reliability*, pages 188–198. ACM, 1957.

[BCVM02] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit Programming. In *AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 10–18. ACM, 2002.

[Bin03] Alfred Binet. *L'étude Expérimentale De l'Intelligence [The Experimental Study of Intelligence]*. Editions L'Harmattan, 1903.

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.

[BLS03] Don Batory, Jia Liu, and Jacob Neal Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 48–57. ACM, 2003.

[Bor00] Jürgen Bortz. *Statistik: für Human- und Sozialwissenschaftler*. Springer, sixth edition, 200.

[Boy77] John P. Boysen. *Factors Affecting Computer Program Comprehension*. PhD thesis, Iowa State University, 1977.

[Boy80] John P Boysen. Measuring Computer Program Comprehension. In *SIGCSE '80: Proceedings of the 11th SIGCSE Technical Symposium on Computer Science Education*, pages 92–102. ACM, 1980.

[BPSP04] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability Management with Feature Models. *Science of Computer Programming*, 53(3):333–352, 2004.

[Bri27] Percy W. Bridgman. *The Logic of Modern Physics*. Macmillan, 1927.

[Bro78] Ruven E. Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *ICSE '78: Proceedings of the 3rd International Conference on Software Engineering*, pages 196–201. IEEE Computer Society, 1978.

[Bro83] Ruven E. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[BSL99] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building Knowledge through Families of Experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[Bux01] Richard B. Buxton. *Introduction to Functional Magnetic Resonance Imaging: Principles and Techniques*. Cambrigde University Press, 2001.

[CC83] Jacob Cohen and Patricia Cohen. *Applied Multiple Regression: Correlation Analysis for the Behavioral Sciences*. Addison Wesley, second edition, 1983.

[CC04] David Coppit and Benjamin Cox. Software plans for separation of concerns. In *ACP4IS '04: Proceedings of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 26–31. ACM, 2004.

[CDFW80] William E. Carlson, Larry E. Druffel, David A. Fisher, and William A. Whitaker. Introducing Ada. In *ACM '80: Proceedings of the ACM 1980 Annual Conference*, pages 263–271. ACM, 1980.

[CFS41] Raymond B. Cattell, S. Norman Feingold, and Seymour B. Sarason. A Culture-free Intelligence Test: II. Evaluation of Cultural Influence on Test Performance. *Journal of Educational Psychology*, 32(2):81–100, 1941.

[CM60] Douglas P Crowne and David Marlowe. A New Scale of Social Desirability Independent of Psychopathology. *Journal of Consulting Psychology*, 24(4):349–354, 1960.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practice and Patterns*. Addison Wesley, fifth edition, 2001.

[CR06] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison Wesley, second edition, 2006.

[Cza98] Krzysztof Czarnecki. *Generative Programming Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technical University of Ilmenau, 1998.

[DBM+95] John W. Daly, Andrew Brooks, James Miller, Marc Roper, and Murray Ian Wood. The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study. In *ICSM '95: Proceedings of the 11th International Conference on Software Maintenance*, pages 20–29. IEEE Computer Society, 1995.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DR00] Alastair Dunsmore and Marc Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS 35-2000, Department of Computer Science, University of Strathclyde, Glasgow, UK, 2000.

[Eas97] Freda Easton. Educating the Whole Child, "Head, Heart, and Hands": Learning from the Waldorf Experience. *Theory into Practice*, 36(2):87–94, 1997.

[EB01] Alexander A. Evstiougov-Babaev. Call Graph and Control Flow Graph Visualization for Developers of Embedded Applications. In *Revised Lectures on Software Visualization, International Seminar*, pages 337–346. Springer, 2001.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[ESEES92] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. SeeSoft – A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.

[FC59] Leon Festinger and James M. Carlsmith. Cognitive Consequences of Forced Compliance. *Journal of Abnormal Psychology*, 58(2):203–210, 1959.

[FCM+08] Eduardo Figueiredo, Nelio Cacho, Mario Monteiro, Uira Kulesza, Ro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 261–270. ACM, 2008.

[FFM88] Lynn S. Fuchs, Douglas Fuchs, and Linn Maxwell. The Validity of Informal Reading Comprehension Measures. *Remedial and Special Education*, 9(2):20–28, 1988.

[FPG94] Norman Fenton, Shari L. Pfleeger, and Robert L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Softwar*, 11(4):86–95, 1994.

[FSD07] Scott D. Fleming, Kurt Stirewalt, and Laura K. Dillon. Using Program Families for Maintenance Experiments. In *ACoM '07: Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques*, pages 9–10. IEEE Computer Society, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, fifth edition, 1995.

[Gol02] Bruce E. Goldstein. *Sensation and Perception*. Cengage Learning Services, fifth edition, 2002.

[Goo99] C. James Goodwin. *Research in Psychology: Methods and Design*. Wiley Publishing, Inc., second edition, 1999.

[Gor77] Ronald D. Gordon. *A Measure of Mental Effort Related to Program Clarity*. PhD thesis, Purdue University, 1977.

[Gou65] Philip B. Gough. The Verification of Sentences. *Journal of Verbal Learning & Verbal Behavior*, 4(2):107–111, 1965.

[GVR02] Robert L. Glass, Iris Vessey, and Venkataraman Ramesh. Research in Software Engineering: An Analysis of the Literature. *Journal of Information and Software Technology*, 44(8):491–506, 2002.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

[Hal86] Diane F. Halpern. *Sex Differences in Cognitive Abilities*. Lawrence Erlbaum, 1986.

[HC01] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001.

[HCM09] Maen Hammad, Michael L. Collard, and Jonathan I. Maletic. Automatically Identifying Changes that Impact Code-to-Design Traceability. In *ICPC '09: Proceedings of the 17th International Conference on Program Comprehension*, pages 20–29. IEEE Computer Society, 2009.

[HH84] John E. Hunter and Ronda F. Hunter. Validity and Utility of Alternative Predictors of Job Performance. *Psychological Bulletin*, 76(1):72–93, 1984.

[HHL90] Sallie Henry, Matthew Humphrey, and John Lewis. Evaluation of the Maintainability of Object-Oriented Software. In *TENCON '90: IEEE Region 10 Conference on Computer and Communication Systems*, pages 404–409. IEEE Computer Society, 1990.

[HO93] William Harrison and Harold Ossher. Subject-oriented Programming: A Critique of Pure Objects. In *OOPSLA '93: Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. IEEE Computer Society, 1993.

[HPJW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbair, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language Version 1.2. *ACM SIGPLAN Notices*, 27(5):1–164, 1992.

[HSJ07] Miles Hewstone, Wolfgang Stroebe, and Klaus Jonas. *Introduction to Social Psychology*. Wiley Publishing, Inc., fourth edition, 2007.

[HW07] Alistair Hutton and Ray Welland. An Experiment Measuring the Effects of Maintenance Tasks on Program Knowledge. In *EASE '07: Proceedings of 11th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10. British Computer Society, 2007.

[JBZZ03] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. XVCL: XML-based Variant Configuration Language. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 810–811. IEEE Computer Society, 2003.

[JCP08] Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.

[JF88] Ralph. E. Johnson and Brian Foote. Designing Reuseable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JM01] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer, 2001.

[JP05] Andreas Jedlitschka and Dietmar Pfahl. Reporting Guidelines for Controlled Experiments in Software Engineering. In *ISESE '05: International Symposium on Empirical Software Engineering*, pages 95–104. IEEE Computer Society, 2005.

[JSB97] Adolf O. Jäger, Heinz-Martin Süß, and Andre Beauducel. *Berliner Intelligenzstruktur-Test*. Hogrefe, 1997.

[KA08] Christian Kästner and Sven Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *McGPLE '08: Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40. Department of Informatics and Mathematics, University of Passau, 2008.

[KAB07] Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 223–232. IEEE Computer Society, 2007.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 311–320. ACM, 2008.

[KAKB+08] Barbara Kitchenham, Hiyam Al-Khilidar, Muhammed Ali Babar, Mike Berry, Karl Cox, Jacky Keung, Felicia Kurniawati, Mark Staples, He Zhang, and Liming Zhu. Evaluating Guidelines for Reporting Empirical Software Engineering Studies. *Empirical Software Engineering*, 13(1):97–121, 2008.

[KAT+09] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *TOOLS EUROPE '09: Proceedings of the 47th International Conference Objects, Models, Components, Patterns*, pages 174–194. Springer, 2009.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990.

[KDHS09] Vigdis B. Kampenes, Tore Dybra, Jo E. Hannay, and Dag I. K. Sjoberg. A Systematic Review of Quasi-Experiments in Software Engineering. *Information and Software Technology*, 51(1):71–82, 2009.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer, 2001.

[KHM08] Huzefa Kagdi, Maen Hammad, and Jonathan I. Maletic. Who Can Help Me with this Source Code Change? In *ICSM '08: Proceedings of the 24th International Conference on Software Maintenance*, pages 157–166. IEEE Computer Society, 2008.

[Kim92] Doreen Kimura. Sex Differences in the Brain. *Scientific American*, 267(9):118–125, 1992.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Mae-da, Christina Lopez, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[KR91] Jürgen Koenemann and Scott P. Robertson. Expert Problem Solving Strategies for Program Comprehension. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 125–130. ACM, 1991.

[Kru08] Charles W. Krueger. The Software Product Line Lifecycle Framework, 2008. White Paper.

[KTA08] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing Software Product Line Variabilities in Source Code. In *ViSPLE '08: Proceedings of the 2nd International SPLC Workshop on Visualisation in Software Product Line Engineering*, pages 303–312. Lero, 2008.

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. FeatureIDE: Tool Framework for Feature-Oriented Software Development. In *ICSE '09: Proceedings of the 31th International Conference on Software Engineering*, pages 611–614. IEEE Computer Society, 2009.

[KU03] Andrew Jensen Ko and Bob Uttl. Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 175–184. IEEE Computer Society, 2003.

[Lan58] Henry A. Landsberger. *Hawthorne Revisited*. Cornell University, 1958.

[Lar44] Paul F. Larzarsfeld. The Controversy over Detailed Interviews – An Offer for Negotiation. *Public Opinion Quarterly*, 8(1):38–60, 1944.

[LHB01] Roberto E. Lopez-Herrejon and Don Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE '01: Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering*, pages 10–24. Springer, 2001.

[LHBC05] Roberto E. Lopez-Herrejon, Don Batory, and William Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 169–194. Springer, 2005.

[Lik32] Rensis Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[LR98] Gustav A. Lienert and Ulrich Raatz. *Testaufbau und Testanalyse*. BeltzPVU, sixth edition, 1998.

[LvM97] Steve Lang and Anneliese von Mayrhauser. Building a Research Infrastructure for Program Comprehension Observations. In *IWPC '97: Proceedings of the 5th International Workshop on Program Comprehension*, pages 165–169. IEEE Computer Society, 1997.

[MBOP83] Margaret G. McKeown, Isabel L. Beck, Richard C. Omanson, and Charles A. Perfetti. The Effects of Long-Term Vocabulary Instruction on Reading Comprehension: A Replication. *Journal of Reading Behavior*, 15(1):3–18, 1983.

[McC76] Thomas J. McCabe. A Complexity Measure. pages 308–320, 1976.

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, second edition, 2004.

[McL89] J. McLeod. *GAP Reading Comprehension Test*. Heinemann Publishers, 1989.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[Mez83] Karen Mezynski. Issues Concerning the Acquisition of Knowledge: Effects of Vocabulary Training on Reading Comprehension. *Review of Educational Research*, 53(2):253–279, 1983.

[MJ74] Eleanor E. Maccoby and Carol Nagy Jacklin. *The Psychology of Sex Differences*. Stanford University Press, 1974.

[MK07] Helfried Moosbrugger and Augustin Kelava. *Testtheorie und Fragebogenkonstruktion*. Springer, Berlin, 2007.

[MMW+01] Jutta Markgraf, Hans-Peter Musahl, Friedrich Wilkening, Karin Wilkening, and Viktor Sarris. Studieneinheit Versuchsplanung, 2001. FIM-Psychologie Modellversuch, Universität Erlangen-Nürnberg.

[MO04] Mira Mezini and Klaus Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *FSE '04: Proceedings of the 12th International Symposium on Foundations of Software Engineering*, pages 127–136. ACM, 2004.

[Moo96] Douglas G. Mook. *Motivation: The Organization of Action*. W.W. Norton & Co., second edition, 1996.

[Nay08] Keyvan Nayyeri. *Professional Visual Studio Extensibility*. Wiley Publishing, Inc., 2008.

[Neu45] John von Neumann. First Draft of a Report on the EDVAC, 1945.

[Par72] David Lorge Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Paw06] Renaud Pawlak. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distributed Systems Online*, 7(11):1, 2006.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[Pen87] Nancy Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[Pop59] Karl Popper. *The Logic of Scientific Discovery*. Routledge, 1959.

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[Pre98] Michael Pressley. *Reading Instruction That Works: The Case for Balanced Teaching*. The Guildford Press, 1998.

[PUPT02] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter F. Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Transactions on Software Engineering*, 28(6):595–606, 2002.

[QPGL05] Claudia Quaiser-Pohl, Christian Geiser, and Wolfgang Lehmann. The Relationship between Computer-Game Preference, Gender, and Mental-Rotation Ability. *Personality and Individual Differences*, 40(3):606–619, 2005.

[Rav36] John C. Raven. Mental Tests Used in Genetic Studies: The Performances of Related Individuals in Tests Mainly Educative and Mainly Reproductive. Master's thesis, University of London, 1936.

[RJ66] Robert Rosenthal and Lenore Jacobson. Teachers' Expectancies: Determinants of Pupils' IQ Gains. *Psychological Reports*, 19(1):115–118, 1966.

[RL04] Susan D. Ridgell and John W. Lounsbury. Predicting Academic Success: General Intelligence, "Big Five" Personality Traits, and Work Drive. *College Student Journal*, 38(4):607–619, 2004.

[RM02] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416. ACM, 2002.

[RM03] Martin P. Robillard and Gail C. Murphy. FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 822–823. IEEE Computer Society, 2003.

[Roe39] Fritz J. Roethlisberger. *Management and the Worker*. Harvard University Press, 1939.

[Sal92] David Salomon. *Assemblers and Loaders*. Ellis Horwood, 1992.

[SB98] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570. Springer, 1998.

[SBS94] Maarten W. Van Someren, Yvonne F. Barnard, and Jacobijn A. C. Sandberg. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.

[SCC02] William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin Company, 2002.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

[SHH+05] Dag I. K. Sjoberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, 2005.

[Shn77] Ben Shneiderman. Measuring Computer Program Quality and Comprehension. *International Journal of Man-Machine Studies*, 9(4):465–478, 1977.

[Slo66] Dan I. Slobin. Grammatical Transformations and Sentence Comprehension in Childhood and Adulthood. *Journal of Verbal Learning & Verbal Behavior*, 5(3):219–227, 1966.

[SM79] Ben Shneiderman and Richard Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Parallel Programming*, 8(3):219–238, 1979.

[SP96] Howard Schuman and Stanley Presser. *Questions and Answers in Attitude Surveys: Experiments on Question Form, Wording, and Context*. SAGE Publications, 1996.

[Ste06] Friedrich Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *OOPSLA '06: Proceedings of the 21st Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 481 – 497. ACM, 2006.

[Stu08] Student. The Probable Error of a Mean. *Biometrika*, 6(1):1–25, 1908.

[SV95] Teresa M. Shaft and Iris Vessey. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3):286–299, 1995.

[SW65] Samuel S. Shapiro and Martin. B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 1965.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

[Tic98] Walter F. Tichy. Should Computer Scientists Experiment More? *Computer*, 31(5):32–40, 1998.

[TLPH95] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[TO01] Peri Tarr and Harold Ossher. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 729–730. IEEE Computer Society, 2001.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.

[Tuk77] John W. Tukey. *Exploratory Data Analysis*. Addison Wesley, 1977.

[vMV93] Anneliese von Mayrhauser and A. Marie Vans. From Program Comprehension to Tool Requirements for an Industrial Environment. In *IWPC '93: Proceedings of 2nd IEEE Workshop on Program Comprehension*, pages 78–86. IEEE Computer Society, 1993.

[Wat96] Gloria S. Waters. The Measurement of Verbal Working Memory Capacity and Its Relation to Reading Comprehension. *The Quarterly Journal of Experimental Psychology Section A*, 49(1):51–79, 1996.

[Wec50] David Wechsler. *The Measurement of Adult Intelligence*. American Psychological Association, third edition, 1950.

[Wie03] Urban Wiesing. *Die Ethik-Kommissionen – Neuere Entwicklungen und Richtlinien*. Deutscher Ärzte-Verlag, 2003.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[Woo39] Robert Sessions Woodworth. *Experimental Psychology*. Henry Holt, 1939.

[Wun74] Wilhelm Wundt. *Grundzüge der Physiologischen Psychologie*. Engelmann, 1874.

[Wun14] Wilhelm Wundt. *Grundriß der Psychologie*. Kröner, 1914.

[Zoh03] Anat Zohar. Her Physics, His Physics: Gender Issues in Israeli Advanced Placement Physics Classes. *International Journal of Science Education*, 25(2):246–268, 2003.

[ZW97] Marvin V. Zelkowitz and Dolores Wallace. Experimental Validation in Software Engineering. *Information and Software Technology*, 39(11):735–743, 1997.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den August 3, 2009

Janet Feigenspan